# Hardness Results for Static Priority Real-Time Scheduling

Martin Stigge and Wang Yi
Uppsala University, Sweden
Email: {martin.stigge | yi}@it.uu.se

*Abstract*—**Real-time systems are often modeled as a collection of tasks, describing the structure of the processor's workload. In the literature, task-models of different expressiveness have been developed, ranging from the traditional periodic task model to highly expressive graph-based models.**

**For dynamic priority schedulers, it has been shown that the schedulability problem can be solved efficiently, even for graph-based models. However, the situation is less clear for the case of static priority schedulers. It has been believed that the problem can be solved in pseudo-polynomial time for the generalized multiframe model (GMF). The GMF model constitutes a compromise in expressiveness by allowing cycling through a static list of behaviors, but disallowing branching. Further, the problem complexity for more expressive models has been unknown so far.**

**In this paper, we show that previous results claiming that a precise and efficient test exists are wrong, giving a counterexample. We prove that the schedulability problem for GMF models (and thus also all more expressive models) using static priority schedulers is in fact coNP-hard in the strong sense. Our result thus establishes the fundamental hardness of analyzing static priority real-time scheduling, in contrast to its dynamic priority counterpart of pseudo-polynomial complexity.**

## I. Introduction

In scheduling theory, the system model usually consists of two parts: a workload model and a scheduling policy. One major objective is to check the schedulability of their composition, i.e., whether all timing constraints of the workload can be met if the given scheduler is used. The analysis complexity depends on the choices for the model and the scheduler.

The traditional *Liu and Layland task model* [1] is a workload model which assumes a collection of independent tasks with periodic activation. Behaviors that are not entirely periodic can not be expressed accurately with this model, so more expressive models have been proposed in recent years. One of the most expressive models is the *Digraph Real-Time task model (DRT)* [2] using arbitrary directed graphs for modeling task activations. For the scheduling policy, two classes are of practical importance: *static* and *dynamic priority* schedulers. Static priority schedulers keep relative task priorities fixed at runtime and are often preferred in implementations because of robustness and code complexity properties. Dynamic priority schedulers can change priorities at runtime, which makes them theoretically more powerful but also more complex.

Schedulability decision procedures for both scheduler classes are using different techniques. It has been shown that schedulability analysis for the DRT model with dynamic priority schedulers is tractable for uniprocessor platforms, using demand bound functions [2]. However, for static priority schedulers, response-time based techniques are usually applied, and the problem complexity for the DRT model has been unknown. For the *generalized multiframe model (GMF)* [3], which is a subclass of the DRT model disallowing branches, it has been believed that there is an exact schedulability test for static priority schedulers running in pseudo-polynomial time [4]. In this paper, we show that this is actually not the case. In particular, we provide the following contributions:

- We consider the schedulability problem for tree-based release structures, a sub-class of the DRT model. For this class we define a *synchronous roots sequence* and identify it as the worst case for the situations under consideration.
- Using this insight, we give a proof of strong *coNP*-hardness of the schedulability problem on uniprocessors for all models at least as expressive as GMF.

Our result provides a general conclusion about the hardness of static priority schedulability analysis. An important implication is that fully polynomial-time approximation schemes (FPTAS) for this problem cannot exist (assuming $P \neq NP$).

### A. Prior Work

When first published in [1], the *periodic task model* by Liu and Layland could be analyzed with a precise test for schedulability with dynamic priority schedulers in polynomial time. In this model, each task releases jobs with the same worst-case execution time (WCET) bound, and with constant minimal inter-release delays equal to the jobs' deadlines. The case of static priority schedulers can be handled in pseudo-polynomial time with Response-Time Analysis first introduced in [5]. A more expressive task model is the *generalized multiframe (GMF)* model [3] which allows a task to cycle through a static list of job types, each with potentially different WCET bounds and deadlines. For this model, the schedulability problem with dynamic priority schedulers has been shown to have a tractable solution. The most general model to date with such a test is the *Digraph Real-Time task model (DRT)* [2]. It models each task with a directed graph in which vertices represent job releases and edges represent branching structures and inter-release delays. A recent extension to this model with global timing constraints [6] shows the tractability borderline regarding schedulability with dynamic priority schedulers.

In the case of static priorities, sufficient tests for DAG-based release structures have been introduced in [7]. A polynomial time test for GMF models is given in [4] by showing an analysis method claimed to be precise. We show in Appendix B that it is in fact only a sufficient test. Further, [8] provides some intuition why the problem might be intractable already for GMF models, but does not supply a proof.

## II. Preliminaries

In this section we introduce syntax and semantics of the task model, the schedulability problem under consideration, and the reduction framework which we use to prove our hardness result.

### A. Task Model

We use the *digraph real-time (DRT) task model* [2] to describe the workload of a system. A DRT task set $\tau = \{T_1, \ldots, T_N\}$ consists of $N$ independent tasks. A task $T$ is represented by a *directed graph* $G(T)$ with both vertex and edge labels. The vertices $\{v_1, \ldots, v_n\}$ of $G(T)$ represent the types of all the jobs that $T$ can release. Each vertex $v_i$ is labeled with an ordered pair $\langle e(v_i), d(v_i)\rangle$ denoting worst-case execution-time demand $e(v_i)$ and relative deadline $d(v_i)$ of the corresponding job. Both values are assumed to be non-negative integers. The edges of $G(T)$ represent the order in which jobs generated by $T$ are released. Each edge $(u, v)$ is labeled with a non-negative integer $p(u, v)$ denoting the minimum job inter-release separation time. We do not assume a relation between job deadlines $d(u)$ and inter-release separation times $p(u, v)$, i.e., the jobs may have *arbitrary deadlines*.

**Example II.1.** *Figure 1 shows an example of a DRT task.*

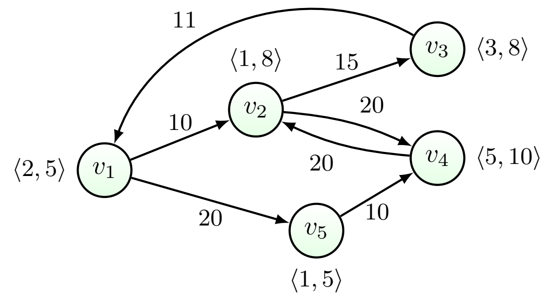


Fig. 1. An example task containing five different types of jobs

*Semantics:* An execution of task $T$ corresponds to a potentially infinite path in $G(T)$. Each visit to a vertex along that path triggers the release of a job with parameters specified by the vertex label. The job releases are constrained by inter-release separation times specified by the edge labels. Formally, we use a 3-tuple $(r, e, d)$ to denote a *job* that is released at (absolute) time $r$, with worst-case execution time $e$ and deadline at (absolute) time $d$. We assume dense time, i.e., $r, e, d \in \mathbb{R}_{\geqslant 0}$. A job sequence[1] $\rho = [(r_1, e_1, d_1), (r_2, e_2, d_2), \ldots]$ is *generated by* $T$, if and only if there is a (potentially infinite) path $\pi = (\pi_1, \pi_2, \ldots)$ in $G(T)$ satisfying for all $i$:

1) $e_i \leqslant e(\pi_i)$,
2) $d_i = r_i + d(\pi_i)$,
3) $r_{i+1} - r_i \geqslant p(\pi_i, \pi_{i+1})$.

For a task set $\tau$, a job sequence $\rho$ is *generated by* $\tau$, if it is a composition of sequences $\{\rho_T\}_{T\in\tau}$, which are individually generated by the tasks $T$ of $\tau$.

[1]Technically, a job sequence is a *set* of jobs. Still, we keep the name for historic reasons.

**Example II.2.** *For the example task $T$ in Figure 1, consider the job sequence $\rho = [(5, 5, 15), (25, 1, 33), (42, 3, 50)]$. It corresponds to path $\pi = (v_4, v_2, v_3)$ in $G(T)$ and is thus generated by $T$.*

*Note that this example demonstrates the "sporadic" behavior allowed by the semantics of our model. While the second job in $\rho$ (associated with $v_2$) is released as early as possible after the first job ($v_4$), the same is not true for the third job ($v_3$).*

*Restricted Models:* For many systems, it is not necessary to have the full power of arbitrary directed graphs in order to model their behavior accurately. By restricting the class of graphs under consideration, analysis methods can potentially be optimized and thus more efficient. We introduce two established sub-classes for which we show our hardness results, despite their restrictions.

- A *branching task* is a DRT task $T$ for which $G(T)$ is a *directed tree*. Intuitively, a branching task introduces (non-deterministic) branching behavior into the modeling framework, but in this simple form, only finite behavior is allowed. That is, all job sequences generated by a branching task are finite.
- A *generalized multi-frame (GMF) task* [3] is a DRT task $T$ for which $G(T)$ is a *cycle graph*. Intuitively, a GMF task is even more restricted by disallowing branching. However, it still allows to model (infinite) cycling through a static list of job types. This pattern is often found in safety critical real-time systems.

**Example II.3.** *Figure 2 shows examples for branching and GMF tasks, respectively.*

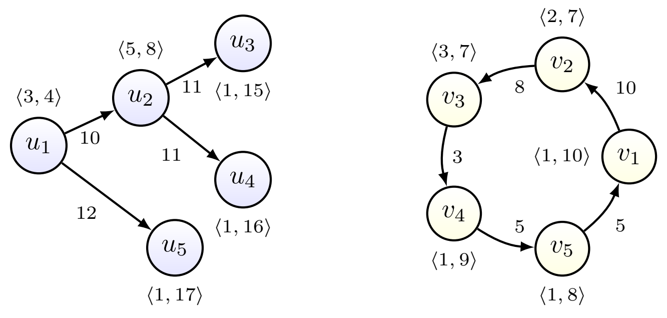


Fig. 2. Examples for branching and GMF tasks

### B. Schedulability

The central problem we are interested in is the standard notion of schedulability: Can a given scheduler schedule all the workload produced by the system?

**Definition II.4** (Schedulability)**.** *A task set $\tau$ is* schedulable *with* scheduler *Sch, if and only if for all job sequences generated by $\tau$, all jobs meet their deadlines when scheduled with Sch. Otherwise, $\tau$ is* unschedulable *with Sch.*

In this work, we assume preemptive scheduling on uniprocessor systems. For analysing schedulability, one usually distinguishes between *dynamic* and *static* priority schedulers. It is well-known that for our setting of independent jobs, the earliest deadline first (EDF) scheduler using dynamic priorities is optimal, i.e., if a task set can be scheduled by any scheduler, it can also be scheduled by EDF. It has been shown that for EDF, schedulability of DRT task sets can be checked in pseudo-polynomial time [2]. Even for an extension of DRT tasks with global timing constraints, this problem has been shown to be tractable [6].

Surprisingly, and as a sharp contrast to these results, it has been unknown whether schedulability can be checked efficiently for *static priority* schedulers, even for the very restricted class of GMF task sets. An earlier attempt to present a polynomial solution [4] unfortunately turned out to be only a sufficient test. We give details in Appendix B why this test is not necessary and thus may fail to positively identify schedulable task sets.

For the rest of the paper, we assume static priority scheduling in which each task set $\tau$ is equipped with a priority order $pr : \tau \to \mathbb{N}$, that assigns a unique priority to each task (with lower numbers for higher priorities). We will show that given a DRT task with a priority order, the schedulability problem with a static priority scheduler is $coNP$-hard in the strong sense. This holds even for the restricted classes of branching (Section III) and GMF task sets (Section IV).

### *C. Reduction Framework*

For our hardness proof, we use the standard notion of a polynomial-time many-one reduction [10], also called "Karp-reduction". We will provide a reduction from the 3-PARTITION problem.

**Definition II.5.** *An instance $I = (A, s, B)$ of* 3-PARTITION *consists of*

1) *a set $A = \{a_1, \dots, a_{3m}\}$ of $3m$ elements,*
2) *a* size function $s : A \to \mathbb{N}$, *and*
3) *a* bound $B \in \mathbb{N}$,

*such that $\sum_{i=1}^{3m} s(a_i) = m \cdot B$ and $B/4 < s(a) < B/2$ for all $a \in A$. An instance $I$ is a* positive instance *if $A$ can be partitioned into $m$ disjoint sets $P_1, \dots, P_m$ such that $\sum_{a \in P_j} s(a) = B$ for all $P_j$.*

It follows from the definition that in case of a positive instance, all partitions $P_j$ contain exactly 3 elements. Therefore, we call these $P_1, \dots, P_m$ derived from a positive instance also a *valid 3-partition*.

A problem can be solved in *pseudo-polynomial time*, if there is a polynomial time algorithm deciding membership for an instance representation where all values are encoded in unary. (This is equivalent to the requirement that polynomial bounds exist for all values.) Problems of that complexity are considered tractable in the real-time systems community. Likewise, a problem is *NP-hard in the strong sense* if it stays $NP$-hard even when all values are encoded in unary. It has been shown that 3-PARTITION is strongly $NP$-hard [10]. Thus, assuming $P \neq NP$, there can not be a pseudo-polynomial time algorithm for 3-PARTITION. We will show the same for the static priority schedulability problem.

## III. Hardness for Tree Models

We now present our $coNP$-hardness proof of the static priority scheduling problem for branching task models, i.e., tree release structures, and first give a proof overview. Given a 3-PARTITION instance $I$, we construct a branching task set $\tau$ with a priority order $pr$ in Section III-A. We construct $\tau$ such that it is *unschedulable* with any static priority scheduler if and only if $I$ is a positive instance, i.e., a 3-partition exists.

- First, we show in Section III-B that a positive instance $I$ results in an unschedulable $\tau$.
- Second, we show in Section III-C that if $\tau$ is missing a deadline in a *synchronous roots sequence* (SRS), then $I$ is a positive instance. In an SRS, all tasks are releasing their roots exactly at the same time.
- Third, we show in Section III-D that the SRS assumption can be removed if we add a minor extension to the task set construction. For the extended task set, it will be the case that if $\tau$ misses a deadline for some job sequence, it can also miss a deadline for a synchronous roots sequence.

Note that the reduction must be polynomial-time and all values in $\tau$ have to be polynomially bounded in the parameters of $I$.

### *A. Task Set Construction*

Given an instance $I = (A, s, B)$ of the 3-PARTITION problem, we construct a task set $\tau(I)$. The idea is to have one task per element $a_i \in A$. Each task has a non-deterministic choice for representing its assignment to one of the $m$ partitions. The choices differ in delay and execution time which are constructed exactly so that only a valid 3-partition can cause the processor to be constantly busy for a long time. Finally, an additional task with lowest priority is constructed, receiving interference from all other tasks. It has a very short execution time, so it will only miss its deadline if the processor is constantly busy for a sufficiently long time. We will see that only in the case of a valid 3-partition, the processor can be kept busy for that long.

We now give the construction details.

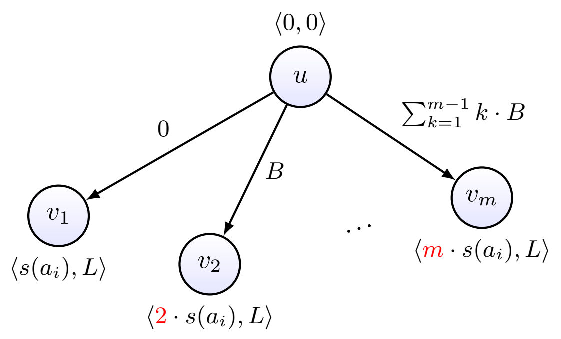
1) We first define a constant $L$ that is used as a deadline which is sufficiently long such that it can not be missed.

$$L := \sum_{i=1}^{3m} m \cdot s(a_i) = m^2 B$$

2) For each $a_i \in A$ with $i = 1, \dots, 3m$ we construct a task $T_i$ with priority $pr(T_i) := i$ as follows. It is a tree with a root vertex $u$ and $m$ children $v_1, \dots, v_m$ which are also the leaves of the tree. The labels are as follows:

$$e(u) := 0, \qquad d(u) := 0,$$

$$e(v_j) := j \cdot s(a_i), \quad d(v_j) := L, \quad p(u, v_j) := \sum_{k=1}^{j-1} k \cdot B.$$



$\langle 0, 0\rangle$
$u$
$0$
$B$
$\sum_{k=1}^{m-1} k \cdot B$
$v_1$
$v_2$
$\cdots$
$v_m$
$\langle s(a_i), L\rangle$
$\langle 2 \cdot s(a_i), L\rangle$
$\langle m \cdot s(a_i), L\rangle$

Fig. 3. Illustration of task construction $T_i$. Note the *scaling factor* for the execution times.

Note that $p(u, v_1) = 0$. We illustrate the construction in Figure 3. The idea is that a branch from dummy vertex $u$ to vertex $v_j$ represents partition $P_j$. Task $T_i$ chooses (non-deterministically) to branch to one of the leaves $v_j$, thereby expressing $a_i \in P_j$. The different delays are constructed so that all tasks choosing partition $P_j$ will have their job released and executed in a *window* of size $j \cdot B$. These $m$ windows are adjacent and the execution times will guarantee that only the existence of a valid 3-partition for $I$ can create a situation in which the processor is continuously busy during all windows. The leaf jobs' execution times are equal to the element size $s(a_i)$ but they are *scaled* by a factor corresponding to the index of the branch. This is the central idea to prevent tasks from choosing the "wrong" branch and thus the "wrong" partition.

3) Further, we construct another task $T_{low}$ with the lowest priority, i.e., $pr(T_{low}) := 3m + 1$. The idea is that $T_{low}$ is the task missing a deadline if and only if there is a valid 3-partition for $I$. Its graph $G(T_{low})$ contains only one single vertex $v$. The labels are

$$e(v) := 1, \quad d(v) := \sum_{j=1}^{m} j \cdot B.$$

The deadline $d(v)$ is the sum of all window sizes, so $T_{low}$ may miss its deadline if all windows are busy.

**Example III.1.** *Consider an instance $I = (A, s, B)$ with $A = \{a_1, \ldots, a_9\}$, $B = 42$ and the sizes $s$ given via:*

| $a \in A$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ |
|---|---|---|---|---|---|---|---|---|---|
| $s(a)$ | 15 | 16 | 12 | 16 | 14 | 11 | 15 | 13 | 14 |

*The resulting task set $\tau(I)$ is shown in Figure 4.*

### *B. Partition implies Deadline Miss*

We first show the potential deadline miss of $T_{low}$ in case of a positive instance $I$.

**Lemma III.2.** *For a positive instance $I$ of* 3-PARTITION, *task set $\tau(I)$ is not schedulable with a static priority scheduler.*

*Proof:* Let $P_1, \ldots, P_m$ be a valid 3-partition of $I = (A, s, B)$. For each $a_i \in A$ with $i = 1, \ldots, 3m$ we construct

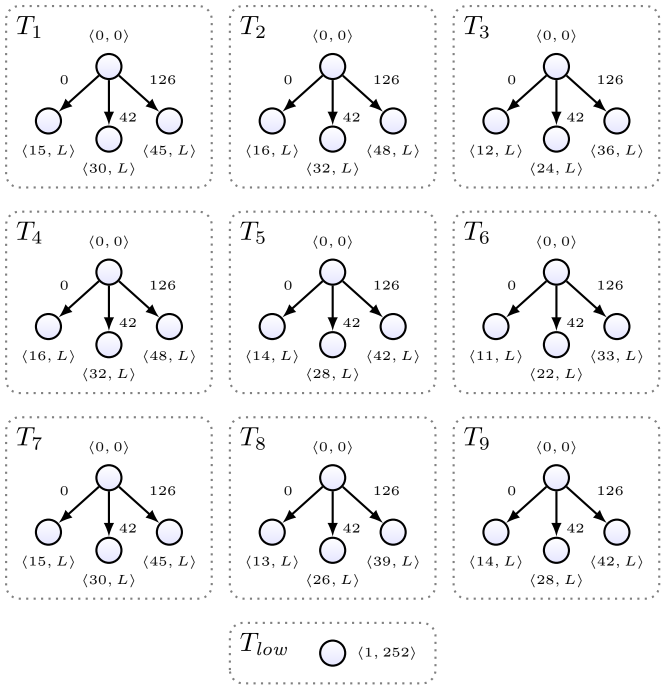


Fig. 4. Example task set $\tau(I)$ constructed from the 3-PARTITION instance $I$ in Example III.1. We have $L = 378$ in this example.

a job sequence for the corresponding task $T_i$. Let $P_j$ be the partition containing $a_i$. The idea is that the dummy root job is released at time 0 and $T_i$ then branches[2] according to partition $P_j$, releasing its job corresponding to $v_j$ in the $j$-th window. Formally, the constructed job sequence is

$$\rho_i := [(0, 0, 0), (r_i, j \cdot s(a_i), r_i + L)], \text{with}$$
$$r_i := \sum_{k=1}^{j-1} k \cdot B.$$

Note that $r_i$ is the start time of the $j$-th window. Further, we release $T_{low}$ also at time 0, which causes it to have a deadline exactly when the $m$-th window ends. The corresponding job sequence is

$$\rho_{low} := [(0, 1, \textstyle\sum_{k=1}^{m} k \cdot B)].$$

If these sequences are composed to a job sequence $\rho$ generated by $\tau$ and scheduled with a static priority scheduler, task $T_{low}$ will miss its deadline. The reason is that for each window $j \in \{1, \ldots, m\}$, the processor is busy during the whole window as follows.

- Window $j$ is of size $j \cdot B$ and starts at time $\sum_{k=1}^{j-1} k \cdot B$. This starting time is the same time point at which window $j - 1$ ends.
- Right when window $j$ starts, the non-zero jobs of all tasks $T_i$ with $a_i \in P_j$ are released. Since this is a 3-partition, these must correspond to three elements $a, b, c \in A$.

[2]Note that this lemma can be proved in a simpler way by just releasing all jobs corresponding to vertices $v_m$ of all tasks at the same time as $T_{low}$. However, our presented proof also holds for the extended task construction in Section III-D, where the simpler version would not suffice.

- We know that $s(a)+s(b)+s(c)=B$ since we chose them from a valid 3-partition.
- The corresponding jobs released at the start of window $j$ have execution times $j \cdot s(a)$, $j \cdot s(b)$ and $j \cdot s(c)$, respectively.
- In summary, window $j$ has a workload of $j \cdot s(a) + j \cdot s(b) + j \cdot s(c) = j \cdot B$ which is exactly the window size.
- With the processor being busy from time 0 until the end of the last window $m$, i.e., time $\sum_{k=1}^{m} k \cdot B$, we see that $T_{low}$ is never executed before its deadline and will therefore miss it.

■

The deadline miss scenario in the above proof can be illustrated with the following example.

**Example III.3.** *Consider the* 3-PARTITION *instance $I$ from Example III.1. It is clearly a positive instance, since the following is a valid 3-partition:*

$$P_1 = \{a_1, a_2, a_6\}, P_2 = \{a_3, a_4, a_5\}, P_3 = \{a_7, a_8, a_9\}.$$

*We illustrate the resulting schedule described in the above proof in Figure 5. Note that the job execution times exactly fit the three windows. Thus, the processor is busy between time* 0 *and* 252 *executing jobs from $T_1, \ldots, T_9$ with higher priorities than $T_{low}$. Since $T_{low}$ has its deadline at time* 252*, it misses its deadline.*

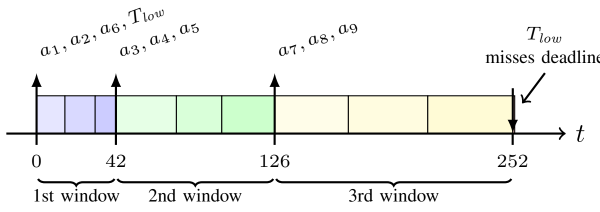


Fig. 5. Schedule resulting from the valid 3-partition in Example III.3. The arrows representing job releases are annotated with the elements $a \in A$ to which the released jobs correspond, in addition to the job of $T_{low}$. Different colors represent task releases in different windows, i.e., assignment to partitions. Note that $T_{low}$ has an execution time of just 1, so it is very slim in the figure.

### C. Deadline Miss with SRS

We focus now on the other direction: the construction of a valid 3-partition from a job sequence $\rho$ in which a task in $\tau(I)$ misses its deadline. The idea is that schedules which do *not* correspond to a valid 3-partition can not cause the processor to be busy in all windows. We illustrate this with the following example.

**Example III.4.** *We use instance $I$ from Example III.1. Consider the following 3-partition which is not valid:*

$$P_1' = \{a_1, a_2, a_4\}, P_2' = \{a_6, a_8, a_9\}, P_3' = \{a_3, a_5, a_7\}.$$

*A schedule according to the construction above in the proof of Lemma III.2 will make all jobs meet their deadlines, since the second window is not entirely busy. We illustrate the schedule in Figure 6.*

*In the remainder of this section, we show that if a schedule does not correspond to a valid 3-partition, some window is not entirely busy and $T_{low}$ can execute*[3]*. Thus, $T_{low}$ will always meet all deadlines for negative* 3-PARTITION *instances.*

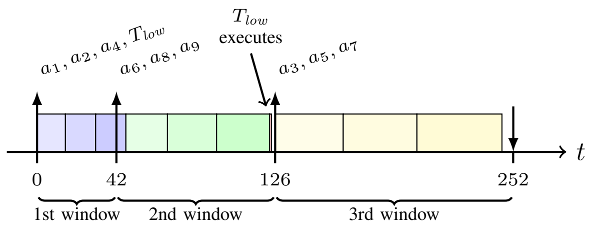


Fig. 6. Schedule resulting from the *invalid* 3-partition in Example III.4. In the end of the second window, the processor does not execute a job of higher priority than $T_{low}$, so $T_{low}$ can execute and thus meets its deadline.

We first establish the proof under the assumption that $\rho$ is a *synchronous roots sequence*.

**Definition III.5.** *For a branching task set $\tau$, a job sequence $\rho$ is a* synchronous roots sequence (SRS) *if for each task $T \in \tau$, the subsequence $\rho_T$ generated by task $T$ satisfies the following:*

1) *The first job in $\rho_T$ corresponds to the root vertex of $G(T)$ and is released at time* 0.
2) *All later jobs in $\rho_T$ are released as early as possible.*

This assumption is necessary: otherwise, all tasks $T_i$ of $\tau(I)$ could release their job associated with vertex $v_m$ at the same time as $T_{low}$ is released. This would clearly cause a deadline miss of $T_{low}$, no matter whether $I$ is a positive or a negative instance. Assuming an SRS is the key to prevent such a situation. (However, the SRS assumption is removed in Section III-D by extending the task set construction.)

**Lemma III.6.** *If there is a synchronous roots sequence $\rho$ of $\tau(I)$ which is not schedulable with a static priority scheduler, then $I$ is a positive instance of* 3-PARTITION.

*Proof:* First we note that it must be the job of $T_{low}$ which is missing its deadline. The deadlines of all other jobs are either 0 (root dummy vertices with 0 execution time) or $L$ (leaf vertices), which is larger than all possibly interfering workload of higher priority tasks. Further, without loss of generality, we may assume that all jobs in $\rho$ are taking the maximum execution time allowed by their releasing vertices. This assumption is safe since increasing the interfering workload will not prevent a deadline miss that is otherwise happening.

The job of $T_{low}$ that is missing its deadline is released at time 0 and has its deadline at time $D := \sum_{k=1}^{m} k \cdot B$. Since

[3]Generally, a schedule does not need to conform to the construction in the proof of Lemma III.2. That is, tasks may start execution anywhere in the tree at any time. However, it will be established in Section III-D that it is sufficient to only consider such schedules where the execution starts synchronously at the root vertices.

it is missing the deadline, the processor is *busy* executing higher priority jobs for strictly more than $D-1$ time units. For presentation reasons we obtain a job sequence $\rho'$ by removing the job of $T_{low}$ from $\rho$. For $\rho'$ we know that the processor is *idle* during $[0, D]$ for strictly less than one time unit. We now use the window concept from above for dividing the time interval $[0, D]$ into $m$ non-overlapping windows. The $j$-th window is of size $j \cdot B$ and starts at time $\sum_{k=1}^{j-1} k \cdot B$. These time points are exactly the time points in $\rho'$ at which all tasks $T_i$ release their non-dummy jobs (because of the SRS assumption). We will now show that during all windows, $\rho'$ keeps the processor constantly busy. Based on this we will construct a valid 3-partition by observing which branch each task $T_i$ chose in $\rho'$.

We first look at the first window. It starts at time 0 and ends at time $B$. Jobs are only released at time 0 and then not again before $B$. This is guaranteed by the inter-release separation constraints on the tree edges and the SRS assumption. We further know that with job sequence $\rho'$, the processor is idle for strictly less than one time unit during the first window. All execution times are integers, so the processor is in fact not idle at all, or in other words, busy executing jobs throughout the whole window. In particular, there is no idle time at the end of the window. Let $\sigma_1$ denote the sum of all $s(a_i)$ for which task $T_i$ chose the *first branch* in $\rho'$. Since the processor is busy throughout the whole window (and possibly even afterwards) executing jobs released at time 0, we have $\sigma_1 \geqslant B$.

The second window has a size of $2B$, starting at time $B$ and ending at time $B + 2B = 3B$. As in the first window, job releases can only take place at the start of the window and then not again until its end. Since the processor is also busy during this whole window by the same reasoning as above, we again know that there is no idle time at the end of the window. We let $\sigma_2$ denote the sum of all $s(a_i)$ for tasks $T_i$ that chose the *second branch* in $\rho'$. The sum of these jobs' execution times is thus $2\sigma_2$ since the execution times of $v_2$ in all trees are scaled by a factor of 2. Taking into consideration that there may be some workload reaching from the first window into the second window, we can express our knowledge that there is no idle time at the end of the second window as $\sigma_1 + 2\sigma_2 \geqslant B + 2B$.

We now generalize this to the $k$-th window. For every $j = 1, \ldots, m$, let $\sigma_j$ denote the sum of all $s(a_i)$ for which task $T_i$ chose the *j-th branch* in $\rho'$. The sum of the jobs' execution times in this branch is $j \cdot \sigma_j$ because of the scaling factor during task construction. Similar to the above reasoning we know that there is no idle time at the end of the $k$-th window, which we can express as

$$\sum_{j=1}^{k} j \cdot \sigma_j \geqslant \sum_{j=1}^{k} j \cdot B. \tag{A}$$

Additionally, we know that the sum of all $s(a_i)$ is $m \cdot B$, so the sum of all $\sigma_j$ is $m \cdot B$ as well:

$$\sum_{j=1}^{m} \sigma_j = m \cdot B \tag{B}$$

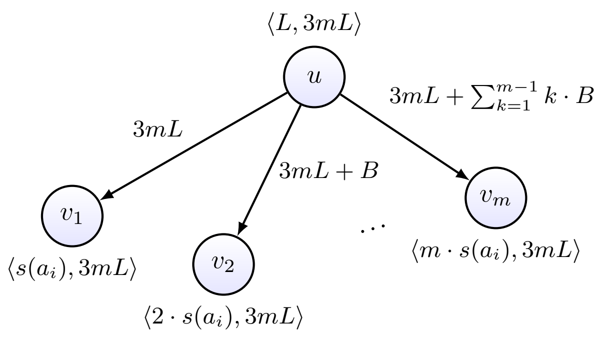

$\langle L, 3mL\rangle$
$u$
$3mL$
$3mL + B$
$3mL + \sum_{k=1}^{m-1} k \cdot B$
$v_1$
$v_2$
$\cdots$
$v_m$
$\langle s(a_i), 3mL\rangle$
$\langle 2 \cdot s(a_i), 3mL\rangle$
$\langle m \cdot s(a_i), 3mL\rangle$

Fig. 7. Illustration of new task construction $T_i$. Note the heavy job released at root vertex $u$.

From both conditions it is easily derived that $\sigma_j = B$ for all $j$. (See Appendix A for details.) This means that for each branch $j$, the sum of the $s(a_i)$ corresponding to those tasks $T_i$ that chose this branch $j$ in $\rho'$ is exactly $B$. In other words, for each $j$, the set of all elements $a_i$ contributing to $\sigma_j$ is the set $P_j$ of a valid 3-partition. ■

### *D. Deadline Miss without SRS Assumption*

In order to show the result without the assumption of a synchronous roots sequence, we need to adjust the construction of task set $\tau(I)$. Recall that this assumption was necessary to prevent the tasks from just directly releasing jobs associated with $v_m$ causing high workload. As a solution to this issue, we add a very long job to the root vertex of all $G(T_i)$. We call this job the *heavy job* of task $T_i$, in contrast to the *leaf jobs* of $T_i$. As we will see, the introduction of this heavy job guarantees that the worst case interference for $T_{low}$ is created when all $T_i$ synchronously release the jobs associated with their root vertices (which are now their heavy jobs). This consequently results in the SRS from above.

The new task set $\tau'(I)$ differs from the original $\tau(I)$ only in the root vertices of the release trees and adjusted delays and deadlines. The details are as follows.

1) We again use the large constant $L$ defined above. It is sufficiently large to represent a time span which is just as long as the workload that all leaves $v_m$ can create if they are released together.

$$L = \sum_{i=1}^{3m} m \cdot s(a_i) = m^2 B$$

2) For each $a_i \in A$ we construct a task $T_i$ with priority $pr(T_i) := i$ as in Section III-A, but with the following labels:

$$\begin{aligned} e(u) &:= L, & d(u) &:= 3mL, \\ e(v_j) &:= j \cdot s(a_i), & d(v_j) &:= 3mL, \\ & & p(u, v_j) &:= 3mL + \sum_{k=1}^{j-1} k \cdot B. \end{aligned}$$

We illustrate the construction in Figure 7. In contrast to the task set $\tau(I)$ from Section III-A, the new tasks

$T_1, \dots, T_{3m}$ of $\tau'(I)$ are constructed such that a worst-case interference of $T_{low}$ (described below) must contain a heavy job of each $T_j$, followed by a leaf job. We will see that this allows us to reason that all these heavy jobs can be assumed to be released at the same time as $T_{low}$ is released.

3) Again, we further construct another task $T_{low}$ with the lowest priority, i.e., $pr(T_{low}) := 3m + 1$. As before, $T_{low}$ is the task missing a deadline if and only if there is a valid 3-partition for $I$. Its graph $G(T_{low})$ contains again only single vertex $v$ with labels

$$e(v) := 1, \quad d(v) := 3mL + \sum_{j=1}^{m} j \cdot B.$$

Note that $d(v)$ now includes time for the heavy jobs of all $3m$ tasks $T_i$ in addition to the sum of all window sizes. Thus, if the processor is first busy executing a heavy job of each task $T_i$ and then leaf jobs of all $T_i$ during all windows, $T_{low}$ is missing its deadline.

An example execution is illustrated in Figure 8, based on task set $\tau'(I)$ resulting from instance $I$ in Example III.1. All heavy jobs are released at time 0, keeping the processor busy for $3mL$ time units. After that, the leaf jobs are released according to their partitions, just as before in Section III-C. This keeps the processor constantly busy and causes a deadline miss for $T_{low}$. Conversely, if there is a deadline miss, it again must be the job of $T_{low}$ missing the deadline, since all other jobs have deadlines $3mL$ which is again more than all possible interfering workload of higher priority tasks.

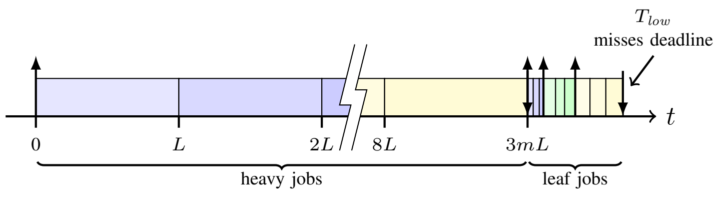


Fig. 8. Schedule resulting from the valid 3-partition for $\tau'(I)$ created from instance $I$ in Example III.1. Note that in the first phase, all heavy jobs execute, after which all leaf jobs create an execution sequence similar to the one in Figure 5.

Based on these observations, it is easily verified that both Lemmas III.2 and III.6 also hold for the new task set $\tau'(I)$. We will now focus on removing the SRS assumption. That is, we will show that the synchronous roots sequence assumed in Lemma III.6 constitutes the worst case interference for task $T_{low}$. In other words, we will show that any job sequence with a deadline miss for $T_{low}$ can be transformed into one with an SRS which still causes a deadline miss.

**Lemma III.7.** *If there is a job sequence generated by $\tau'(I)$ such that $T_{low}$ misses a deadline when scheduled with a static priority scheduler, then there is a synchronous roots sequence in which $T_{low}$ misses a deadline as well.*

*Proof:* By shifting all release times and deadlines in a given job sequence $\rho$, we may assume that the job of $T_{low}$ that is missing its deadline is released at time 0 and has its deadline at time $D = 3mL + \sum_{j=1}^{m} j \cdot B$. We now transform $\rho$ such that the deadline miss of $T_{low}$ occurs in a synchronous roots sequence. During all stages of the transformation, we ensure that the processor work during the time interval $[0, D]$ does not decrease.

1) The first change we apply in case the processor is busy at time 0, i.e., right when $T_{low}$ has its release time. We find a time point $t < 0$ which is the earliest time point such that the processor is continuously busy executing jobs until time 0. Thus, there is a *busy period* of length $\delta := 0 - t$ right before time point 0. We now change $\rho$ by shifting all release times and deadlines of tasks $T_1, \dots, T_{3m}$ by $\delta$. This change ensures that
   - by construction, the processor is now idle right when $T_{low}$ is released at time 0, and
   - during $[0, D]$, the interference of $T_{low}$ by higher priority tasks does not decrease. This is because we moved $\delta$ time units of interfering work *into* the interval, but at most $\delta$ *out* of the interval.
2) Because of the above step, we may assume that the processor was actually idle right when $T_{low}$ was released at time 0. For each $T_i$ that does *not* have the release of a heavy job at time 0, we can therefore do the following:
   - If $T_i$ releases a heavy job at a time $t'$ inside the interval $(0, D]$, we shift all jobs of $T_i$ in $\rho$ by $-t'$, i.e., to earlier time points. Since $t'$ is inside the interval, this does not move the execution of any job of $T_i$ out of the interval, but only potentially moves work into the interval. Thus, the total work inside the interval does not decrease, and we end up with $T_i$'s heavy job being released at time 0.
   - Otherwise, if $T_i$ does *not* release a heavy job inside the interval but a leaf job instead, we move it together with its preceding heavy job (already finished before time 0) so that the heavy job is released at time 0. If there is no such heavy job release, we just insert one at the latest time possible before doing the described transformation. In this step we are moving $L$ time units of work into the interval (that is, the heavy job of $T_i$), but less than $L$ out of the interval (at most a leaf job). Again, work inside $[0, D]$ does not decrease.
   - Finally, if $T_i$ does not release *any* job within $[0, D]$ we can just delete all jobs of $T_i$ from $\rho$ and add its heavy job at time 0. The deletion did not remove any work from the interval by assumption, so in summary the amount of work inside $[0, D]$ increases.
3) After the above steps, there is no job in $\rho$ which is released before $[0, D]$. Further, we can remove all jobs from $\rho$ which are released at or later than $D$ since they do not influence the work inside $[0, D]$. Finally,

releasing the leaf jobs inside the window $[0, D]$ as early as possible will also not decrease the amount of work inside $[0, D]$.

In summary, the construction transforms $\rho$ into a synchronous roots sequence. Since the work of tasks with priority higher than $T_{low}$ in the interval $[0, D]$ did not decrease, $T_{low}$ still misses its deadline. ■

We conclude Section III with our main theorem for branching task models, i.e., with tree release structure.

**Theorem III.8.** *For branching task models, the schedulability problem for static priority schedulers is coNP-hard in the strong sense.*

*Proof:* Given an instance $I$ of 3-PARTITION we described a reduction to an instance $\tau'(I)$ of the static priority scheduling problem. We established that

1) a positive instance $I$ leads to a negative instance $\tau'(I)$, Lemma III.2, and
2) a negative instance $I$ leads to a positive instance $\tau'(I)$, Lemmas III.6 and III.7.

Since 3-PARTITION is $NP$-hard, the static priority scheduling problem for branching task models is $coNP$-hard. Further, this reduction can clearly be executed in polynomial time, and all values in $\tau'(I)$ are bounded polynomially in the values of $I$. With 3-PARTITION being $NP$-hard in the strong sense, our problem is shown $coNP$-hard in the strong sense. ■

## IV. Hardness for Cycle Models

In this section, we show that the static priority scheduling problem is strongly $coNP$-hard also for GMF models. The central idea here is that we can construct a GMF task set that exhibits essentially the same behavior as the branching task in Section III.

For branching task models, the idea behind the task construction was to have one task $T_i$ for each of the $3m$ elements $a_i \in A$. Each task includes a branch for choosing one of the $m$ partitions $P_j$ which $a_i$ should be part of. For GMF models, we would like to have a similar construction, even though GMF tasks do not allow branches. However, we can unroll the tree construction from Section III such that all edges are now serially included in a cyclic graph. The non-deterministic choice of the branch in the original branching task now translates into a non-deterministic choice about at which vertex to start in the cycle graph of the constructed GMF task.

Formally, we construct a GMF task set $\tau''(I)$ from a 3-PARTITION instance $I$ as follows.

1) Just as in Section III, we use a large constant.

$$L = \sum_{i=1}^{3m} m \cdot s(a_i) = m^2 B$$

2) For each $a_i \in A$ we construct a GMF task $T_i$ with priority $pr(T_i) := i$ as follows. It includes $m$ vertices $u_1, \ldots, u_m$ representing $m$ copies of the root vertex

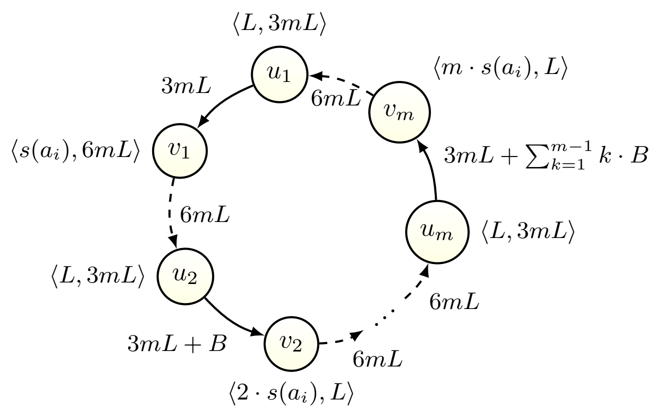


Fig. 9. Illustration of GMF task construction $T_i$. Edges corresponding to tree edges in the tree construction are solid. The dashed edges are connecting to the next root vertex copy.

in $\tau'(I)$ from Section III. Further, it also includes $m$ vertices $v_1, \ldots, v_m$ corresponding to the leaf vertices in $\tau'(I)$. The edges of $G(T_i)$ alternate between vertices $u_j$ and $v_j$, i.e., for all $j = 1, \ldots, m$, we have

- an edge $(u_j, v_j)$ corresponding to a tree edge in $\tau'(I)$, and
- an edge $(v_j, u_{j+1})$ connecting to the next root vertex copy. (We use $u_{m+1} := u_1$ here to simplify notation.) The label of this edge needs to be sufficiently large so that $T_i$ can not release a heavy job too soon after a leaf job, which could create too high workload in case of a negative instance $I$.

The labels are similar to those in $\tau'(I)$:

$$e(u_j) := L, \qquad d(u_j) := 3mL,$$
$$e(v_j) := j \cdot s(a_i), \qquad d(v_j) := 6mL,$$

$$p(u_j, v_j) := 3mL + \sum_{k=1}^{j-1} k \cdot B, \quad p(v_j, u_{j+1}) := 6mL.$$

We illustrate the construction in Figure 9. Note that the label of all edges $(v_j, u_{j+1})$ is $6mL$ which is twice the work that all heavy jobs can create together. This guarantees that at the release of the next heavy job, the processor is not executing work that was connected to the previous leaf job. This construction allows us to apply the reasoning from Section III.

3) Just as before, we construct another task $T_{low}$ with the lowest priority $pr(T_{low}) := 3m + 1$ and only a single vertex $v$ with a self loop. The labels are the same as in Section III for $\tau'(I)$, in addition to the long waiting delay from above.

$$e(v) := 1, \quad d(v) := 3mL + \sum_{j=1}^{m} j \cdot B, \quad p(v, v) := 6mL$$

A synchronous roots sequence for $\tau'(I)$ translates into a job sequence for $\tau''(I)$ which releases a job from a vertex $u_j$ at time 0 for each task $T_i$ in $\tau''(I)$ where each task releases only two jobs. Conversely, because of the long $6mL$ delays, any job sequence generated by $\tau''(I)$ corresponds piecewise to a sequence that is generated by $\tau'(I)$. With both insights, it is easily verified that the three Lemmas III.2, III.6 and III.7 also hold for $\tau''(I)$. Consequently, the main theorem also holds for GMF models:

**Theorem IV.1.** *For GMF models, the schedulability problem for static priority schedulers is coNP-hard in the strong sense.*

*Proof:* By the above discussion, similar to Theorem III.8. ■

## V. Conclusions and Future Work

In this paper we have shown that the schedulability problem for DRT task models with static priority schedulers is *coNP*-hard in the strong sense. This holds even for restricted classes like branching or GMF task models. We have given a proof via a reduction from the 3-PARTITION problem. A central part of the proof was to show that synchronous roots sequences are a worst case in this setting.

As a consequence from our result, it is now firmly established in the real-time scheduling theory that precise schedulability analysis for static priority schedulers is fundamentally more difficult than for dynamic priority schedulers. For future work, we wish to investigate approximations that allow efficient yet precise schedulability analysis for typical instances of the task models at hand. Even with a high worst-case complexity, many typical instances may still be efficiently solvable.

## References

[1] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The Digraph Real-Time Task Model," in *Proc. of RTAS 2011*, pp. 71–80.

[3] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Syst.*, vol. 17, no. 1, pp. 5–22, 1999.

[4] H. Takada and K. Sakamura, "Schedulability of Generalized Multiframe Task Sets under Static Priority Assignment," in *Proc. of RTCSA 1997*, pp. 80–86.

[5] M. Joseph and P. K. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, pp. 390–395, 1986.

[6] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "On the Tractability of Digraph-Based Task Models," in *Proc. of ECRTS 2011*, pp. 162–171.

[7] S. Chakraborty, T. Erlebach, and L. Thiele, "On the complexity of scheduling conditional real-time code," in *WADS 2001*, pp. 38–49.

[8] A. Zuhily, "Exact Response Time Analysis for Multiframe Tasks," Tech. Rep., 2007.

[9] S. Baruah, "Feasibility Analysis of Recurring Branching Tasks," in *Proc. of ECRTS 1998*, pp. 138–145.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

## Appendix

### A. Details for Proof of Lemma III.6

**Lemma A.1.** *Conditions (A) and (B) imply $\sigma_j = B$ for all $j = 1, \ldots, m$.*

*Proof:* For presentation, we re-state both conditions:

$$\forall k = 1, \ldots, m : \sum_{j=1}^{k} j \cdot \sigma_j \geqslant \sum_{j=1}^{k} j \cdot B \tag{A}$$

$$\sum_{j=1}^{m} \sigma_j = m \cdot B \tag{B}$$

We apply some arithmetic transformations on both conditions in order to show our goal $\forall j : \sigma_j = B$. First, we set $\hat{\sigma}_j := \sigma_j - B$. Our goal now becomes $\forall j : \hat{\sigma}_j = 0$ and the conditions become:

$$\forall k : \sum_{j=1}^{k} j \cdot \hat{\sigma}_j \geqslant 0 \tag{A'}$$

$$\sum_{j=1}^{m} \hat{\sigma}_j = 0 \tag{B'}$$

Second, we set $\alpha_k := \sum_{j=1}^{k} j \cdot \hat{\sigma}_j$ which is the LHS of Condition (A') for all $k$. Intuitively, it indicates for each $k$ the idle time until the end of the $k$-th window. (More precisely, negative values are accumulated idle time, positive values are workload that reaches into the following window.) For substitution purposes, we want to express all $\hat{\sigma}_j$ using the new symbols:

$$\alpha_k - \alpha_{k-1} = k \cdot \hat{\sigma}_k \implies \hat{\sigma}_k = \frac{1}{k}(\alpha_k - \alpha_{k-1})$$

We do the substitution in both Conditions (A') and (B') and get:

$$\forall k : \alpha_k \geqslant 0 \tag{A''}$$

$$\sum_{j=1}^{m} \frac{1}{j}(\alpha_j - \alpha_{j-1}) = 0 \tag{B''}$$

With straightforward arithmetics, we can derive from Condition (B"):

$$\begin{aligned}
\sum_{j=1}^{m} \frac{1}{j}(\alpha_j - \alpha_{j-1}) &= \sum_{j=1}^{m} \frac{1}{j}\alpha_j - \sum_{j=1}^{m} \frac{1}{j}\alpha_{j-1} \\
&= \sum_{j=1}^{m} \frac{1}{j}\alpha_j - \sum_{j=0}^{m-1} \frac{1}{j+1}\alpha_j \\
&= \frac{\alpha_m}{m} + \sum_{j=1}^{m-1} \alpha_j \left(\frac{1}{j} - \frac{1}{j+1}\right) - \underbrace{\frac{\alpha_0}{1}}_{=0} \\
&= \frac{\alpha_m}{m} + \sum_{j=1}^{m-1} \frac{\alpha_j}{j(j+1)} \\
&\overset{\text{(B")}}{=} 0
\end{aligned}$$

Since the coefficients of all $\alpha_j$ are positive, a non-negative solution to this condition can only exist with $\alpha_j = 0$ for all $j$. Note that Condition (A") tells us that only non-negative solutions are allowed.

In conclusion, we have $\forall j : \alpha_j = 0$, which is equivalent to $\forall j : \sigma_j = B$. ■

## B. Counterexample for [4]

We show that the schedulability test in [4] is sufficient, but not necessary. Therefore, there is no contradiction to our result, even though the method runs in polynomial time.

We start with a brief overview of the method from [4]. The fundamental concept in this method is the *Maximum Interference Function (MIF)*.

**Definition A.2** (MIF, [4]). *For a task $T$, $M_T(t)$ denotes the maximum time that the execution of $T$ can interfere with the execution of lower priority tasks within $t$ time units.*

As an example, consider task $T_1$ with a release structure represented by the schedule in Figure 10. Its MIF is shown in Figure 11(a).

The idea with that concept is the following. If for any job of a task, the sum of its computation time and the interference of all higher priority tasks exceeds its deadline, the task is unschedulable with that particular priority order. Formally, this condition can be stated for all jobs $J$ of any task $T$:

$$\exists t \leqslant d(J) : \sum_{T' \in hp(T)} M_{T'}(t) + e(J) \leqslant t \qquad (1)$$

With $hp(T)$ we denote the set of tasks with higher priority than $T$. The condition can be read as: if there is a time interval, starting from the release of $J$, that fits both the computation time of $J$ and also the interfering workload of all tasks of higher priority, then $J$ is schedulable. If that holds for all tasks $T$ and all their jobs $J$, then the whole task set is schedulable.

While that can be shown to be true, the converse unfortunately does not hold. The reason for this is that the MIF is precise just for a *single task*, but the sum of MIFs of several tasks is not always precise for a *task set*. Instead, this sum overapproximates the interference that a lower priority task can experience. To illustrate this, consider the following task set $\tau$.

Task $T_1$: This task first releases a job with execution time 5, and then two jobs, each with execution time 1 after 11 units, separated by 2 time units.

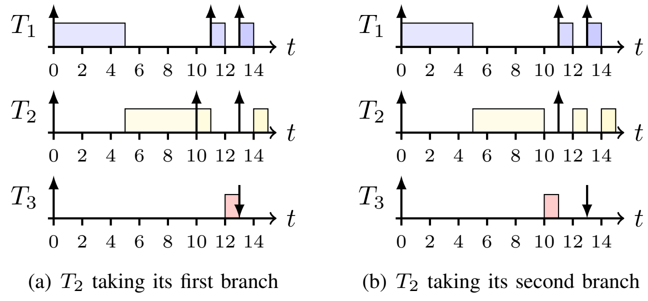
Task $T_2$: This task has a branching behavior. It first also releases a job with execution time 5, and then it either

- releases two jobs with execution time 1 after 10 time units, separated by 3 time units, or it
- releases one job with exeuction time 2 after 11 time units.

Task $T_3$: Finally, this third task is the one for which we want to check schedulability. It releases just one job which has an execution time of 1 and a deadline of 13. Having the lowest priority of the task set, we want to know whether it can experience consecutive interference of (strictly) more than 12 time units.

We first see that in any concrete execution of the system, $T_3$ will meet its deadline. Worst cases for both possible behaviors of $T_2$ are shown in Figure 10.

However, the MIF abstraction with Condition (1) predicts a deadline miss. The MIFs for $T_1$ and $T_2$ are shown in



$T_1$
$T_2$
$T_3$
$t$
0 2 4 6 8 10 12 14
(a) $T_2$ taking its first branch
(b) $T_2$ taking its second branch

Fig. 10. Two possible schedules for the given task set $\tau$, assuming that $T_1$ has highest priority. However, the particular priority order of $T_1$ and $T_2$ does not influence schedulability of $T_3$.

Figures 11(a) and 11(b). The sum of $M_{T_1}$ and $M_{T_2}$ clearly satisfies for $t \in [0, 14]$:

$$M_{T_1}(t) + M_{T_2}(t) \geqslant t.$$

Thus, Condition (1) would classify $T_3$ as unschedulable if it is assigned the lowest priority.

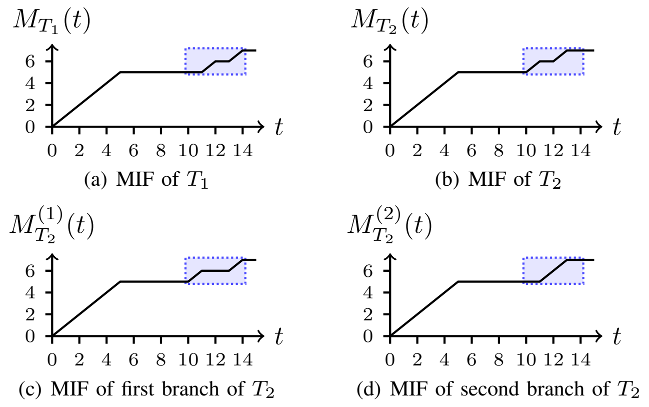


Fig. 11. MIFs for tasks $T_1$ and $T_2$. The MIFs for both branches of $T_2$ are shown separately. We emphasize the parts of the MIFs that differ.

Why is that so? For task $T_2$, $M_{T_2}$ is an abstraction of both branching possibilities, by taking the maximum of their respective MIFs. We show these separately in Figures 11(c) and 11(d) and see that if they are *separately* considered together with $M_{T_1}$, the system is classified as being schedulable by Condition (1). Clearly, the MIF abstraction is overapproximate and thus not suitable for an exact analysis.

Note that for $\tau$ to be schedulable, the deadlines of the jobs of $T_1$ and $T_2$ need to be sufficiently large. In particular, they would need to be larger than the minimum inter-release separation times. Since [4] assumes constrained deadlines, this example would not be a counterexample in the strict sense. However, $\tau$ can easily be transformed into a task set $\tau'$ with constrained deadlines by "chopping up" the heavy tasks in smaller pieces, so that the shorter jobs all can meet their deadlines. For brevity, we skip the details.