2014 26th Euromicro Conference on Real-Time Systems

Refinement-based Exact Response-Time Analysis

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

***Abstract*—A recent trend in the theory of real-time scheduling is to consider generalizations of the classical periodic task model. Work on the associated schedulability and feasibility problems has resulted in algorithms that run efficiently and provide exact results. While these analyses give black-and-white answers about whether timing constraints are being met or not, response-time analysis adds a quantitative dimension.**

This brings new challenges for models more expressive than the classical periodic task model. An exact quantification of response time is difficult because of non-deterministic task behavior and a lack of combinable task-local worst cases. Therefore, previous approaches all make a trade-off between efficiency and precision, resulting in either prohibitively slow analysis run-times or imprecise over-approximate results.

In this paper, we show that analysis can be both exact and efficient at the same time. We develop novel response-time characterizations to which we apply combinatorial abstraction refinement. Our algorithms for static-priority and EDF scheduling give exact results and are shown to be efficient for typical problem sizes. We advance the state-of-the-art by providing the first exact response-time analysis framework for graph-based task models.

I. Introduction

Response time analysis (RTA) characterizes the maximal delay to complete a computational request in a multitasking environment. It is useful for local schedulability analysis, but its potential uses extend to other complex design problems. An example is the case of distributed real-time systems where the completion of a task triggers the invocation of other computation or communication tasks. RTA can help to bound invocation jitters of individual tasks and enable timing analysis of the overall system [1], [2].

The classical RTA technique [3] was developed for the simple *periodic* task model, which represents a collection of independent recurrently invoked processes with certain periods. Unfortunately, behaviors that are not entirely periodic cannot be expressed accurately with this model. Important examples include variable rate-dependent behavior in controllers for fuel injection in combustion engines [4] or frame-dependent execution times in video codecs [5]. The *Digraph Real-Time (DRT)* task model [6] is a rather expressive model allowing large flexibility to express release patterns accurately by representing each task by a directed graph. It generalizes most existing models in real-time scheduling theory.

The expressiveness of DRT comes at the price of analysis complexity. As for many concurrent system models, the main challenge in the analysis of DRT is due to combinatorial explosion for problems where local worst cases do not exist. Firstly, the *number* of possible paths of each individual graph is generally unbounded, and is exponential even if restricted to bounded time intervals. Secondly, all *combinations* of paths from different graphs need to be considered to find global worst-case behavior in response-time computations.

The combinatorial explosion problem already exists in simpler special cases of DRT. Existing analysis methods either explicitly enumerate combinations, leading to extremely poor scalability, or over-approximate the workload to improve efficiency at the cost of precision loss. In this work, we develop efficient techniques of exact response-time analysis for DRT task models, for both static-priority and EDF scheduling. Our proposed techniques avoid explicit enumerations and combinatorial explosions via abstraction and refinement methods.

In particular, we make the following contributions:

- We formulate exact characterizations for the response time of DRT tasks under both static priorities and EDF.
- We develop efficient RTA algorithms by extending a combinatorial abstraction refinement technique [7] and applying it to the above formulation.

Experiments show that our algorithms reduce the necessary analysis effort to practically feasible amounts. They are therefore applicable to task systems of interesting size.

A. Prior Work

Response time analysis has been proposed first to analyze static-priority scheduling of periodic tasks [3], and was then intensively studied to deal with arbitrary deadlines [8], jitters and burst behaviors [9], [10], offsets between tasks [11], as well as dynamic-priority scheduling policies such as EDF [12], [13] and preemptive round-robin [14].

The RTA problem suffers from combinatorial explosion as soon as phase information is explicit in the task model [15], [16]. For efficient analysis, previous work either resorts to over-approximations [5], [11] or restricts to special cases with certain monotonicity properties such that exponential combinations are simply dominated by one of them [17], [18].

The feasibility problem of DRT with *dynamic* priorities can be solved in pseudo-polynomial time [6]. Significant speed-up can be achieved by exploring periodicity of workload bounds using results from max-plus algebra [19]. The corresponding problem with *static* priorities becomes strongly $coNP$-hard [20], but recently abstraction and refinement techniques have been developed to deal with the combinatorial explosion [7]. Timed automata have been used as well to model real-time workload very expressively [21], but the state-space explosion problem makes RTA using a model checker impractical.

For interval-domain workload and resource availability characterizations (arrival and service curves in Real-time Calculus [22]), RTA techniques have been developed [23], [24],

[13]. However, applying them to detailed workload characterizations as the task model used in our work is inherently over-approximate since relevant information of exact task behavior is lost in these models.

II. Preliminaries

This section provides an introduction to our task model and a problem definition.

A. Task Model

We use the *digraph real-time (DRT) task model* [6] to describe the workload of a system. A DRT task set $\tau = \{T_1, \dots, T_N\}$ consists of N independent tasks. A task T is represented by a *directed graph* $G(T)$ with both vertex and edge labels. The vertices $\{v_1, \dots, v_n\}$ 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 positive 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 positive integer $p(u, v)$ denoting the minimum job inter-release separation time. 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). Finally, given a vertex v, we denote its task with $T(v)$.

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 $\rho = [(r_1, e_1, d_1), (r_2, e_2, d_2), \dots]$ is *generated by* T, if and only if there is a (potentially infinite) path $\pi = (\pi_1, \pi_2, \dots)$ in $G(T)$ satisfying for all i:

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

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.1. *For the example task T in Figure 1, consider the job sequence $\rho = [(3, 6, 15), (15, 2.1, 25), (27, 4, 37)]$. It corresponds to path $\pi = (v_4, v_1, v_2)$ 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_1) is released as early as possible after the first job (v_4), the same is not true for the third job (v_2).

We assume that job sequences are executed on a uniprocessor system and scheduled by a preemptive scheduler. We distinguish between *dynamic* and *static* priority schedulers,

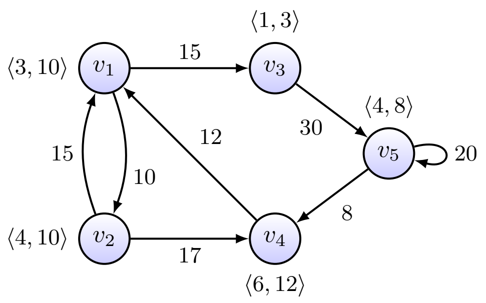


Fig. 1. An example task containing five different types of jobs

i.e., whether the scheduler has to obey a certain order of relative priorities on the task set. Our analysis focuses on two particular scheduling algorithms, *static priority (SP)* schedulers and *earliest deadline first (EDF)* schedulers.

SP scheduler: Given a priority order $\mathcal{P} : \tau \to \mathbb{N}$ which assigns a unique priority to each task, the scheduler picks the job for execution that was released by the task T with highest priority, i.e., minimal $\mathcal{P}(T)$. Our analysis for SP scheduling is presented in Section III.

EDF scheduler: The scheduler picks the job with the smallest absolute deadline, ties broken arbitrarily. This dynamic priority scheduling strategy is known to be optimal for our setting of independent jobs, i.e., if a task set can be scheduled with any scheduler, it can also be scheduled with EDF. Our analysis for EDF scheduling is presented in Section IV.

We use standard notions of schedulability and feasibility. It has been shown that for EDF, schedulability of DRT task sets can be checked in pseudo-polynomial time [6]. In contrast, the schedulability problem for the SP case is known to be strongly *coNP*-hard [20], even though the state-of-the-art algorithm based on *combinatorial abstraction refinement* [7] enables very efficient analysis even for the SP case. The algorithms presented in Sections III and IV are also based on this refinement scheme.

B. Response-Time Analysis

The main objective of this work is to compute worst-case *response times* of all jobs generated by the task system. We distinguish jobs corresponding to different vertices.

Definition II.2 (Response Time)**.** *Given a task set τ and a scheduler Sch, the* response time *$R_{Sch}(v)$ of a vertex $v \in G(T)$ for a task $T \in \tau$ is the maximal time between release and finish of any job corresponding to v for all job sequences generated by τ when scheduled with Sch.*

We briefly review the classic response-time analysis for periodic tasks and SP scheduling [3]. Consider a set τ of periodic tasks, i.e., for each $T \in \tau$ its graph $G(T)$ contains only a single vertex v and a self-loop (v, v). We write $E(T)$ for $e(v)$, $P(T)$ for $p(v, v)$ and $T' > T$ for a task T' with higher priority than T. The response time $R(T) = R_{SP}(v)$ can be computed with the following equation.

$$R(T) = \min_{t>0} \left\{ t \mid E(T) + \sum_{T'>T} \left\lceil \frac{t}{P(T')} \right\rceil E(T') \leqslant t \right\} \quad (1)$$

The term $\lceil t/P(T') \rceil \, E(T')$ characterizing interference of each higher priority task T' in a time interval of size t is a (left-continuous) step function in t. Intuitively, the minimum searches for the first time point t in which the job under consideration and all interfering work load of higher-priority tasks have finished their execution. A critical observation is that the response time of the job released by v is maximized when all other tasks release a job at the same time instant as v, with subsequent job releases as early as possible. This situation is usually called the *critical instant*.

The next two sections generalize this analysis approach to DRT task systems scheduled with SP and EDF schedulers.

III. RTA for SP Scheduling

For a generalization of the above approach, we need to cope with tasks represented by arbitrary graphs containing different types of jobs, in contrast to the relatively simple structure of periodic tasks.

Suppose we want to compute the response time of a vertex $v \in G(T)$ of some task T. For each task $T' > T$ we pick a path $\pi^{(T')}$. For our job v and the set of paths $\left\{\pi^{(T')}\right\}_{T'>T}$ we can simulate the *synchronous arrival sequence (SAS)*, i.e., a job sequence where all jobs take their maximal execution time, the job from v and the first job from each path $\pi^{(T')}$ are released at time 0, and all subsequent jobs are released as early as allowed by the edge labels. Simulating this sequence leads to a response time for this particular scenario. Figure 2 illustrates an example.

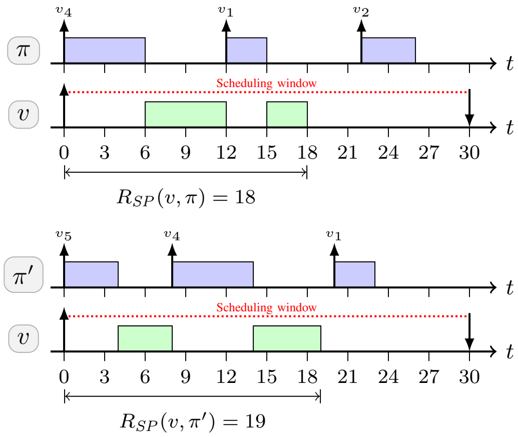


Fig. 2. Simulated synchronous arrival sequence in the analysis of a job v with $e(v) = 9$ and $d(v) = 30$. The interfering task of higher priority is task T from Figure 1. We simulate paths $\pi = (v_4, v_1, v_2)$ and $\pi' = (v_5, v_4, v_1)$. In the upper scenario, v has a response time of 18, in the lower one a response time of 19 which is in fact the worst-case response time.

An exhaustive enumeration and combination of all such paths with a simulation of the corresponding SAS in the interval $[0, d(v)]$ would be a naive approach for computing the worst-case response time of v. It is sufficient to focus on synchronous arrival sequences in this interval since jobs of a task cannot cause any direct or indirect interference to each other. This is because of strict task priority order and our assumption that all tasks are schedulable[1]. The naive approach is of course impractical, we will therefore introduce path abstractions and a refinement scheme to efficiently find a path combination obtaining the worst-case response time.

A. Request Functions

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 III.1. *For a path $\pi = (\pi_0, \ldots, \pi_l)$ 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 $e(\pi) := \sum_{i=0}^{l} e(\pi_i)$ and $p(\pi) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1})$.

In particular, $rf_\pi(0) = 0$ and $rf_\pi(1) = e(\pi_0)$, since all edge labels are strictly positive integers. We give a few examples in Figure 3.

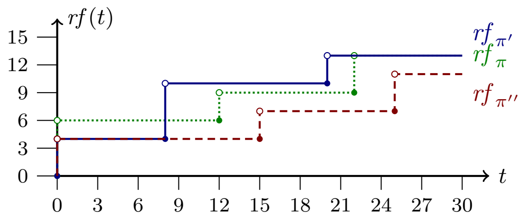


Fig. 3. Example of request functions on $[0, 30]$. Paths are taken from $G(T)$ in Figure 1 with $\pi = (v_4, v_1, v_2)$, $\pi' = (v_5, v_4, v_1)$ and $\pi'' = (v_2, v_1, v_2)$, cf. Figure 2.

Note that a periodic task T is uniquely represented by $rf(t) = \lceil t/P(T) \rceil \, E(T)$, cf. Equation (1), since it contains only one relevant path.

Let $\bar{\pi}$ denote a tuple $\left(\pi^{(T_1)}, \pi^{(T_2)}, \ldots\right)$ of paths through graphs $G(T_1), G(T_2), \ldots$ and $\Pi(\tau) = \Pi(T_1) \times \Pi(T_2) \times \ldots$ its domain. That is, $\Pi(T)$ is the set of all paths in $G(T)$ for each task T. We can express the response time of a vertex v for this *particular* combination of paths as follows.

$$R_{SP}(v, \bar{\pi}) = \min_{t>0} \left\{ t \mid e(v) + \sum_{T'>T} rf_{\pi^{(T')}}(t) \leqslant t \right\} \quad (2)$$

This is a generalization of the case for periodic tasks in Equation (1). The overall worst-case response time of v is the maximum over all such path combinations $\bar{\pi}$.

$$R_{SP}(v) = \max_{\bar{\pi} \in \Pi(\tau)} R_{SP}(v, \bar{\pi}) \quad (3)$$

[1]We assume schedulability of task sets in order to simplify presentation. Our method can be generalized to include tardy jobs.

This condition gives a formal characterization of $R_{SP}(v)$. However, two challenges remain before it can be used in an efficient implementation. First, each $\Pi(T)$ is of infinite size and contains exponentially many paths for each length. Second, the domain $\Pi(\tau)$ suffers from a combinatorial explosion, since all combinations of paths from all tasks need to be considered. We address this issue in Section III-B.

For the first challenge, we will lift the computation from individual paths to the level of request functions which are path abstractions. A crucial observation is that only a subset of all request functions representing a task is necessary to describe its worst-case behaviors. We restrict our focus to the interval $[0, d(v)]$ which is the scheduling window of v. Consider two request functions rf and rf' such that $rf(t) \geqslant rf'(t)$ for all t in $[0, d(v)]$. If rf is used in the minimum computation in Equation (2) for some path, it will result in a *larger* value than if rf' had been used instead since the LHS of the inequality will be smaller. Clearly, only rf needs to be considered.

Formally, we introduce a partial order on the set of request functions.

Definition III.2. *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.

For each task T, the set of critical request functions on a given domain can be computed efficiently as sketched in [7] based on an iterative procedure from [6].

Example III.3. *Consider again the three request functions shown in Figure 3. On domain $[0, 30]$, function $rf_{\pi''}$ is dominated by both rf_{π} and $rf_{\pi'}$ (which are actually both critical). Note that the example in Figure 2 does in fact not need to be simulated with path π'' because of this. The interference caused by path π'' is lower than that of π and π' during all time points of the considered interval $[0, 30]$.*

The computation in Equation (2) is only necessary for those combinations $\bar{\pi}$ with critical request functions, since other combinations will not contribute to the maximum in Equation (3).

We can express the worst-case response time $R_{SP}(v)$ directly in terms of critical request functions. This lifts the abstraction and allows us to work with only few request functions instead of exponentially many paths. Let $\bar{rf}$ denote a tuple $\left(rf^{(T_1)}, rf^{(T_2)}, \ldots\right)$ of request functions. For each T, let $RF(T)$ denote the set of critical request functions on domain $[0, d]$ and $RF(\tau) = RF(T_1) \times RF(T_2) \times \ldots$ all their combinations. Using this notation, Equations (2) and (3) can be expressed as follows.

$$R_{SP}(v, \bar{rf}) = \min_{t>0} \left\{ t \mid e(v) + \sum_{T'>T} rf^{(T')}(t) \leqslant t \right\} \quad (4)$$

$$R_{SP}(v) = \max_{\bar{rf} \in RF(\tau)} R_{SP}(v, \bar{rf}) \quad (5)$$

While the critical request functions abstraction dramatically reduces the number of *objects* to be considered for each task (tens of functions compared to thousands or even millions of paths), the number of *combinations* that need to be considered in Equation (5) still explodes exponentially. We will therefore now employ a method from [7] called combinatorial abstraction refinement which we adapt to the setting of response-time analysis.

B. Combinatorial Abstraction Refinement

The key idea of this abstraction scheme is that several request functions can be represented by a single one by taking their point-wise maximum. Using abstract request functions results in an over-approximate response time. However, by step-wise refinement of the abstraction, the result can be made more and more precise until it eventually is not over-approximate anymore. This leads to a dramatically improved performance without compromising precision of the result.

For the abstraction refinement, we define an abstraction semi-lattice containing the partial order from Definition III.2 together with the following join operator.

Definition III.4. *We call rf a* concrete request function *if it is derived from a path π in a graph $G(T)$ as in Definition III.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 $rf = rf_1 \sqcup \ldots \sqcup rf_k$.

As an example for an over-approximate response-time computation, consider for each T the function $mrf^{(T)}$ defined as the point-wise maximum of *all* its concrete request functions $RF(T)$. We call $mrf^{(T)}$ the *most abstract request function* for T (some authors call it the *request bound function*) and can use it for a response-time over-approximation:

$$R_{SP}(v) \leqslant \min_{t>0} \left\{ t \mid e(v) + \sum_{T'>T} mrf^{(T')}(t) \leqslant t \right\}$$

Note that the RHS is not using a maximum operator over a set of combinations. Only *one* combination of (abstract) request functions is used, the $mrf^{(T')}$ for every task T' of higher priority than T. This is very efficient, but does not necessarily compute a precise response time as Figure 4 demonstrates.

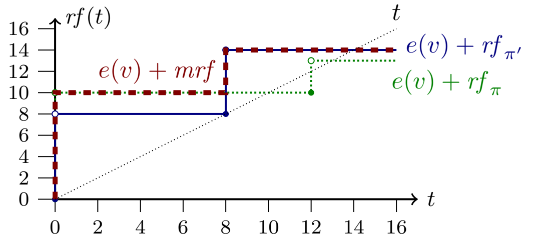


Fig. 4. Example of imprecision if only mrf is used. Consider the analysis of a vertex v with $e(v) = 4$ and $d(v) = 16$. Task T from Figure 1 interferes with paths $\pi = (v_4, v_1, v_2)$ and $\pi' = (v_5, v_4, v_1)$. As we can see, we have $R_{SP}(v, rf_{\pi}) = 10$ and $R_{SP}(v, rf_{\pi'}) = 8$, resulting in $R_{SP}(v) = 10$. However, $R_{SP}(v, mrf) = 14$, which is strictly over-approximate.

The idea is to start with such an over-approximate result and refine it step-by-step. In order to know how to refine an abstract request function, we define an *abstraction tree* for each T.

Abstraction Trees: Given the set $\{rf_1, \dots, rf_k\}$ of concrete request functions of a task T, we build an abstraction tree bottom-up. Each concrete request function is a leaf in the tree. 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" via some similarity metric [7]. 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 5 illustrates the abstraction tree.

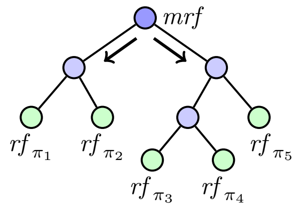


Fig. 5. Request function abstraction tree for request functions of task T in Figure 1. The leaves are all five concrete (critical) request functions on $[0, 50]$. Each inner node is the point-wise maximum of all descendants and thus an abstract request function. Abstraction refinements happen downwards along the edges, starting at the root.

Refinement Procedure: The procedure starts with just one tuple $\bar{rf}$ of abstract request functions, i.e., $(mrf^{(T_1)}, mrf^{(T_2)}, \dots)$ for computing an over-approximate response time $R_{SP}(v, \bar{rf})$. In each step, one abstract request function rf in the tuple is replaced by refinements rf' and rf'' with

$$rf \succcurlyeq rf', \quad rf \succcurlyeq rf'', \quad rf = rf' \sqcup rf''.$$

In other words, rf is replaced by its two child nodes in the corresponding abstraction tree. We call this a *split* of rf, leading to two new tuples of request functions $\bar{rf}'$ and $\bar{rf}''$ which are identical to $\bar{rf}$ except that rf is exchanged for rf' and rf'', respectively. The new response times $R_{SP}(v, \bar{rf}')$ and $R_{SP}(v, \bar{rf}'')$ are still over-approximate, but more precise than the first one. Of the current set of tuples, we always split the one with the *largest* response time over-approximation. The splitting results in more and more tuples in the current set and eventually, some of them will consist of only *concrete* request functions. The procedure ends as soon as one of these tuples of concrete request functions is the maximum of any tuple in the current set. We give our refinement procedure for computing static priority response times in Figure 6.

The procedure in Figure 6 uses a priority queue PQ for storing tuples of request functions. In order to add a value to PQ, the function $PQ.add(value, priority)$ expects a priority, so that the head of PQ always contains the value with the highest priority. We assume a function $generate\text{-}mrf(T, d)$ which computes the abstraction tree for task T and returns $mrf^{(T)}$ on domain $[0, d]$.

```
function $compute\text{-}R_{SP}(v)$ :
 1: $PQ \leftarrow \emptyset$
 2: for all $T' > T(v)$ do
 3:     $rf^{(T')} \leftarrow generate\text{-}mrf(T', d(v))$
 4: end for
 5: $PQ.add(\bar{rf}, R_{SP}(v, \bar{rf}))$
 6: while $isabstract(PQ.head)$ do
 7:     $\bar{rf} \leftarrow PQ.pophead()$
 8:     $(\bar{rf}', \bar{rf}'') \leftarrow refine(\bar{rf})$
 9:     $PQ.add(\bar{rf}', R_{SP}(v, \bar{rf}'))$
10:     $PQ.add(\bar{rf}'', R_{SP}(v, \bar{rf}''))$
11: end while
12: return $R_{SP}(v, PQ.head)$
```

Fig. 6. Algorithm for computing worst-case response time $R_{SP}(v)$ of a vertex v.

Correctness: The worst-case response time returned by procedure $compute\text{-}R_{SP}(v)$ is exact. The intuition behind this is as follows. The procedure starts with a combination $\bar{rf}$ of all most abstract request functions. The obtained value $R_{SP}(v, \bar{rf})$ is a safe upper bound for $R_{SP}(v)$. Each time we split a request function in order to obtain two new tuples containing combinations of abstract request functions, the maximal $R_{SP}(v, \bar{rf})$ of any $\bar{rf}$ in PQ is still a safe upper bound for $R_{SP}(v)$. The reason for this is that each combination of leafs from all abstraction trees, i.e., each combination of concrete request functions, is always dominated by one combination $\bar{rf}$ in PQ. This implies the invariant. Note that the maximum value is achieved with the tuple stored in $PQ.head$. Eventually, there will be a combination $\bar{rf}$ containing only concrete request functions and with a value $R_{SP}(v, \bar{rf})$ equal to the current maximum over all tuples in PQ. This means that there must be a job sequence, represented by this particular $\bar{rf}$, which leads to a response time of v which is equal to the current safe upper bound in PQ. Thus, this upper bound is indeed tight, since there is a concrete example. In this case, the algorithm terminates and its return value is equal to $R_{SP}(v)$. A more formal and detailed proof can be found in [25].

IV. RTA FOR EDF SCHEDULING

In this section we adjust the above response-time analysis method to the setting of an EDF scheduler. The major difference is the response-time computation given a combination of paths (cf. Equation 4) which needs to be changed and extended similar to [13]. However, we will see that the basic refinement framework from above can be reused in order to obtain exact results from initial over-approximate estimations.

A. Workload Functions

In SP scheduling, the static priority order of tasks directly determines which jobs may cause interference to each other: each job of a task of higher priority may preempt any job from a task of lower priority and thus delay its execution and increase its response time. There is never any interference in the opposite direction, from low to high priorities. Based on this property, we were able to use request functions in order to quantify interference.

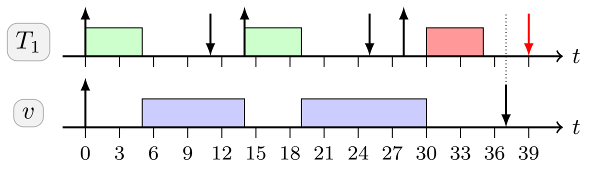
In EDF scheduling, the situation changes. When analysing the response time of a particular job, certain jobs from other tasks may cause interference but others do not, at least not in the same release scenario. Consider the example in Figure 7(a). Illustrated is the response-time analysis of v with $e(v) = 20$ and $d(v) = 37$ with an interfering periodic task. In this scenario, only the first two jobs of the interfering task T_1 do interfere with v since the third one has a later deadline than v and therefore lower priority under EDF. In order to quantify this interference, we would like to use a function similar to a request function, but it should be counting only jobs with higher priority, i.e., earlier absolute deadline. Thus, such a function needs to consider deadlines instead of release times. We define a *demand function* for a path π to return the accumulated execution requirement of all jobs of π that are released and have a deadline during the first t time units.

Definition IV.1. *For a path $\pi = (\pi_0, \dots, \pi_l)$ through the graph $G(T)$ of a task T, we define its* demand function *as*

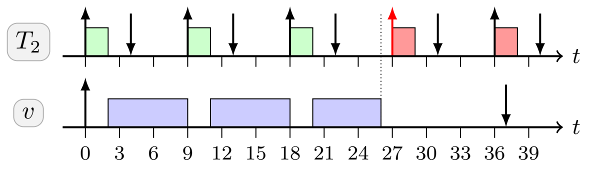
$$df_\pi(t) := \max\{e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } d(\pi') \leqslant t\}$$

where $d(\pi) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + d(\pi_l)$ and $e(\pi)$ as before.

However, using just a demand function does still not exactly quantify the interference since it may also be over-approximate. Consider the example in Figure 7(b). In this second scenario, only the first three jobs of the interfering task T_2 do interfere with v. Unlike the fifth job, the fourth one has its deadline before $d(v)$ and is therefore included in a demand function. However, it also does not cause interference since it is actually released after the finish time of v. Thus, only the first three jobs should be counted when trying to exactly quantify the maximal interference.



(a) T_1 with $P(T_1) = 14$, $D(T_1) = 11$ and $E(T_1) = 5$.



(b) T_2 with $P(T_2) = 9$, $D(T_2) = 4$ and $E(T_2) = 2$.

Fig. 7. Two different scenarios for a job v to experience interference from periodic tasks T_1 or T_2. Not all of their jobs that are released during the scheduling window of v do actually cause interference.

The solution is to define for each path a function which counts only jobs with a deadline earlier than the deadline of v but also with a release time earlier than the finish time of v. A *workload function* has therefore a two-dimensional domain. Intuitively, the first argument t represents the supposed finish time of v and jobs from π that are counted should be released earlier than t. The second argument t' represents the deadline of v and jobs from π that are counted should have their deadline latest at t'.

Definition IV.2. *For a path $\pi = (\pi_0, \dots, \pi_l)$ through the graph $G(T)$ of a task T, we define its* workload function *as*

$$wf_\pi(t, t') := \max\{e(\pi') \mid \pi' \text{ is prefix of } \pi, \; p(\pi') < t \text{ and } d(\pi') \leqslant t'\}.$$

Example IV.3. *Consider the periodic task T_1 in Figure 7(a). We have $wf(30, 37) = 10$ since two jobs of 5 execution time units each are released within 30 time units and have their deadlines within 37 time units. As a consequence, the exact interference that v experiences is 10 time units and it finishes after 30 time units. Note that we leave out to write a path π in wf_π since a periodic task has only one (infinite) path.*

As a second example, consider the periodic task T_2 in Figure 7(b). In this case we have $wf(26, 37) = 6$, counting the first three jobs of T_2, since these are the only ones being released within 26 time units and having a deadline within 37 time units. Therefore, v experiences 6 time units of interference and finishes after 26 time units.

Workload functions are a generalization of request functions as well as demand functions, since

$$rf_\pi(t) = wf_\pi(t, \infty) \text{ and } df_\pi(t) = wf_\pi(\infty, t).$$

In fact, a workload function can be derived from request and demand functions of the same path, which allows compact representation in memory.

Lemma IV.4. *For a path π through a graph $G(T)$, the functions rf_π, df_π and wf_π are related via*

$$\forall t, t' \geqslant 0 : wf_\pi(t, t') = \min(rf_\pi(t), df_\pi(t')).$$

Proof: Given t, t' and a path π, let π'_{rf} and π'_{df} be the paths used in the maximum for the computations of $rf_\pi(t)$ and $df_\pi(t')$, respectively. Both paths are prefixes of π and we have two cases depending on their lengths.

Case 1: π'_{rf} is a prefix of π'_{df}. Thus, $rf_\pi(t) \leqslant df_\pi(t')$. Further, $d(\pi'_{rf}) \leqslant d(\pi'_{df})$ since deadlines are constrained and therefore ordered monotonically. Hence $wf_\pi(t, t') \geqslant e(\pi'_{rf}) = rf_\pi(t)$ since π'_{rf} satisfies both conditions in the definition of wf. However, any prefix π' of π longer than π'_{rf} has $p(\pi') \geqslant t$, therefore $rf_\pi(t)$ is the maximal value for $wf_\pi(t, t')$.

Case 2: π'_{df} is a (strict) prefix of π'_{rf}, same proof but with swapped roles of df and rf. ■

We can now do a first attempt to use workload functions for response time analysis. Consider a vertex $v \in G(T)$ from a task T for which we want to compute the worst-case response time with an EDF scheduler and pick a workload function $wf^{(T)}$ for each task T. We search for the first time instant t_f at which a job v does finish its execution. While doing so, we need to consider interference of jobs from other tasks which

1) are released strictly before t_f and
2) have their deadline latest at $d(v)$.

The following expression captures this intuition and is very similar to Equation 4 for SP scheduling.

$$t_f = \min_{t>0} \left\{ t \mid e(v) + \sum_{T' \neq T} wf^{(T')}(t, d(v)) \leqslant t \right\} \qquad (6)$$

B. Busy Window Extension

As we will see now, this t_f does *not* correctly capture the worst-case response time of v. The underlying assumption is that a synchronous arrival sequence creates the maximal interference for a job. Unfortunately, this assumption does not hold in the EDF case as Figure 8 shows.

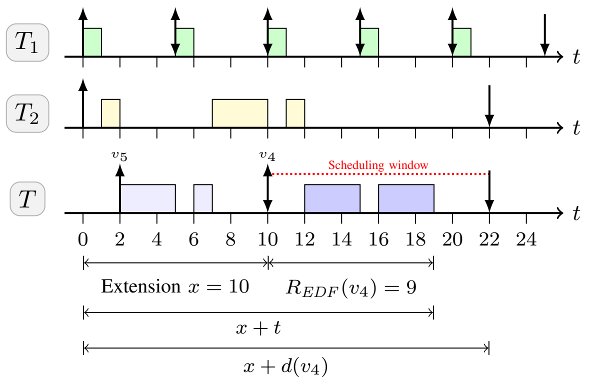


Fig. 8. Example demonstrating that the worst-case response time may be achieved by a job sequence which is *not* an SAS. The schedule shows the response time calculation of v_4 in task T from Figure 1, preempted by two periodic tasks T_1 and T_2. T first releases a job from v_5 at time $t = 2$ followed by the job v_4 to be analyzed at $t = 10$. For the busy window extension we have $x = 10$ which leads to the worst-case response time $R_{EDF}(v_4) = 9$. In an SAS, v_4 would finish after already 8 time units.

In order to solve this problem, we use the common *busy window extension* technique, summarized as follows. The worst-case response time of v is caused by an arrival sequence where all tasks $T' \neq T(v)$ synchronously release their jobs at some time point *before* the release of v. Together with jobs from $T(v)$, the processor is kept continuously busy until v is released and finally finishes. This period is called the *busy window*. It is difficult to predict the exact size of the busy window leading to the worst-case response time of v, but an upper bound can be given by computing the maximal size of any busy window[2] for τ. With this upper bound, all possibilities can be enumerated.

We now give details of this procedure. We consider extensions of the analyzed window, which initially is just the scheduling window of v. We extend this window by additional x time units into the past, i.e., to the left, cf. Figure 8. For each integer $x \geqslant 0$, we analyze a scenario in which all tasks $T' \neq T$ start releasing their jobs at time 0 while the job corresponding to v is released at time x and has its deadline at $x + d(v)$, i.e., its scheduling window is the interval $[x, x + d(v)]$. In addition to this job, T releases other jobs *before* time x as late as possible, i.e., corresponding to a path through $G(T)$ *ending* in v. This construction maximizes the interference experienced by v for a particular x. In Figure 8 this is the case for $x = 10$ where T is following path (v_5, v_4). In order to find the worst-case response time, all x are enumerated up to the size L of the maximal busy window.

Formally, we can use workload functions as before to describe the interference to v by tasks $T' \neq T$. The value of $wf_\pi(x+t, x+d(v))$ for a path captures the interference of all jobs along that path that

1) are released within the extended busy window before a supposed finish time $x + t$ of v, but
2) still have their deadline latest at $x + d(v)$ so that they actually interfere with v because of having a higher priority under EDF.

In addition to that, we also need to have a way of describing the interference of jobs from T that are released *before* v, since they can cause indirect interference (by delaying jobs of some T' which then interfere with v). For this purpose, we define a *suffix demand function* which is defined like a demand function but ensures that the *last vertex* of a path is always included in the workload.

Definition IV.5. *For a path $\pi = (\pi_0, \dots, \pi_l)$ through the graph $G(T)$ of a task T, we define its* suffix demand function *as*

$$df_\pi^{sfx}(t) := \max\{e(\pi') \mid \pi' \textit{ is suffix of } \pi \textit{ and } d(\pi') \leqslant t.\}$$

Example IV.6. *We give an example in Figure 9.*

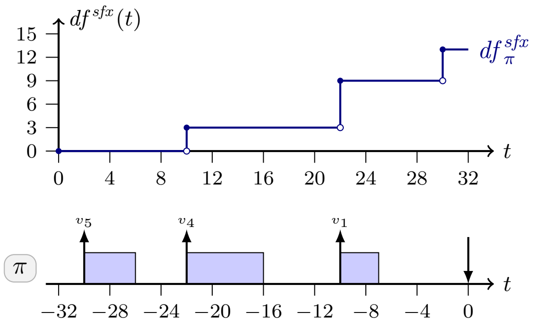


Fig. 9. Example of a suffix demand function on $[0, 30]$ for $\pi = (v_5, v_4, v_1)$ from $G(T)$ in Figure 1. We also show the job sequence in a graphical schedule, with time 0 being the deadline for the job from v_1. This illustrates that df_π^{sfx} builds *backwards* from this deadline.

We are now finally ready to give the full method for computing the worst-case response time of a vertex $v \in G(T)$ for a task T. Let $\overline{wf}$ be a vector containing

- a workload function $wf^{(T')}$ for each task $T' \neq T$ describing its workload interfering with T, and
- for task T a function $f(t, t')$ defined as

$$f(t, t') := \max(df^{sfx}(t'), e(v)).$$

This function expresses the workload of T and always contains the job of v itself. All jobs released by T are

[2]A bound of the maximal busy window size can be easily computed by finding smallest interval size t for which $\sum_{T \in \tau} mrf^{(T)}(t) \leqslant t$.

always implicitly released before time t which therefore is discarded, i.e., just a dummy argument. The role of this function is just to unify notation so that we can represent all functions in vector $\bar{wf}$ containing only functions in two arguments.

Using $\bar{wf}$ we can express the response time $R_{EDF}(v, \bar{wf}, x)$ of v for this particular $\bar{wf}$ and a particular x. The worst-case response time $R_{EDF}(v)$ of v is derived by maximizing over all x and $\bar{wf}$. Note that the maximum is over all $\bar{wf}$ that can be constructed from individual $wf^{(T)}$ and df^{sfx} as described above.

$$R_{EDF}(v, \bar{wf}, x) := \min_{t>0} \left\{ t \mid \sum_{T\in\tau} wf^{(T)}(x+t, x+d(v)) \leqslant x+t \right\} \tag{7}$$

$$R_{EDF}(v) = \max_{\bar{wf}} \max_{x\leqslant L} R_{EDF}(v, \bar{wf}, x) \tag{8}$$

C. Refinement Procedure

As for the SP scheduler case, enumerating all combinations of workload functions and suffix demand functions leads to combinatorial explosion. However, we can employ the same combinatorial abstraction refinement technique as in Section III. Abstract workload functions and suffix demand functions can be defined just as in the case of request functions. Their abstraction trees are also built in a similar way. They can be stored efficiently since for abstract workload functions, we may choose to just store pairs of abstract request and demand functions instead. Because of Lemma IV.4, these can be used to derive corresponding abstract workload functions using the minimum operator.

In summary, the full algorithm is given in Figure 10.

```
function $compute\text{-}R_{EDF}(v)$ :
 1: $PQ \leftarrow \emptyset$
 2: $wf^{(T(v))} \leftarrow generate\text{-}mdf^{sfx}(T(v), L + d(v))$
 3: for all $T' \neq T(v)$ do
 4:     $wf^{(T')} \leftarrow generate\text{-}mwf(T', L + d(v))$
 5: end for
 6: $PQ.add(\bar{wf}, \max_{x\leqslant L} R_{EDF}(v, \bar{wf}, x))$
 7: while $isabstract(PQ.head)$ do
 8:     $\bar{wf} \leftarrow PQ.pophead()$
 9:     $(\bar{wf}', \bar{wf}'') \leftarrow refine(\bar{wf})$
10:     $PQ.add(\bar{wf}', \max_{x\leqslant L} R_{EDF}(v, \bar{wf}', x))$
11:     $PQ.add(\bar{wf}'', \max_{x\leqslant L} R_{EDF}(v, \bar{wf}'', x))$
12: end while
13: return $\max_{x\leqslant L} R_{EDF}(v, PQ.head, x)$
```

Fig. 10. Algorithm for computing worst-case response time $R_{EDF}(v)$ of a vertex v.

The structure is very similar to the algorithm for SP scheduling in Section III. It is assumed that L is the size of the maximal busy window. Further, functions $generate\text{-}mwf(T, d)$ and $generate\text{-}mdf^{sfx}(T, d)$ compute abstraction trees for a task T on domain $[0, d]$ for all workload functions and suffix demand functions, respectively. Efficient implementations are similar to those for request functions [7]. They return the tree roots to be used for the first over-approximate estimate and later refinements, i.e., splitting. The major difference to the SP algorithm is that each response-time computation maximizes over all possible busy window extensions $x \leqslant L$.

D. Optimizations

Our algorithm for computing the worst-case response time under EDF in this section is slower than the one we present in Section III for SP. One of the reasons is that for any given combination $\bar{wf}$ of (abstract) workload functions, all $x \leqslant L$ need to be enumerated and $R_{EDF}(v, \bar{wf}, x)$ needs to be computed via Equation (7) again for each x. Evaluating Equation (7) so many times can be costly if L is large. Thus, we employ an optimization that allows to skip a large fraction of possible values for x.

The key idea is to use a simple over-approximation $R^*_{EDF}(v, \bar{wf}, x)$ of the response time $R_{EDF}(v, \bar{wf}, x)$.

$$R^*_{EDF}(v, \bar{wf}, x) := \sum_{T\in\tau} wf^{(T)}(\infty, x + d(v)) - x$$

Clearly, $R_{EDF}(v, \bar{wf}, x) \leqslant R^*_{EDF}(v, \bar{wf}, x)$, since R^* even counts all jobs from other tasks until the deadline of v, including those released after its finish time, cf. Figure 7(b). While searching for the maximal value of $R_{EDF}(v, \bar{wf}, x)$ over all x, we only need to do an exact evaluation of $R_{EDF}(v, \bar{wf}, x)$ if $R^*_{EDF}(v, \bar{wf}, x)$ is strictly larger than the current maximum over those x already considered. If $R^*_{EDF}(v, \bar{wf}, x)$ is smaller than the current maximum, then an exact evaluation will not find a new maximum either, so we can directly skip to the next value of x.

Another possible optimization is based on the property of rf and df to be step functions. Consider an expression with structure $f(x) - x$ like the above for R^*_{EDF}. If f is piecewise constant and we are looking for maximal values of that expression for increasing x, then it is sufficient to evaluate the expression for x where f changes. This can be done for the above optimization using R^*_{EDF}, but similar optimizations can be applied to the search for maxima with R_{SP} and R_{EDF}.

V. Experimental Evaluation

For evaluating our response-time analysis algorithms, we run them on synthetic task sets of different sizes. We use an implementation in the Python programming language. This prototype is not the most optimized in terms of speed, but is suitable for qualitative comparisons of scaling and other interesting properties. We evaluate two key properties of our method.

Efficiency. In order to evaluate efficiency of our method, we investigate how many combinations of path abstractions are actually tested compared to the total number of possible combinations. As an effect of this, we further evaluate how the run-time changes when task sets change in size and utilization.

Precision improvement. We evaluate how much the final exact worst-case response time value changes compared to the first imprecise estimate where only the most abstract functions are used.

A. *Task Set Generation*

Task set size and its utilization are directly related: 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 $G(T)$. The system utilization is the sum of the task utilizations and therefore increases with the number of tasks. 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 given goal.

A task is generated randomly as follows. A random number of vertices is created with edges connecting them according to a specified branching degree ("fan-out"). Edges are placed such that the graph is strongly connected. After choosing edge labels with uniform probability, deadlines are chosen randomly with a uniformly chosen ratio to the minimal outgoing edge label. Finally, execution time labels are chosen randomly with a uniformly chosen ratio to the vertices deadlines. The following table gives details of the used parameter ranges.

Vertices	Fan-out	p	d/p	e/d
$[5, 10]$	$[1, 3]$	$[100, 300]$	$[0.5, 1]$	$[0, 0.07]$

Feasible task sets created by this method have sizes up to about 20 tasks with over 100 individual job types in total. Note that we only consider task sets which are feasible for SP or EDF scheduling, i.e., do not miss any deadlines. This is a basic assumption for the correctness of our method. However, this is not a fundamental limitation, since our analysis can be extended to cope with tardy jobs.

B. *Efficiency*

Work Reduction: For evaluating the effectiveness of the abstraction refinement scheme, we capture for each call to the algorithms shown in Figures 6 and 10 how many combinations of request or workload functions have been *tested*. We compare this number with the *total* number of their combinations. This ratio indicates how much computational work the refinement scheme saves, compared to a naive brute-force style test. We capture 10^5 samples and show our results in Figure 11.

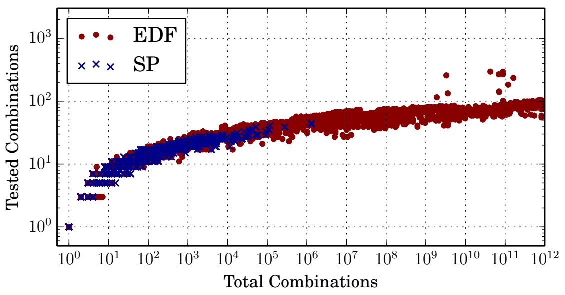
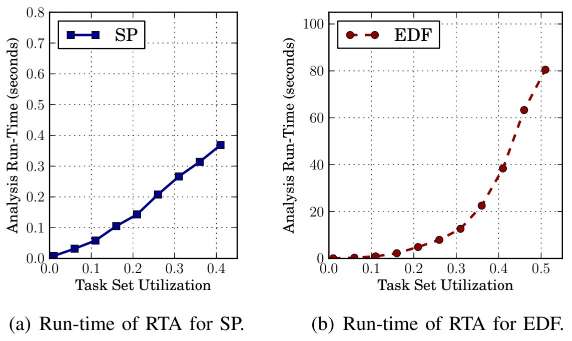


Fig. 11. Tested versus total number of combinations of path abstractions. Crosses represent SP, dots EDF cases. Both scales are logarithmic.

We see that the combinatorial abstraction refinement scheme saves work in the order of several magnitudes. Except for a few cases, no more than 100 tests have been executed. The improvement is more dramatic for the EDF case since more path abstractions represent each task. The refinement scheme is effective in masking this effect.

Run-Time Scaling: We evaluate how increasing sizes of task sets and their utilization influences run-time of our method. Since the problem is fundamentally exponential in complexity, we expect also the run-time to increase exponentially. However, the abstraction refinement technique is expected to be effective in hiding the exponential factor which is due to combinatorial explosion. We plot the run-time results in Figure 12.



(a) Run-time of RTA for SP. (b) Run-time of RTA for EDF.

Fig. 12. Run-times of SP and EDF response-time analyses. We only consider feasible task sets, leading to a cut-off at about 40% and 50% utilizations, respectively.

In the SP case in Figure 12(a), we note that the run-time is very low even for the largest task sets that are feasible. With our set of parameters, the schedulability phase-change is at about 40%, i.e., task sets with higher utilization are not schedulable and therefore not eligible for our RTA computation. We see that up to this point, the abstraction refinement technique is effective in hiding most of the exponential increase in run-time.

In the EDF case in Figure 12(b), the run-time of our algorithm is longer by about two orders of magnitude. We also note a clear exponential increase in run-time. This is due to several contributing factors. Higher utilization results in longer sizes of the maximal busy window. This means that the busy window extension to be considered grows with increasing utilization. Further, the domain of workload functions increases because of this, leading to increased computational effort for deriving them. Finally, the set of critical workload functions is generally larger than the set of critical request functions (used in the SP case), contributing further to increased complexity of the analysis. All three factors have an exponential effect that is independent of the issue of combinatorial explosion which the abstraction refinement scheme is designed to solve.

C. *Precision Improvement*

During the refinement procedure, the worst-case response time estimate is improved further and further until the algorithm terminates with a precise value. We evaluate how significant this improvement is. We compare two types of task sets which are created with different sets of parameters. Task sets of type A are created with the same parameters as for the experiments above. Task sets of type B are created with a significantly larger interval of possible edge labels. This impacts also choices of deadlines and WCET labels.

Type	Vertices	Fan-out	p	d/p	e/d
A	$[5, 10]$	$[1, 3]$	$[100, 300]$	$[0.5, 1]$	$[0, 0.07]$
B	$[5, 10]$	$[1, 3]$	$[10, 300]$	$[0.5, 1]$	$[0, 0.07]$

The idea is to demonstrate that large intra-task differences in labels have an impact on the precision of the initial estimate. For each generated task set of either type, we record the fraction of jobs for which the refinement procedure improved upon the initial over-approximate estimate. Further, for each job where this was the case, we record how large the improvement was. For both types A and B, we run 250 tests each with EDF RTA of task sets with 10 tasks and plot the results in Figure 13.

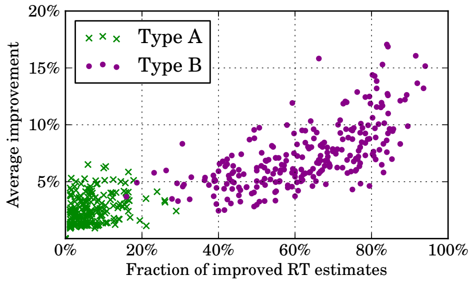


Fig. 13. Precision improvement for two different types of task sets.

A clear difference can be seen between both types of task sets. The precision improvement for type A task sets is both seldom and small. For only up to about 20% of all jobs, the initial estimate is actually over-approximate, and if it is, the precise one is only at most about 5% lower. On the other hand, type B task sets improve the estimate for roughly between 40% and 90% of all jobs for each task, and the average improvement per job is up to about 15% and beyond.

This comparison tells us that our refinement scheme has the potential to significantly improve response-time estimates that were only rough over-approximations with previous methods. This is especially the case if jobs of different types have very different WCET bounds and inter-release delays. Further, since we provide the first tractable exact method, it is now possible to characterize workload for which over-approximate estimates are already rather precise. The more equal WCET bounds and inter-release delays are within tasks, the higher the precision.

VI. Conclusions and Future Work

This paper proposes an exact response-time analysis framework based on combinatorial abstraction refinement. Novel characterizations of worst-case response times for DRT task sets are the key for efficient analysis, taking advantage of the speed-up provided by our refinement approach. Experiments show that our method provides good performance when analyzing task sets of interesting size, demonstrating that combinatorial abstraction refinement is applicable beyond feasibility analysis. Our result leads to important insights about precision of response-time over-approximations. These insights were not possible before as exact and efficient tests were not available. In future work, we would like to use more domain insights in order to speed up the process even further. We would also like to explore applications of this framework to different extensions of the DRT task model.

References

[1] K. Tindell, "Holistic schedulability analysis for distributed hard real-time system," in *Microprocessing and Microprogramming*, 1994.

[2] J. Rox and R. Ernst, "Compositional performance analysis with improved analysis techniques for obtaining viable end-to-end latencies in distributed embedded systems," in *STTT (15) 3*, 2013, pp. 171–187.

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

[4] R. I. Davis, T. Feld, V. Pollex, and F. Slomka, "Schedulability Tests for Tasks with Variable Rate-Dependent Behaviour under Fixed Priority Scheduling," in *Proc. of RTAS*, 2014, to appear.

[5] 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.

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

[7] M. Stigge and W. Yi, "Combinatorial Abstraction Refinement for Feasibility Analysis," in *Proc. of RTSS*, 2013, pp. 340–349.

[8] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proc. of RTSS*, 1990, pp. 201–209.

[9] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," in *Software Engineering Journal*, 1993.

[10] K. Richter, "Compositional Scheduling Analysis Using Standard Event Models - The SymTA/S Approach," in *Ph.D. dissertation, University of Braunschweig*, 1999.

[11] J. P. Gutierrez and M. G. Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets," in *Proc. of RTSS*, 1998, pp. 26–37.

[12] M. Spuri, "Analysis of Deadline Scheduled Real-Time Systems," in *RR-2772, INRIA, France*, 1996.

[13] N. Guan and W. Yi, "General and Efficient Response Time Analysis for EDF Scheduling," in *Proc. of DATE*, 2014, pp. 1–6.

[14] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst, "Improved response time analysis of tasks scheduled under preemptive Round-Robin," in *Proc. of CODES+ISSS*, 2007, pp. 179–184.

[15] K. Tindell, "Adding Time-Offsets to Schedulability Analysis," in *Technical Report YCS 221, University of York*, 1994.

[16] A. Zuhily and A. Burns, "Exact scheduling analysis of non-accumulatively monotonic multiframe tasks," in *Real-Time Syst.*, 2009, pp. 119–146.

[17] ——, "Exact Response Time Scheduling Analysis of Accumulatively Monotonic Multiframe Real Time Tasks," in *Proc. of ICTAC*, 2008, pp. 410–424.

[18] K. Traore, E. Grolleau, and F. Cottet, "Characterization and Analysis of Tasks with Offsets: Monotonic Transactions," in *Proc. of RTCSA*, 2006, pp. 10–16.

[19] H. Zeng and M. D. Natale, "Using Max-Plus Algebra to Improve the Analysis of Non-cyclic Task Models," in *Proc. of ECRTS*, 2013, pp. 205–214.

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

[21] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES - A Tool for Modelling and Implementation of Embedded Systems," in *Proc. of TACAS 2002*. Springer-Verlag, 2002, pp. 460–464.

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

[23] M. Naedele, L. Thiele, and M. Eisenring, "Characterizing Variable Task Releases and Processor Capacities," in *TIK-Report 45, Engineering and Networks Lab, ETHZ*, 1999.

[24] V. Pollex, S. Kollmann, and F. Slomka, "Generalizing Response-Time Analysis," in *Proc. of RTCSA*, 2010, pp. 203–211.

[25] M. Stigge, "Real-Time Workload Models: Expressiveness vs. Analysis Efficiency," Ph.D. dissertation, Uppsala University, Sweden, 2014.