Combinatorial Abstraction Refinement for Feasibility Analysis

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

***Abstract*—The traditional periodic workload model for hard real-time systems has been extended by more expressive models in recent years. These models based on different classes of directed graphs allow modeling of structures like frames, branching and loops.**

With more expressiveness comes higher complexity of the associated analysis problems. Feasibility of digraph-based models with dynamic priority schedulers has been shown to be tractable via pseudo-polynomial algorithms. However, the problem was shown to be intractable for static priority scheduling since it is strongly coNP-hard already for the relatively simple class of cyclic digraphs. The core of this problem is an inherent combinatorial explosion caused by combining different behaviors of the participating tasks, lacking local worst cases.

We introduce a novel iterative approach to efficiently cope with this combinatorial explosion, called combinatorial abstraction refinement. In combination with other techniques it significantly reduces exponential growth of run-time for most inputs. A prototype implementation for analysing static priority feasibility outperforms the state-of-the art pseudo-polynomial analysis for dynamic priority feasibility. It further shows better scaling behavior for typical problem sizes. We believe that this method can be applicable to a variety of combinatorial problems in the theory of real-time systems with certain abstraction structures.

I. Introduction

For a given system description, one of the core objectives in the theory of hard real-time systems is to analyse whether the system workload will meet all its timing requirements at run-time for all conceivable situations. The classical *periodic task model* by Liu and Layland [1] describes system workload as a collection of independent tasks. Each task is activated periodically and the run-time of each activation is bounded by a value derived from a worst-case execution time analysis.

The model's assumption that each task is behaving periodically is often too simple. More expressive models have been proposed in recent years [2], [3], [4] in order to increase modeling power. One of the most expressive models is the *Digraph Real-Time task model (DRT)* [5] which models each task using a directed graph. Timing constraints are represented as deadlines until which task activations need to finish their executions. Schedulability analysis aims at statically proving the absence of deadline misses, assuming a given scheduler for the workload. Analysis complexity increases if expressiveness of the workload model grows.

The analysis complexity is further dependent on the type of scheduler. The two common scheduler classes, *dynamic* and *static priority schedulers*, differ in how they pick which task to execute in cases where more than one of them is waiting for execution. For dynamic priority schedulers, the feasibility problem for the DRT model has been shown to be tractable for uniprocessor platforms [5]. The method is based on evaluating demand bound functions and leads to a pseudo-polynomial test. However, the problem has been shown to be strongly coNP-hard for static priority schedulers [6], implying that a pseudo-polynomial test cannot exist (assuming $P \neq NP$). Thus, the worst-case run-time of a schedulability test can be expected to be exponential in the task parameters.

One of the fundamental reasons for this hardness is that tasks do not have local worst-cases which can be combined to a global worst-case. As a consequence, exponentially many combinations of scenarios from all tasks need to be considered. Such *combinatorial explosions* are major sources of intractability for many problems in the theory of real-time systems and beyond. They introduce exponential algorithm run-times leading to poor scaling behaviour.

In this paper, we show that by carefully considering the properties of tasks and their interactions, a static priority schedulability test can be developed which runs efficiently for typical problem instances. The key insights are abstractions which allow to potentially prune large parts of the search space and efficiently guide the search for a deadline miss. In particular, we provide the following contributions:

- We present methods to significantly reduce the exponential number of relevant objects to be tested by introducing *dominance relations* on two domains relevant for the analysis (critical vertices and critical request functions).
- We introduce an iterative technique called *combinatorial abstraction refinement* in order to deal with a combinatorial explosion in the feasibility test for static priorities.

Despite its exponential worst-case complexity, a prototype implementation of our method outperforms the state-of-the-art algorithm for dynamic priority feasibility which is pseudo-polynomial. Task sets with 50 tasks and more can be analysed within a few seconds. This demonstrates competitiveness of our approach using domain knowledge for sophisticated optimizations.

We believe that our abstraction refinement technique may be applicable beyond the concrete problem we are solving in this paper. Many problems in the theory of real-time systems are combinatorial in nature and our approach can be used as soon as certain lattice structures for hierarchical abstractions are defined.

A. Prior Work

The *periodic task model* [1] represents each task with two integers for period and worst-case execution time (WCET). Deadlines are implicit, i.e., equal to periods. Efficient analysis procedures are known for dynamic priorities via the

utilization bound [1] and for static priorities via response-time analysis [7]. More expressive models include the *Multiframe (MF)* [2] and *Generalized Multiframe (GMF)* [3] task models. Each task in these models cycles through a list of different *frames* with different execution times (for both MF and GMF) and different inter-release separation times and deadlines (only GMF). Feasibility tests for these models are based on demand bound functions [3]. Static priority schedulability tests generalize response time analysis [8], [9] or utilization bounds [2], [10], [11], [12]. These tests however are either imprecise, i.e., over-approximate, or very slow because of exponential explosion in complexity.

The most general model to date with a tractable feasibility problem is the *Digraph Real-Time task model (DRT)* [5]. Each task is modeled with a directed graph in which vertices represent job releases and edges represent branches and inter-release delays. Feasibility for dynamic schedulers has been shown to be tractable [5], even for an extension with a bounded number of global timing constraints [13]. However, similar results have been shown to be unlikely for static priorities since the problem becomes strongly coNP-hard in this case [6].

The introduction of *task automata* [14] is taking a model checking approach. This model is based on timed automata, extended with real-time tasks. Expressiveness is so high that the associated schedulability problem is even undecidable in a few variants of the model. The decidable cases, including models for dynamic and static priority schedulers, suffer from the same scalability problem that is common to all model checking approaches.

An approximative solution is the *real-time calculus (RTC)* [15] which uses arrival and service curves (not unlike the request functions we use) in order to describe task activations and availability of computing resources. There is a large body of research around RTC but it is inherently over-approximate.

II. Preliminaries

This section presents the task model with its syntax and semantics, followed by the problem description.

A. Task Model

We use the *digraph real-time (DRT) task model* [5] 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 is labeled with an ordered pair $\langle e(v), d(v)\rangle$ denoting worst-case execution-time demand $e(v)$ and relative deadline $d(v)$ 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. In this work, we assume deadlines to be *constrained* by inter-release separation times, i.e., for each vertex u, its deadline label $d(u)$ is bounded by the minimal $p(u, v)$ for all outgoing edges (u, v).

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

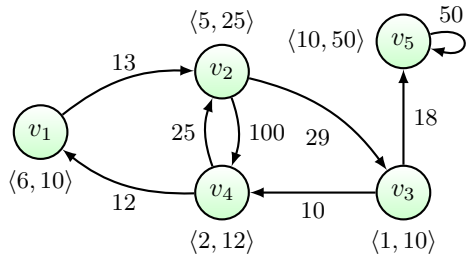


Fig. 1. An example task containing five different types of jobs

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 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 $\rho = [(r_1, e_1, d_1), (r_2, e_2, d_2), \ldots]$ 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 τ, a job sequence ρ is *generated by* τ, if it is a composition of sequences $\{\rho_T\}_{T\in\tau}$, which are individually generated by the tasks T of τ.

Example II.2. *For the example task T in Figure 1, consider the job sequence $\rho = [(6, 6, 16), (19, 3.3, 44), (51, 1, 61)]$. It corresponds to path $\pi = (v_1, v_2, v_3)$ 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 ρ (associated with v_2) is released as early as possible after the first job (v_1), the same is not true for the third job (v_3).

B. Schedulability and Feasibility

We assume preemptive scheduling on uniprocessor systems and use the standard notion of schedulability:

Definition II.3 (Schedulability)**.** *A task set τ is* schedulable with scheduler *Sch, if and only if for all job sequences generated by τ, all jobs meet their deadlines when scheduled with Sch. Otherwise, τ is* unschedulable with *Sch.*

While the notion of schedulability fixes a particular scheduler, feasibility is a related problem about the existence of such a scheduler:

Definition II.4 (Feasibility)**.** *A task set τ is* feasible*, if and only if there is a scheduler Sch such that τ is schedulable with Sch.*

We distinguish between *dynamic* and *static* priority schedulers, i.e., whether the scheduler has to obey a given order of relative priorities on the task set. In general, dynamic priority schedulers have more freedom in their scheduling decisions than static priority schedulers and can therefore successfully schedule more task sets.

For dynamic priority schedulers, it is well-known that in our setting of independent jobs, the earliest deadline first (EDF) scheduler is optimal. This means that if a task set can be scheduled with 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 [5] for systems with bounded utilization. This is even the case for an extension of DRT with global timing constraints [13]. Thus, feasibility is considered to be a tractable problem.

Our focus in this work is on static priority scheduling. A priority order $\mathcal{P} : \tau \to \mathbb{N}$ assigns a priority to each task (with lower numbers for higher priorities). We assume priorities to be unique, i.e., $\mathcal{P}$ is a bijection onto $\{1, \ldots, \|\tau\|\}$. We say that a task set τ is *SP schedulable with* $\mathcal{P}$ if a static priority scheduler using priority order $\mathcal{P}$ can successfully schedule τ. We further say that τ is *SP feasible* if there is a $\mathcal{P}$ such that τ is SP schedulable with $\mathcal{P}$. As we will see in Section III, both problems are equivalent up to a linear factor. Previous work has proved that SP schedulability is strongly $coNP$-hard already for sub-classes of DRT with cyclic graphs [6]. This means that no exact pseudo-polynomial algorithm can test SP schedulability or SP feasibility for a given task set. However, since both problems are highly relevant, we present in Sections III to V an efficient algorithm that solves typical instances in time comparable to state-of-the-art solutions for EDF schedulability.

III. Method Overview

In this section, we give an overview of our algorithm for checking SP schedulability and SP feasibility.

A. Lowest-Priority Feasibility

Our decision procedures are based on checking whether a task in a task set may be assigned the lowest priority.

Definition III.1. *For a task set $\tau = \{T_1, \ldots, T_N\}$, a task $T \in \tau$ is* lowest-priority feasible in τ *if there is a priority order $\mathcal{P}$ with $\mathcal{P}(T) = N$ such that T does not miss any deadlines if τ is SP scheduled with $\mathcal{P}$.*

Note that this definition does not state anything about deadline misses of any other tasks in τ. Further, if $T \in \tau$ is lowest-priority feasible in τ, then this property is independent of the relative priorities of all other tasks in τ. We summarize this insight in the following Lemma.

Lemma III.2. *For a task set $\tau = \{T_1, \ldots, T_N\}$, if a task $T \in \tau$ is lowest-priority feasible shown by a priority order $\mathcal{P}$, then it will always meet its deadline if SP scheduled with any permutation of $\mathcal{P}$.*

Proof: The amount of interference that a task T experiences from tasks of higher priorities does not change when their relative priorities change. In fact, even the actual interference patterns do not change, i.e., the exact timing of the interference. Thus, all permutations of priorities of tasks with higher priority than T lead to the same schedulability behavior of T. ∎

As we will see now, SP schedulability and SP feasibility can both be reduced to checking lowest-priority feasibility of individual tasks.

B. SP Schedulability

Given a task set $\tau = \{T_1, \ldots, T_N\}$ with a priority order $\mathcal{P}$, SP schedulability of τ with $\mathcal{P}$ can be decided as follows. For each task $T \in \tau$, check whether T is lowest-priority feasible in the set of all tasks with priority up to $\mathcal{P}(T)$. Note that this condition is both sufficient and necessary, since adding tasks of lower priority to a task set does neither introduce nor remove deadline misses of higher priority tasks.

Lemma III.3. *A task set $\tau = \{T_1, \ldots, T_N\}$ is SP schedulable with a priority order $\mathcal{P}$ if and only if each $T \in \tau$ is lowest-priority feasible in $\{T' \mid \mathcal{P}(T') \leqslant \mathcal{P}(T)\}$.*

Proof: By above discussion. ∎

C. SP Feasibility: Audsley's Algorithm

Checking SP feasibility of a task set τ is possible using a similar method which is usually called *Audsley's Algorithm*. First, check all $T \in \tau$ for lowest-priority feasibility in τ. If this check is successful for any T, recursively apply the algorithm to $\tau \setminus \{T\}$. However, if during this recursive procedure for some subset $\tau' \subseteq \tau$ no such T is found, τ is not SP feasible. This method has the additional advantage of synthesizing a priority order if the task set is found to be SP feasible, by taking the reverse order in which the tasks were found to be lowest-priority feasible.

Lemma III.4. *A task set $\tau = \{T_1, \ldots, T_N\}$ is SP feasible if and only if either it is empty or there is a $T \in \tau$ which is lowest-priority feasible in τ and $\tau \setminus \{T\}$ is SP feasible.*

Proof: It is clear that if this test succeeds, τ is indeed SP feasible since the synthesized priority order automatically satisfies the condition in Lemma III.3 from above.

Conversely, let the test fail for some subset $\tau' \subseteq \tau$ but assume there is a priority order $\mathcal{P}$ for which τ is SP schedulable. Of all tasks in τ', some task $T \in \tau'$ is assigned lowest priority by $\mathcal{P}$. Since τ is assumed to be SP schedulable with $\mathcal{P}$, this task T will always meet all deadlines even with interference of all tasks of higher priority in τ. However, τ' is just a subset of τ, therefore the interference experienced by T from all other tasks in τ' can not be larger (Lemma III.2). Thus, T must be lowest-priority feasible in τ', contradicting the assumption that the test failed for τ'. ∎

Note that this means that in the process of synthesizing a priority order starting with the lowest priority, one can never "pick wrong" among all tasks that are lowest-priority feasible.

D. Critical Vertices

As we will see now, it is sufficient to test all vertices of a task separately in order to conclude that the task is lowest-priority feasible. The fundamental assumption for this to hold is that deadlines are constrained, which implies that jobs of the same task do not cause interference to each other[1].

More specifically, given a vertex v with WCET $e(v)$ and relative deadline $d(v)$, it is sufficient to check whether the tasks of higher priority τ_{high} can execute for an accumulated time of strictly more than $d(v)-e(v)$ time units in any time interval of size $d(v)$. In case that is possible, the task containing v can not be lowest-priority feasible since the corresponding job may miss its deadline. However, if that is not the case, we say that v is *schedulable with interference set* τ_{high} or just *schedulable* if τ_{high} is clear from the context. Our algorithm for testing schedulability of a single vertex is described in Section IV.

Lemma III.5. *Given a task set τ, a task $T \in \tau$ is lowest-priority feasible if and only if all vertices $v \in G(T)$ are schedulable with interference set $\tau \setminus \{T\}$.*

Proof: By above discussion. ∎

In fact, not all vertices need to be checked. Consider two vertices v_1 and v_2 of a task with

$$\langle e(v_1), d(v_1)\rangle = \langle 3, 10\rangle \text{ and } \langle e(v_2), d(v_2)\rangle = \langle 2, 20\rangle .$$

Assume that v_1 is schedulable with some interference set τ. This immediately implies that v_2 is schedulable as well, since the execution demand of jobs corresponding to v_2 is lower and only needs to meet a deadline that is larger. We say that v_1 *dominates* v_2 and call a set of vertices in a task which are *not* dominated by others *critical vertices*. Clearly, only critical vertices need to be checked for schedulability, which also implies schedulability of all other vertices and thus lowest-priority feasibility of the whole task.

This observation is important for run-time complexity of our analysis method. For a set of n vertices with a uniform random distribution of WCET and deadlines, the expected number of critical vertices is $\mathcal{O}(\sqrt{n})$, dramatically reducing the run-time of a loop that tests all vertices individually for schedulability. Further, as we will see in Section IV, testing a vertex v for schedulability is in the worst case exponential in $d(v)$. Therefore, an optimization that tends to remove vertices v with large $d(v)$ has the additional benefit of avoiding the most expensive individual tests.

The concept of a domination relation between two vertices can be extended to vertices of different tasks with different priorities. Take the two vertices v_1 and v_2 from above and now assume that v_2 is part of a task with higher priority than the one containing v_1. If v_1 turns out to be schedulable, then v_2 is as well, since the set of tasks interfering with the jobs corresponding to v_2 is smaller, thus causing less interference. We summarize this concept as follows.

[1]Another important condition for this is that all jobs released by the same task do have the same priority, i.e., this method is not directly applicable to a scheduler where different vertices could be assigned different static priorities.

Definition III.6. *For a task set τ with priority order $\mathcal{P}$ and tasks $T, T' \in \tau$, we say that $v \in G(T)$* dominates *$v' \in G(T')$, written $v \succcurlyeq v'$, if and only if:*

1) $e(v) \geqslant e(v')$,
2) $d(v) \leqslant d(v')$ *and*
3) $\mathcal{P}(T) \geqslant \mathcal{P}(T')$.

If $T = T'$, then we call this an intra-task dominance*, otherwise an* inter-task dominance*. A maximal set of vertices v containing no other v' with $v' \succcurlyeq v$ is called a set of* critical vertices*.*

For checking SP schedulability the application of this is straightforward. Test a set of critical vertices for schedulability, directly leading to SP schedulability of the whole task set. For checking SP feasibility, the priority order is not known a priori. Thus, initially only intra-task dominance can be considered. However, each time a task T is found to be lowest-priority feasible, all inter-task dominated vertices in the remaining tasks are clearly non-critical and do not need to be tested anymore.

IV. Single-Job Interference Testing

We now focus on checking whether a single job may experience sufficient interference from tasks of higher priority such that it misses its deadline. For the rest of this section, we assume that we want to check schedulability of a vertex v with label $\langle e, d\rangle$, i.e., with WCET e and deadline d. We want to check whether a given task set τ of higher priority tasks may cause more than $d - e$ time units of interference in any time window of d time units. Note that the relative priorities of all tasks in the interference set τ do not matter in this case.

A naive approach for this test could be as follows. For each task $T \in \tau$, pick a path $\pi^{(T)}$. Given the set of paths $\{\pi^{(T)}\}_{T\in\tau}$, the *synchronous arrival sequence* can be simulated, i.e., a job sequence where all jobs take their maximal execution time, the first job from each path $\pi^{(T)}$ is released at time 0 and all following jobs as soon as allowed by the edge labels. See Figure 2 for an example. Vertex v is schedulable if and only if for an exhaustive enumeration and combination of all such paths, each simulation turned out to detect at least e idle time units within the first d time units.

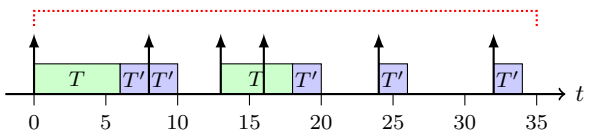


Fig. 2. Example of simulating a synchronous arrival sequence in a time interval of size 35. The interference set is $\tau = \{T, T'\}$ with T from Figure 1 and T' a periodic task with $(e, d, p) = (2, 8, 8)$. From task T, we simulate path (v_1, v_2, v_3). In this concrete scenario, 14 idle time units are detected.

Such an approach is of course prohibitively slow since there are two sources of exponential explosion: the number of paths in each task, and the number of path combinations to be simulated. In the rest of this section, we present ways of reducing the relevant number of paths. Section V introduces a method for reducing the number of combination tests.

A. Request Functions

In order to deal with the exponential number of paths in each task, we introduce a path abstraction that is sufficient for testing interference but allows to substantially reduce the number of paths that have to be considered. We abstract a path π with a *request function* which for each t returns the accumulated execution requirement of all jobs that π may release during the first t time units.

Definition IV.1. *For a path $\pi = (v_0, \ldots, v_l)$ through the graph $G(T)$ of a task T, we define its* request function *as*

$$rf_\pi(t) := \max\{e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } p(\pi') < t\}$$

where $e(\pi) := \sum_{i=0}^{l} e(v_i)$ and $p(\pi) := \sum_{i=0}^{l-1} p(v_i, v_{i+1})$.

In particular, $rf_\pi(0) = 0$ and $rf_\pi(1) = e(v_0)$, assuming that all edge labels are strictly positive. Note further that two paths sharing a prefix π have request functions that are identical up to the duration $p(\pi)$ of that prefix. We give an example in Figure 3.

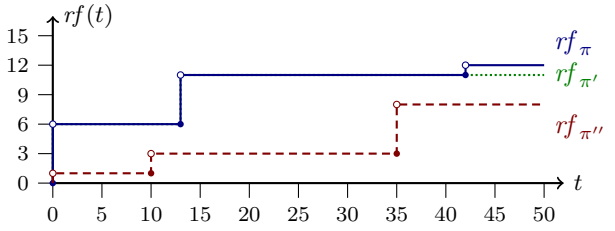


Fig. 3. Example of request functions on $[0, 50]$. Paths are taken from $G(T)$ in Figure 1 with $\pi = (v_1, v_2, v_3)$, $\pi' = (v_1, v_2, v_4)$ and $\pi'' = (v_3, v_4, v_2)$. Note that π and π' share a prefix and therefore rf_π and $rf_{\pi'}$ coincide on interval $[0, 42]$.

Using this path abstraction, we can give a precise characterization of schedulability of a vertex v. The following theorem considers all combinations of all request functions corresponding to paths in all tasks of higher priority. Intuitively, the jobs corresponding to a vertex v are schedulable if and only if for each combination there is some time interval smaller than $d(v)$ in which the sum of all requests in addition to $e(v)$ does not exceed the size of the time interval. This means that the job in question is always able to finish execution at some point before $d(v)$ time units have passed because the interference up to this point allows enough time for it to execute to completion. We write $\Pi(T)$ for the set of paths in $G(T)$ and $\Pi(\tau)$ for $\Pi(T_1) \times \ldots \times \Pi(T_N)$, i.e., the set of all combinations of paths from all tasks. Further, let $\bar{\pi} = (\pi^{(T_1)}, \ldots, \pi^{(T_N)})$ denote an element of $\Pi(\tau)$. A detailed proof of the theorem is given in Appendix A.

Theorem IV.2. *A vertex v is schedulable with interference set τ if and only if*

$$\forall \bar{\pi} \in \Pi(\tau) : \exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf_{\pi^{(T)}}(t) \leqslant t. \quad (1)$$

Note that in Condition (1) it is sufficient to only test $t \leqslant d(v)$ which are integers since request functions only change at integer points. There are two reasons for that: (1) we assume all graph labels to be integers, and (2) the release sequences represented by request functions have all job releases as early as possibly allowed by the edge labels. For the rest of the paper, functions only need to be evaluated at integer points.

Generally, each $\Pi(T)$ may be infinite, since there are infinitely many paths in directed graphs with (directed) cycles. However, as we have already seen for paths sharing a prefix, only finitely many prefixes of paths in $\Pi(T)$ are relevant. This is because only a bounded number of them has request functions that differ somewhere on the interval $[0, d(v)]$. Formally, let $RF(T)$ denote the set of request functions corresponding to the paths in $G(T)$, restricted to domain $[0, d(v)]$. As with Π before, we write $RF(\tau)$ for all combinations of tasks, i.e., $RF(T_1) \times \ldots \times RF(T_N)$ and $\bar{rf} = (rf^{(T_1)}, \ldots, rf^{(T_N)})$ for elements of $RF(\tau)$. With this notation, Condition (1) is equivalent to

$$\forall \bar{rf} \in RF(\tau) : \exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) \leqslant t. \quad (2)$$

B. Critical Request Functions

The test in Condition (2) is already finite, since $RF(\tau)$ can be effectively and finitely enumerated. However, the number of request functions per task is exponential in d. We will see that only a small fraction of them is in fact relevant. Consider two request functions rf and rf' such that $rf(t) \geqslant rf'(t)$ for all t in $[0, d(v)]$. If Condition (2) is satisfied using rf for some task, then it will also be satisfied with rf' instead for the same task, since the LHS of the inequality is even smaller with rf'. Clearly, only rf needs to be considered.

We formalize this by introducing a notion of dominance on the set of request functions.

Definition IV.3. *For two request functions rf and rf' on domain $[0, d]$, we say that rf* dominates *rf', written $rf \succcurlyeq rf'$, if and only if*

$$\forall t \in [0, d] : rf(t) \geqslant rf'(t).$$

A maximal set of request functions rf containing no other rf' with $rf' \succcurlyeq rf$ is called a set of critical request functions.

Example IV.4. *As an example, we take again the request functions on $[0, 50]$ in Figure 3. Note that rf_π has at each point a value at least as large as $rf_{\pi'}$. Therefore we have $rf_\pi \succcurlyeq rf_{\pi'}$. The same holds with $rf_{\pi''}$, i.e., $rf_\pi \succcurlyeq rf_{\pi''}$. In fact, rf_π is a* critical *request function for this task.*

Let $RF^*(T)$ denote the (unique) set of critical request functions for T and let $RF^*(\tau)$ be defined analogously. Then Condition (2) is equivalent to the following which is quantifying only over all combinations $RF^*(\tau)$ of *critical* request functions instead of $RF(\tau)$.

$$\forall \bar{rf} \in RF^*(\tau) : \exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) \leqslant t. \quad (3)$$

Typically, only a rather small fraction of request functions in a task is critical (tens versus thousands or millions). This already

reduces the number of combinations dramatically for which Condition (3) needs to be checked, despite the theoretically exponential size of all $RF^*(T)$ in the worst case.

C. Computation of Request Functions

Critical request functions are our solution to the exponential explosion of paths in each graph $G(T)$ by carefully considering only the relevant ones. Before we present in Section V our solution to the second source of exponential complexity, i.e., the number of *combinations* of request functions from different tasks, we sketch an efficient method for computing critical request functions for a given graph $G(T)$.

The algorithm is based on an iterative graph exploration technique presented in [5] based on path abstractions, which in our case are request functions. The idea is to start with all 0-paths in the graph, i.e., paths containing just a single vertex, and iteratively extending each already generated path with all successor vertices. During that procedure, request functions that are found to be dominated by an already generated one are discarded. The procedure ends when all critical request functions on domain $[0, d(v)]$ have been generated. Fore more details about the general algorithm framework we refer to [5].

D. Full Algorithm

We summarize this section by presenting the first version of the full algorithm in Figures 4 to 7, based on Lemmas III.3 to III.5 and Theorem IV.2. Assumed is a function $generate\text{-}rfs(T)$ returning a set of critical request functions for a task T on the relevant time interval. Such a function can be implemented as sketched above in Section IV-C. We further assume that vertices have been marked as critical and implicitly update these markings in $SP\text{-}feasible(\tau)$.

function $schedulable(v, \tau)$:
1: **for all** $T \in \tau$ **do**
2: $\quad RF^*(T) \leftarrow generate\text{-}rfs(T)$
3: **end for**
4: **for all** $\bar{rf} \in RF^*(\tau)$ **do**
5: $\quad$ **if** $\forall t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) > t$ **then**
6: $\quad\quad$ **return false**
7: $\quad$ **end if**
8: **end for**
9: **return true**

Fig. 4. Algorithm for schedulability of a vertex v with interference set τ.

function $lp\text{-}feasible(T, \tau)$:
1: **for all** critical $v \in G(T)$ **do**
2: $\quad$ **if not** $schedulable(v, \tau \setminus \{T\})$ **then**
3: $\quad\quad$ **return false**
4: $\quad$ **end if**
5: **end for**
6: **return true**

Fig. 5. Algorithm for lowest-priority feasibility of a task $T \in \tau$.

function $SP\text{-}schedulable(\tau, \mathcal{P})$:
1: **for all** $T \in \tau$ **do**
2: $\quad$ **if not** $lp\text{-}feasible(T, \{T' \in \tau \mid \mathcal{P}(T') \leqslant \mathcal{P}(T)\})$ **then**
3: $\quad\quad$ **return false**
4: $\quad$ **end if**
5: **end for**
6: **return true**

Fig. 6. Algorithm for SP schedulability of a task set τ with priorities $\mathcal{P}$.

function $SP\text{-}feasible(\tau)$:
1: **if** $\tau = \emptyset$ **then**
2: $\quad$ **return true**
3: **end if**
4: **for all** $T \in \tau$ **do**
5: $\quad$ **if** $lp\text{-}feasible(T, \tau)$ **then**
6: $\quad\quad$ **return** $SP\text{-}feasible(\tau \setminus \{T\})$
7: $\quad$ **end if**
8: **end for**
9: **return false**

Fig. 7. Algorithm for SP feasibility of a task set τ.

Note that $SP\text{-}schedulable(\tau, \mathcal{P})$ makes $\mathcal{O}(\|\tau\|)$ calls to $lp\text{-}feasible(T, \tau)$, compared to $SP\text{-}feasible(\tau)$ making $\mathcal{O}(\|\tau\|^2)$ such calls. Thus, the run-time difference is only about a factor linear in the number of tasks.

The main bottleneck of both algorithms is the combinatorial explosion in line 4 of $schedulable(v, \tau)$. Even though the number of critical request functions per task is low, a brute force style test of all combinations is still prohibitively expensive. We deal with this problem in the following section by replacing the rather naive combinatorial test with our proposed iterative approach using combinatorial abstraction refinement.

V. Combinatorial Abstraction Refinement

In the previous section we dealt with exponential problem sizes by introducing dominance relations on the domains of vertices and request functions in order to discard large fractions of the search space. However, the combinatorial problem of having to try all combinations of (critical) request functions remains.

In order to deal with this problem, we introduce an abstraction on top of request functions, called *abstract request functions*. This abstraction is still sound: if a combination of abstract request functions signals schedulability of a vertex, this conclusion is indeed true. However, if a combination signals non-schedulability, it may be that this conclusion is over-approximate. In such a case, in order to still give precise results, we *refine* the abstraction into combinations of "less abstract" request functions. This is iterated until either a vertex is finally found to be schedulable, or we arrive at a combination of concrete request functions, i.e., without any remaining abstraction, conclusively resulting in unschedulability.

As we will see, the result is a precise analysis method which avoids combinatorial explosion for typical inputs. The rest of this section presents the details of our technique.

A. Abstract Request Functions

We introduce an abstraction of a *set* of request functions by taking their point-wise maximum.

Definition V.1. *We call rf a* concrete request function *if it is derived from a path π in a graph $G(T)$ as in Definition IV.1.*

We call rf an abstract request function *if there is a set $\{rf_1, \ldots, rf_k\}$ of concrete request functions, such that*

$$\forall t : rf(t) = \max\{rf_1(t), \ldots, rf_k(t)\}.$$

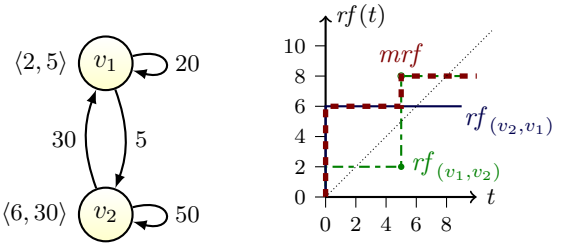
In that case we write $rf = rf_1 \sqcup \ldots \sqcup rf_k$.

Abstract request functions can be directly used in a schedulability test in order to get over-approximate results. Specifically, for each task T, let $mrf^{(T)}$ be the abstract request function derived from the *whole* set $RF^*(T)$ of all critical (concrete) request functions. We call $mrf^{(T)}$ the *most abstract request function* for T. Using the combination of all $mrf^{(T)}$, a vertex v is schedulable if

$$\exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} mrf^{(T)}(t) \leqslant t. \tag{4}$$

This holds because Condition (4) implies Condition (3) from Section IV-B. The test is much more efficient since it uses only *one* combination of (now abstract) request functions instead of exponentially many.

However, Condition (4) is over-approximate. If it is satisfied, vertex v is indeed schedulable, but if it fails, v may still be schedulable. See Figure 8 for an example of this. The reason is that the abstraction loses information, and therefore the implication does not hold in the other direction, i.e., Conditions (3) and (4) are *not* equivalent.



(a) $G(T)$ of task T (b) Request functions

Fig. 8. Example demonstrating imprecise results if just the most abstract request function mrf is used. The two critical request functions $rf_{(v_1,v_2)}$ and $rf_{(v_2,v_1)}$ for T are shown. Both scenarios identify idle intervals on $[0, 8]$: path (v_1, v_2) from $t = 2$ and path (v_2, v_1) from $t = 6$. Thus, a vertex with $\langle e, d\rangle = \langle 1, 8\rangle$ would be schedulable with T having higher priority. However, this information is lost if only mrf is considered.

In order to turn this back into a precise test while still taking advantage of the abstraction power, we now introduce an abstraction refinement technique which allows us to iteratively refine the abstraction until a precise answer can be given.

B. Abstraction Refinement

As we have seen above, testing schedulability using just the most abstract request function may give an imprecise result in case the test fails. Instead of falling back to testing all combinations of concrete request functions, the abstraction can be *refined* by trying intermediate steps. For example, the test can be applied to abstract request functions that do not take the maximum over *all* concrete request functions, but for example just half of them. The result is a test which is more precise than Condition (4) but still more efficient than Condition (3). Since this step is more precise, it is more likely that the test succeeds in case v is schedulable. If the test still fails, the abstractions can be refined even further.

We now make this idea formal and present the details. For each task T, we build an *abstraction tree* bottom-up as follows. The leaves are represented by all concrete request functions from $RF^*(T)$. In each step of the construction, we take two nodes rf_1 and rf_2 which do not yet have a parent node and which are "closest", for example by using a similarity metric on request functions (see Section V-C). For these two nodes, we create their parent node by taking their point-wise maximum $rf_1 \sqcup rf_2$. This is repeated until we have created the full tree, in which case the tree root is the most abstract request function $mrf^{(T)}$. Figure 9 illustrates the abstraction tree[2].

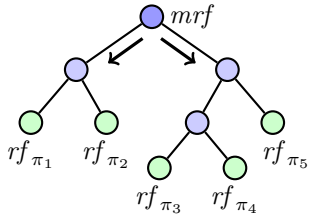


Fig. 9. Request function abstraction tree for request functions of task T in Figure 1. The leaves are all five concrete (critical) request functions on $[0, 50]$. Each inner node is the point-wise maximum of all descendants and thus an abstract request function. Abstraction refinements happen downwards along the edges, starting at the root.

Our abstraction refinement algorithm works on these abstraction trees as follows. First, test schedulability of a vertex v by testing the combination of all tree roots, exactly as in Condition (4). If that test fails, replace one of the abstract request functions with its child nodes from the tree, creating several new combinations to be tested. This is iterated until either all tests conclude that v is schedulable, or until a combination of leaves, i.e., concrete request functions, turns out to make v unschedulable.

The resulting method is precise and much more efficient than testing all possible combinations of request functions. The reason for the efficiency is that in the schedulable case, the test is likely to succeed already on rather high abstraction

[2]The point-wise maximum on request functions and the dominance relation from Definition IV.3 are a join-semilattice $(\succcurlyeq, \sqcup)$ on the request functions for each task. These semilattices are the core structure of our abstraction refinement technique.

levels. Further, in the unschedulable case, the combination of concrete request functions that violates schedulability is found in a rather guided way trough the trees down to the tree leaves since schedulable subtrees are avoided.

C. Similarity Metric

We use a similarity metric on request functions in two situations: when building the abstraction tree and when refining a combination of abstract request functions.

Building the Abstraction Tree. During construction of the abstraction tree, we want to merge the two "most similar" request functions. The effect of this is that the abstract request function representing them is a good representation of the two abstracted ones.

Abstraction Refinement. When a combination of request functions signals a potential deadline miss, we want to replace one of them with its child nodes in the corresponding abstraction tree of its task. It is beneficiary to choose the one where the child nodes are "least similar" since this will lead to rather different situations being tested next, i.e., different regions of the search space.

Formally, we define a metric on the space of request functions. It captures our intuitive notion of "distance" between two functions as representing the difference in behavior in simulated sequences.

Definition V.2. *For two request functions rf and rf' we define their* distance *on domain $[0, d]$ as*

$$dist_d(rf, rf') := \sum_{i=0}^{d} \alpha^i \cdot |rf(i) - rf'(i)|.$$

We choose to introduce a weighting factor α which results in differences in early values weighing more than in later values. The rationale is that idle intervals early in the considered synchronous arrival sequence have an overall larger effect on schedulability of a vertex. Therefore, request functions that are very similar early in the interval should be considered more alike than request functions that are rather different early in the interval and only become more similar later. In our tests, we found that a good compromise value is when early values are weighted with a factor of about 10 compared to late values. This leads to $\alpha = \sqrt[d]{0.1}$ with $\alpha^0 = 1$ and $\alpha^d = 0.1$.

D. Full Improved Algorithm

We give now the full algorithm that incorporates the abstraction refinement technique. The only change to the pseudo-code given in Section IV is the implementation of $schedulable(v, \tau)$ which we replace with $schedulable\text{-}iar(v, \tau)$ in Figure 10.

The implementation assumes a function $generate\text{-}arf(T)$ which generates the abstraction tree and returns the tree root, i.e., the most abstract request function for T. We further assume a function $refine(\bar{rf})$ which takes a combination of request functions and returns a set of combinations where one or more of the abstract request functions are replaced by child nodes from the abstraction tree(s). Further, the implementation uses a store for combinations of request functions. This could be a stack or a queue or any other data structure that implements insertion (add) and retrieval (pop) operations and a test for emptiness ($isempty$). The algorithm returns if either a combination of concrete request functions is found to make v unschedulable or if the test of all combinations concludes schedulability of v.

function $schedulable\text{-}iar(v, \tau)$:

```
1:  store ← ∅
2:  for all T ∈ τ do
3:      rf^(T) ← generate-arf(T, d(v))
4:  end for
5:  store.add(r̄f)
6:  while not store.isempty() do
7:      r̄f ← store.pop()
8:      if ∀t ⩽ d(v) : e(v) + Σ_{T∈τ} rf^(T)(t) > t then
9:          if isabstract(r̄f) then
10:             store.add(refine(r̄f))
11:         else
12:             return false
13:         end if
14:     end if
15: end while
16: return true
```

Fig. 10. Improved algorithm for schedulability of a vertex v with interference set τ.

VI. Experimental Evaluation

We evaluate our method by running it on task sets of different sizes while measuring run-times, acceptance ratios and a set of other parameters in order to show the effectiveness of our optimization techniques. We use an implementation in the Python programming language running on a standard desktop computer. The implementation is not optimized down to the last instruction, but is suitable for a qualitative comparison of scaling properties. Task sets have typical sizes of about 20 to 50 tasks and are analysed within a few seconds (while a naive enumeration approach would already take days for just five tasks). As we will see, our algorithm is very effective in preventing combinatorial explosion and scales very well even though we are dealing with a $coNP$-hard problem.

A. Task Set Generation

We define the utilization of a task T as the highest ratio of the sum of WCET vertex labels over the sum of edge labels in all cycles in $G(T)$. Each task set is randomly generated with a given goal of a task set utilization. Tasks are added to a task set until it satisfies the set goal. In order to simulate different types of tasks, we create *small tasks*, *medium tasks* and *large tasks*. These types differ in the number of vertices and in the values of their edge and vertex labels. This results in a wide range of shorter versus longer release-separation times and deadlines. The actual task parameters are selected uniformly from the intervals given in the following table. The type of each task is chosen with probability of one third.

Task Type	Small	Medium	Large
Vertices	$[3, 5]$	$[5, 9]$	$[7, 13]$
Branching degree	$[1, 3]$	$[1, 4]$	$[1, 5]$
p	$[50, 100]$	$[100, 200]$	$[200, 400]$
e	$[1, 2]$	$[1, 4]$	$[1, 8]$
d	$[25, 100]$	$[50, 200]$	$[100, 400]$

B. Run-Time Scaling

The first property we evaluate is the run-time of our method compared to the state-of-the-art EDF feasibility test from [5]. The pseudo-polynomial algorithm is dbf-based and we use an implementation in the same Python framework as our SP feasibility test. In the resulting run-time plot in Figure 11, we see that our method clearly outperforms the EDF feasibility test. For low utilizations, our method has comparable run-time and has much better scaling behavior for increasing utilizations. This is not surprising, since the run-time of the EDF feasibility test is inversely proportional to the distance to 100% utilization. Our method is less sensitive to the utilization. It is exponential in the number of tasks, but the combinatorial abstraction refinement is effective in hiding the exponential growth for the analyzed tasks.

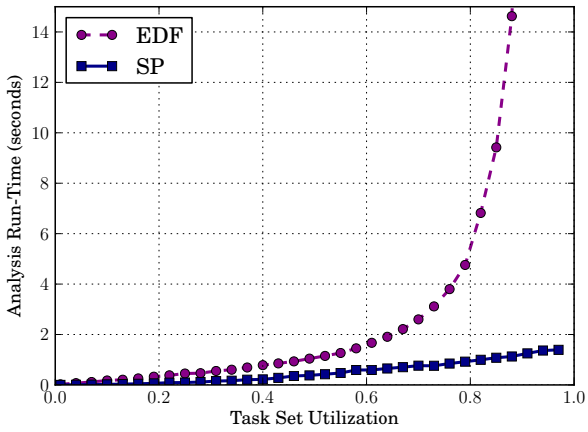


Fig. 11. Runtimes of EDF and SP feasibility analyses. Our method clearly outperforms the EDF test, both in absolute time and in average scaling.

C. Analysis Stages

We evaluate two aspects of our method in more detail. One is the time distribution between computation of all critical request functions and checking their combinations for schedulability. The other aspect is the effectiveness of the abstraction refinement in terms of avoided combination tests.

Time Distribution.: Our analysis has two phases. First, all critical request functions are derived by traversing all graphs. Second, their combinations are tested using combinatorial abstraction refinement. The first phase is linear in the utilization since it is executed in isolation for each task, and the task set size is proportional to the utilization. However, the second phase is in the worst case exponential in the number of tasks. Therefore we expect it to grow exponentially with increasing utilization.

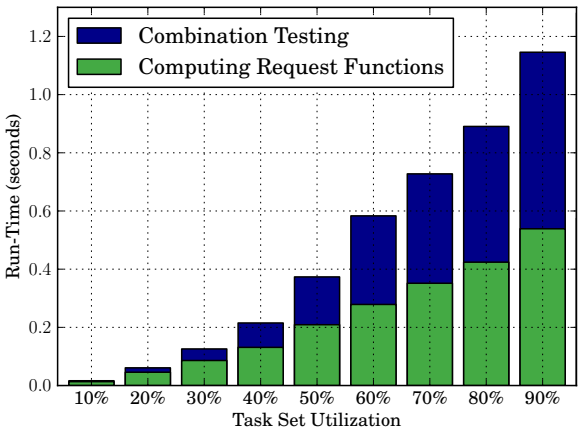


Fig. 12. Runtime of SP feasibility analysis split into computation of request functions and combination tests.

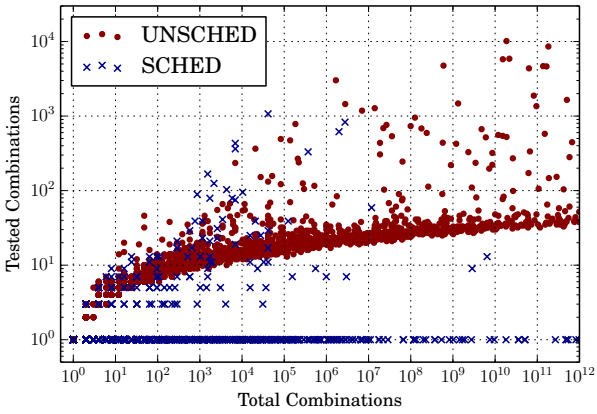


Fig. 13. Tested versus total number of combinations of request functions. Crosses represent schedulable, dots unschedulable cases. Both scales are logarithmic.

In Figure 12 we show the analysis run-time split into both phases. We see that the computation of request functions scales linearly as expected. The combination part grows more than linearly, but our abstraction refinement technique is very effective in dramatically reducing the combinatorial explosion. Even at a high utilization of 90% the abstraction refinement phase does not significantly exceed half of the analysis time.

Combination Testing.: We captured 10^5 calls to the iterative abstraction refinement procedure (Figure 10) and recorded for each call (i) how many tests where executed (Line 8 in Figure 10), (ii) how many combinations of concrete request functions there were in total and (iii) its return value. We plot the result in Figure 13 showing that our method clearly saves work in the order of several magnitudes. In more than 99.9% of all cases, less than 100 tests were executed.

D. Acceptance Ratio

As a last comparison, we look at acceptance ratios for both EDF and static priority feasibility, shown in Figure 14. Note that this comparison is *not* evaluating the quality of our analysis method. It rather compares the relative scheduling abilities of EDF versus static priority scheduling of DRT tasks. To the best of our knowledge, this is the first time that such a comparison can be made, since we present the first method that is able to efficiently and precisely analyse task sets of this size for feasibility with static priority schedulers.

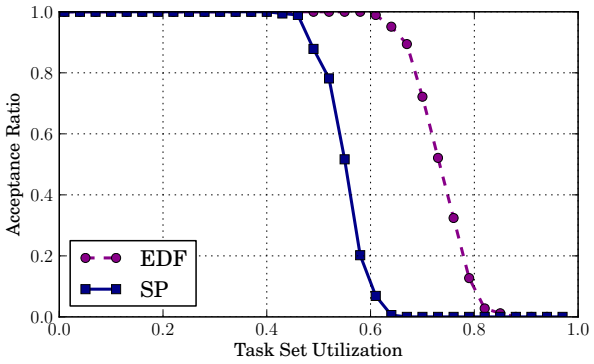


Fig. 14. Acceptance ratios of EDF and static priority schedulers.

VII. Conclusions and Future Work

In this paper, we have introduced an efficient method for analysing static priority feasibility and schedulability for DRT task sets. The method is based on different techniques for pruning significant parts of the worst-case exponential search space. Experiments have shown that our method has better performance than pseudo-polynomial algorithms for EDF feasibility.

A key part of our method is the combinatorial abstraction refinement technique. Using an abstraction lattice, it allows to quickly derive results about models which otherwise suffer from combinatorial explosion.

We believe that combinatorial abstraction refinement can be applied as a general technique to many combinatorial problems that have a certain abstraction structure which is often the case for real-time scheduling problems. As future work, we are planning to apply the technique to other problems in the theory of real-time systems. We would also like to extend the concrete algorithm presented in this paper to variants of DRT with arbitrary deadlines, static *vertex* priorities or synchronization.

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] A. K. Mok and D. Chen, "A Multiframe Model for Real-Time Tasks," *IEEE Trans. Softw. Eng.*, vol. 23, no. 10, pp. 635–645, 1997.

[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] S. K. Baruah, "A general model for recurring real-time tasks," in *Proc. of RTSS*, 1998, pp. 114–122.

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

[6] M. Stigge and W. Yi, "Hardness Results for Static Priority Real-Time Scheduling," in *Proc. of ECRTS 2012*, 2012, pp. 189–198.

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

[8] A. Zuhily and A. Burns, "Exact Scheduling Analysis of Non-Accumulatively Monotonic Multiframe Tasks," *Real-Time Systems Journal*, vol. 43, pp. 119–146, 2009.

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

[10] C.-C. J. Han, "A Better Polynomial-Time Schedulability Test for Real-Time Multiframe Tasks," in *Proc. of RTSS*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 104–.

[11] T. wei Kuo, L. pin Chang, Y. hua Liu, and K. jay Lin, "Efficient online schedulability tests for real-time systems," *IEEE Transactions On Software Engineering*, vol. 29, pp. 734–751, 2003.

[12] W.-C. Lu, K.-J. Lin, H.-W. Wei, and W.-K. Shih, "New Schedulability Conditions for Real-Time Multiframe Tasks," in *Proc. of ECRTS*, 2007, pp. 39–50.

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

[14] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Inf. Comput.*, vol. 205, no. 8, pp. 1149–1172, 2007.

[15] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS 2000*, vol. 4, 2000.

Appendix

A. Proof of Theorem IV.2

Proof: Assume Condition (1) holds but v is unschedulable. Because of the latter, there must be a combination of paths $\bar{\pi} = (\pi^{(T_1)}, \ldots, \pi^{(T_N)})$ executing in a synchronous arrival sequence for strictly more than $d(v) - e(v)$ time units within time interval $[0, d(v)]$, causing a job corresponding to v to miss its deadline. In particular, for each $t \leqslant d(v)$, tasks from τ are executing for strictly more than $t - e(v)$ time units within $[0, t]$. Since $rf_{\pi^{(T)}}(t)$ gives an upper bound for how many time units task T is executing within $[0, t]$ along path $\pi^{(T)}$, we have

$$\sum_{T \in \tau} rf_{\pi^{(T)}}(t) > t - e(v) \tag{5}$$

for each $t \leqslant d(v)$. This contradicts the assumption that Condition (1) holds.

Assume now that v is schedulable but Condition (1) does not hold. Because of the latter, there is $\bar{\pi} \in \Pi(\tau)$ such that Condition (5) holds for all $t \leqslant d(v)$. Let $t_0 \leqslant d(v)$ minimal such that τ leaves $e(v)$ time units of idle time in $[0, t_0]$ when $\bar{\pi}$ is executing in a synchronous arrival sequence. Such a t_0 must exist since v is schedulable. Thus, up to t_0, the accumulated sum of execution times of jobs released along the paths $\pi^{(T)}$ does not exceed $t_0 - e(v)$. Since there is idle time at t_0, this accumulated sum is equal to $\sum_{T \in \tau} rf_{\pi^{(T)}}(t_0)$ by Definition IV.1, so Condition (5) can not hold for this particular t_0, leading to a contradiction. ■