Delay Analysis of Structural Real-Time Workload

Nan Guan[1,2], Yue Tang[1], Yang Wang[1] and Wang Yi[2]

[1] Northeastern University, China
[2] Uppsala University, Sweden

***Abstract*—In many complex embedded systems, real-time workload is generated conforming certain structural constraints. In this paper we study how to analyze the delay of real-time workloads of which the generation pattern can be modeled by task graph models. We first show that directly combining path abstraction technique (PAT) in real-time scheduling theory and real-time calculus (RTC) can provide safe delay bounds, but the results are typically over-pessimistic. Then we propose new algorithms to efficiently and precisely solve the delay analysis problem. Experiments with randomly generated task systems are conducted to evaluate the performance of the proposed methods.**

I. Introduction

Traditionally, real-time workload is modeled as collections of periodically repeating computational requests [9], [1]. However, behaviors that are not entirely periodic cannot be expressed accurately with this simple periodic task model. Examples include variable rate-dependent behavior in controllers for fuel injection in combustion engines [7] and frame dependent execution times in video codecs [10]. The workload of these systems are subject to certain structural constraints. A natural representation of these structures is a task graph. Over years, more and more expressive graph-based task models are proposed to precisely describe complex embedded real-time systems [10], [4], [2], [3], [12], [13].

In this paper, we study the problem of bounding the worst-case delay of real-time workloads generated by task graph models. Note that a task graph can generate different types of *jobs*, and our target is to derive delay bounds for individual job types. A special case of this problem has been studied in [14], where a released workload is not allowed to be accumulated over the release time of the next one, i.e., tasks have constrained deadlines. In this paper, we break this constraint and study the delay analysis problem in the general form without the constrained-deadline limitation.

The analysis of real-time task graph models is a difficult problem. This is mainly because a graph contains exponentially many different paths in a certain time scale, and it is computationally intractable to evaluate the workload generated along each path. Refinement-based techniques [15], [14] have been developed to improve the analysis efficiency for the special case of constrained deadlines. However, these analysis methods (i) still have exponential complexity and cannot handle large-scale systems (ii) are not applicable to the general case where a released workload may be processed after the release of its successors.

Real-Time Calculus (RTC) is a powerful framework for general delay analysis of real-time workload streams. RTC uses *variability characterization curves* [16] to model workload and resource, with which the performance characteristics such as delay and backlog can be derived in a very intuitive way. However, the variability characterization curve is a very general abstraction which cannot express any structure or type information of the workload.

In this paper, we integrate insights from real-time scheduling theory and RTC to analyze the delay of real-time workloads generated by task graph models. We first show that the problem can be solved by directly combining the *path abstraction technique* (PAT) [12], [11] and RTC: PAT derives the variability characterization curves to represent the workload generated by each task graph, and these curves are fed into RTC to calculate the delay bounds. However, such a naive combination of PAT and RTC in general leads to over-pessimistic results.

As the main contribution of this paper, we proposed new algorithms to efficiently and precisely solve the delay analysis problem, which address the pessimism in the native method directly combining PAT and RTC. Our algorithms have a pseudo-polynomial time complexity, and can handle large-scale task systems in very short time. We conduct experiments with randomly generated task systems to evaluate the performance of the proposed algorithms.

II. Preliminaries

A. Digraph Task Model

A task set consists of N independent Digraph Real-Time (DRT) tasks $\{T_1, \cdots, T_N\}$. Tasks are sorted in priority from high to low, i.e., T_1 has the highest priority. A task T is represented by a directed graph $G(T) = (V(T), E(T))$ with $V(T)$ denoting the set of vertices and $E(T)$ the set of edges of the graph. The vertices $V(T) = \{v_1, \cdots, v_n\}$ represent the types of all jobs that can be released by T. Each vertex v is labeled with $e(v)$, which denotes the worst-case execution time (WCET) of the job type. The edges of $G(T)$ represent the order in which job instances generated by T are released. Each edge $(u, v) \in E(T)$ is labeled with $p(u, v) \in \mathbb{N}$ denoting the minimum inter-release separation time between u and v.

A job instance $\overline{v}$ of type v is represented by a tuple (r, e) consisting of an absolute release time r and an execution time e. The semantics of a DRT task system is defined as the set of *job sequences* it may generate: $\sigma = [(r_0, e_0), (r_1, e_1), ...]$ is a job sequence if all job instances are monotonically ordered by release times, i.e., $r_i \leq r_j$ for $i \leq j$. A job sequence $\sigma = [(r_0, e_0), (r_1, e_1), ...]$ is generated by T if $\pi = (v_0, v_1, \cdots)$ is a path in $G(T)$ and for all $i \geq 0$: $r_{i+1} - r_i \geq p(v_i, v_{i+1})$ and

978-3-9815370-4-8/DATE15/©2015 EDAA

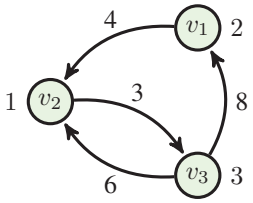


Figure 1. An example task containing three different job types.

$e_i \le e(v_i)$. Combining the job sequences of individual tasks results in a job sequence of the task set.

Figure 1 shows an example to illustrate the semantics of DRT tasks. When the system starts, T releases its first run-time job instance by an arbitrary vertex (job type). Then the released sequence corresponds to a particular direct path through $G(T)$. Consider the job sequence $\sigma = [(0,2),(4,1),(8,3),(16,2)]$ which corresponds to path $\pi = (v_1, v_2, v_3, v_1)$ in $G(T)$. Note that the second job instance in σ (v_2) is released 1 time unit later than its earliest possible release time.

The problem to solve in this paper is to calculate the worst-case *delay* (the time between the release and the finishing time) of each job type in each task over all possible job sequences of the task set. Note that different from the task model used in [15], [14], in this paper there is no deadline constraint for each job type. This means a job instant $\overline{v}$ may still be unfinished when its successor has been released. In this case, the successor cannot start execution until $\overline{v}$ is finished. Therefore, the delay of a job is caused not only by workload of higher-priority tasks, but also by its predecessor jobs in the same task.

B. Real-Time Calculus Basics

We introduce some basic concepts in RTC [6] that are relevant to the analysis in this paper. RTC uses *variability characterization curves* (*curves* for short) to describe timing properties of event streams and available resource:

Definition 1 (Arrival Curve)**.** *Let $R[s,t)$ denote the total amount of requested capacity to process in time interval $[s,t)$. Then, the corresponding (upper) arrival curve is denoted as α, and satisfy $\forall s < t, \quad R[s,t) \le \alpha(t-s)$, where $\alpha(0) = 0$.*

Definition 2 (Service Curve)**.** *Let $C[s,t)$ denote the amount of workload that a resource can process in time interval $[s,t)$. Then, the corresponding (lower) service curve is denoted as β, and satisfy $\forall s < t, \quad C[s,t) \ge \beta(t-s)$, where $\beta(0) = 0$.*

In particular, the service curve of a fully-dedicated processor is the diagonal $\beta(t) = t$.

One can compute the output curves of a processing component from its input arrival and service curves. In particular, the remaining service curve, which represents the available resource after serving the input workload, as follows:

$$\forall t \ge 0 : \beta'(t) = \sup_{0 \le \lambda \le t} \{\beta(\lambda) - \alpha(\lambda)\} \tag{1}$$

which is represented by $\beta' = \beta \ominus \alpha$ for short. Moreover, one can compute the maximal delay of each input event by

$$delay \triangleq \sup_{\lambda \ge 0} \{\inf\{\tau \in [0,\lambda] : \alpha(\lambda - \tau) \le \beta(\lambda)\}\} \tag{2}$$

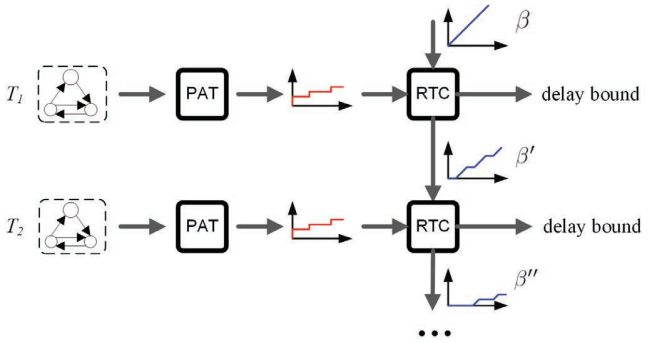


Figure 2. Illustration of the naive approach.

Intuitively, $delay$ is the maximal horizontal distance from α to β.

III. A Naive Approach

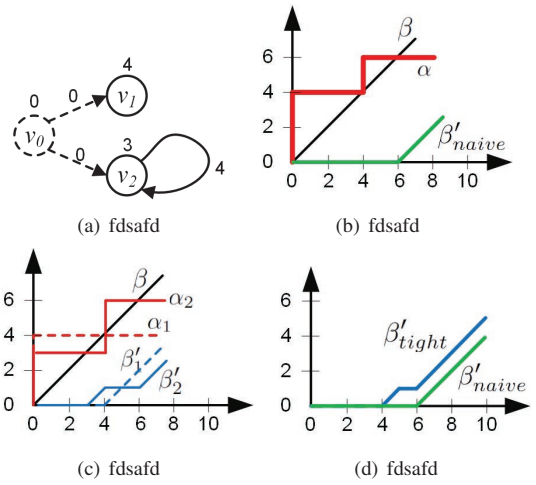
Stigge et al. [12], [11] developed the Path Abstraction Technique (PAT) to efficiently compute the so-called *request bound function* of a DRT task, which is equivalent to the arrival curve in RTC. Instead of explicitly enumerating and evaluating all the possible paths in the graph, PAT represents workload generated by a path by compact information and uses dynamic programming to extend the information from a shorter path to longer one. By PAT, the arrival curve of a DRT task T for time intervals of size up to l can be precisely computed in polynomial time with respect to l.

It is natural to come to the idea of combining PAT and RTC to solve the delay analysis problem of this paper, as shown in Figure 2. This follows the modular analysis framework in RTC, in which the analysis procedure starts with the highest-priority task T_1 and the initial service curve β that describes the available resource to the whole task system. The delay bound for T_1 is calculated by (2), and the remaining service curve β' is computed by (1). Then the next step analyzes the delay bound for T_2 with β' as the input service curve, and compute the remaining service curve β''. This procedure repeats until the lowest-priority task is analyzed.

Pessimism of the Naive Approach

Unfortunately, such a naive approach may grossly overestimate the delay bounds of individual job types. The pessimism comes from two sources. The first source is quite obvious to see: the arrival curves lose the information of different job types, and RTC can only derive a common delay bound for *all* job types in a task, which is in general larger than the delay bound of an particular job type.

The second source of pessimism comes from the fact that directly applying the standard RTC to the request bound function of a DRT task and the service curve in general results in pessimistic remaining service curves. We use the example in Figure 3 to illustrate this. The task in Figure 3-(a) has four job types. v_0 is a "dummy" node with zero execution demand, which is used to model the non-deterministic choice between taking path $\pi_1 = \{v_1\}$ and path $\pi_2 = \{v_2, v_2, \cdots\}$. Suppose the input resource is a fully-dedicated processor, i.e., $\beta(t) = t$. If



(a) fdsafd (b) fdsafd

(c) fdsafd (d) fdsafd

Figure 3. An example showing the pessimism if we compute the

we use the algorithm in [12] to compute the request bound function[1] of the task graph (denoted by α) and apply (1) to α and β, the resulting remaining service curve (denoted by β'_{naive}) is depicted in Figure 3-(b). On the other hand, we can derive the remaining service curves with the two possible paths, as shown in Figure 3-(c). We use α_1 and α_2 to represent the request bound function of path π_1 and π_2, and β'_1 and β'_2 the corresponding remaining service curves, respectively. By getting the point-wise minimum of β'_1 and β'_2 we can get the remaining service curve $\beta'_{tight} = \beta'_1 \wedge \beta'_2$, which is more precise than β'_{naive} as shown in Figure 3-(d).

In the following sections we will address the above two sources of pessimism, to more precisely bound the delay of individual job types in a task graph and compute the remaining service curves.

IV. Delay Bounds

In this section we introduce how to compute the delay bound of a particular job type of a task T. For this moment, we assume the input service curve is known for the current task T. In the next section we will introduce how to compute the service curves available to each task.

The delay of a job instance is caused by the accumulated workload released before it. So to analyze the delay of a job type, we shall look at the workload released along each path that ends with this particular job type. To capture this, we first define the *reverse request functions*:

Definition 1 (Reverse Request Function)**.** *For a path $\pi = (v_0, \cdots, v_l)$ through the graph $G(T)$ of a task T, we define its* reverse request function *as:*

$$\mathsf{rrf}_\pi(t) = \max\{e(\pi') | \pi' \text{ is a suffix of } \pi \text{ and } p(\pi') < t\}$$

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

[1]The algorithm in [12] computes the *demand bound functions* of DRT tasks. However, this algorithm can be easily adapted to the computation of request bound functions by ignoring the deadlines.

Lemma 1. *The delay of job type v is bounded by*

$$delay'(v) := \max_{\pi=\{\cdots,v\}} \left\{ \sup_{t\geq 0}\{\inf\{\lambda \in [0,t] : \mathsf{rrf}_\pi(t) \leq \beta(t+\lambda)\}\} \right\}$$

The proof of the lemma follows the similar idea with the standard delay bound in RTC [5], and is omitted due to space limit. Note that $\max_{\pi=\{\cdots,v\}}\{...\}$ represents getting the maximum over all paths in the task graph that ends with v.

Directly using Lemma 1 to compute the delay bound of v requires to enumerate all the paths that ends with v in $G(T)$, which has an exponential time complexity. In the following, we will refine Lemma 1 for efficient computation. For this, we define the *reverse request bound functions*:

Definition 2 (Reverse Request Bound Function)**.** *For a job type v, we define its* reverse request bound function *as:*

$$\mathsf{rrbf}_v(t) = \max_{\pi=\{\cdots,v\}} \{\mathsf{rrf}_\pi(t)\}$$

Theorem 1. *The response time of job type V is bounded by*

$$delay(v) := \sup_{t\geq 0}\{\inf\{\lambda \in [0,t] : \mathsf{rrbf}_v(t) \leq \beta(t+\lambda)\}\}$$

Proof: We will show $delay'(v) = delay(v)$:

$$\begin{aligned} delay'(v) &= \max_{\pi=\{\cdots,v\}} \left\{ \sup_{t\geq 0}\{\inf\{\lambda \in [0,t] : \mathsf{rrf}_\pi(t) \leq \beta(t+\lambda)\}\} \right\} \\ &= \sup_{t\geq 0} \left\{ \inf \left\{ \lambda \in [0,t] : \max_{\pi=\{\cdots,v\}} \{\mathsf{rrf}_\pi(t)\} \leq \beta(t+\lambda) \right\} \right\} \\ &= \sup_{t\geq 0} \{\inf\{\lambda \in [0,t] : \mathsf{rrbf}_v(t) \leq \beta(t+\lambda)\}\} \end{aligned}$$

which equals $delay(v)$. Proved. ■

We can use the idea of PAT to compute $\mathsf{rrbf}_v(t)$ in polynomial time with respect to t, the pseudo-code of which is shown in Figure 4. Similar to the algorithm in [12], we use $\langle u, e, r\rangle$ to record useful information of a path $\pi = \{u, \cdots, v\}$, where u is the starting job type of the path, e is the total workload released along the path and p is the length of the path (sum of the release separations of job types long the path) and . A notable feature of this algorithm that differs from the standard path abstraction techniques in [12] is that it traverses the graph "backwards": the path exploration procedure starts with the considered job type v, and extends the paths by appending their predecessors to them.

The time complexity of the algorithm in Figure 4 is polynomial with respect to t. For any task system whose total utilization is bounded by a constant $c < 1$, the size of a *busy period* is bounded by a pseudo-polynomially large number [12], before which rrbf_v and β must intersect with each other. It is also known that the maximal vertical distance between the arrival curves and service curves occurs before they intersect [8], so we only need to evaluate $\mathsf{rrbf}_\pi(t)$ till a pseudo-polynomially large number t, and the overall time complexity of the delay analysis of each job type is pseudo-polynomial. Several optimizations [12] can be applied to significantly improve the efficiency of the algorithm. We refer to [12] for details of these optimizations, as well as the intuition of the path abstraction technique. Note that $\mathsf{rrbf}_v(\delta)$ is a left-open and right-close staircase function, but the function generated

```
Ω ← {⟨v, e(v), 0⟩} // v is the job type under analysis
while ∃⟨u, e, r⟩ ∈ Ω : r < t do
  for all ⟨u, e, r⟩ ∈ Ω : r < t do
    for all edges (w, u) ∈ G(T) do
      e' ← e + e(w)
      r' ← r + p(w, u)
      if r' ≤ t then
        Ω ← Ω ∪ {⟨w, e', r'⟩}
      end if
    end for
  end for
end while
return max{ e | ⟨u, e, r⟩ ∈ Ω ∧ r ≤ t}
```

Figure 4. Algorithm for computing $\mathsf{rrbf}_v(t)$ for job type v.

by algorithm in Figure 4 is left-close and right-open, which can be easily transformed into its left-open and right-close correspondence in polynomial time.

V. Remaining Resource

Recall in the example of Section III, to get a more precise remaining service curve we should compute the remaining resource with respect to each individual path and then join the resulting remaining service curves, as formally presented in the following theorem[2]:

Theorem 2. *The remaining service is lower bounded by*

$$\beta' = \min_{\pi \in G(\tau)} \{\beta \ominus rf_\pi\} \tag{3}$$

Proof: Without loss of generality we assume the system starts at time 0. We consider an arbitrary time interval $[s,t)$, let s' be the earliest time instant no earlier than s such that at s' the resource is idle, and let t' be the earliest time instant no later than t such that at t' the resource is idle. Suppose the job instances released by the task in $[s',t')$ follow a particular path π. The released workload is at most $rf_\pi(t'-s')$. So the remaining resource in $[s',t')$ is .

$$R'_\pi(s',t') \geq R(s',t') - rf_\pi(t'-s')$$

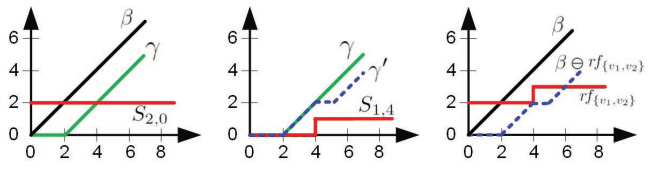
Although s' and t' are unknown variables, we know $R'_\pi[s,t)$ is larger than or equal to $R'_\pi[s',t')$ with any assignment of s' and t', so we have

$$\begin{aligned} R'_\pi(s,t) &\geq \sup_{s'\geq s, t'\leq t}\{R'_\pi(s',t')\} \geq \sup_{s'\geq s, t'\leq t}\{R(s',t') - rf_\pi(t'-s')\} \\ &\geq \sup_{s'\geq s, t'\leq t}\{\beta(t'-s') - rf_\pi(t'-s')\} \\ &= \sup_{0\leq\lambda\leq t-s}\{\beta(\lambda) - rf_\pi(\lambda)\} \end{aligned}$$

Moreover, we know $R'[s,t) \geq \min_{\pi\in G(\tau)}\{R'_\pi[s,t)\}$, so in summary, we have

$$R'[s,t) \geq \min_{\pi\in G(\tau)}\left\{\sup_{0\leq\lambda\leq t-s}\{\beta(\lambda) - rf_\pi(\lambda)\}\right\}$$

[2]For readers who are familiar with the time-interval domain representation in RTC, we would like to clarify that a service curve in RTC is a *supper-additive* function. However, $\beta \ominus rf_\pi$ for a particular path π may *not* be a *super-additive* function, since rf_π is defined in the time domain and may not be a *sub-additive* function. However, joining $\beta \ominus rf_\pi$ for all possible paths indeed yields a super-additive service curve in the time-interval domain.



(a) $\gamma = \beta \ominus S_{2,0}$ (b) $\gamma' = \gamma \ominus S_{1,4}$ (c) $\beta \ominus rf_\pi$

Figure 5. Illustration of Lemma 2.

Proved. ■

Using (3) to compute the remaining service curve needs to enumerate all the paths in $G(T)$, which is computationally intractable. In the following we present an efficient algorithm using the idea of path abstraction.

We use a *triple* $\langle v,p,\gamma\rangle$ to record useful information of a path π, where v is the ending job type of the path, p is the length of the path, and γ represents the remaining resource after serving the request released along the path (as discussed above γ may *not* be a super-additive function). For each path consisting of a single job type u, the corresponding triple is $\Upsilon = \langle u, 0, \beta \ominus e(u)\rangle$ where β is the input service curve to the analyzed task graph. Starting from a single-job path, we can extend the path by updating the corresponding triple by the $\diamond$ operator:

Definition 3 ($\diamond$ Operator)**.** *Let $\langle u,p,\gamma\rangle$ be a triple for a path ending with u, and v be a successor of u. We define the $\diamond$ operator by $\langle u,p,\gamma\rangle \diamond v = \langle v,p',\gamma'\rangle$, where $p' = p + p(u,v)$ and $\gamma' = \gamma \ominus S_{e(v),p'}$.*

Lemma 2. *Given a path $\pi = \{v_0, v_1, \cdots, v_l\}$ and β is the input service curve. Let $\{w,p,\gamma\} = \langle v_0, 0, \beta \ominus S_{e(v_0),0}\rangle \diamond v_1 \diamond \cdots \diamond v_l$, then we have $\gamma = \beta \ominus rf_\pi$.*

The lemma is proved in the appendix. For example, given a fully dedicated processor (the input service curve is $\beta(t) = t$), the triple corresponding to path $\pi = \{v_1\}$ is $\langle v_1, 0, \gamma\rangle$, where γ is shown in Figure 5-(a). To extend from π to $\pi' = \{v_1, v_2\}$, we have $\langle v_1, 0, \gamma\rangle \diamond v_2 = \langle v_2, 4, \gamma'\rangle$ where γ' is shown in Figure 5-(b). On the other hand, directly using $\beta \ominus rf_{\{v_1,v_2\}}$ we get the remaining resource as shown in Figure 5-(c), which is the same as γ'.

By Lemma 2 we know in the path exploration procedure we can update triples by the $\diamond$ operator instead of extending concrete paths. However, the path exploration procedure may still generate exponentially many triples. In the following, we show how to merge different triples in order to pseudo-polynomially bound the total number of different triples need to be maintained during the path exploration procedure.

Definition 4 ($\wedge$ Operator)**.** *For $\langle u,p,\beta_1\rangle$ and $\langle u,p,\beta_2\rangle$ with the same u and p, we define the $\wedge$ operator*

$$\langle u,p,\beta_1\rangle \wedge \langle u,p,\beta_2\rangle = \langle u,p,\beta_1 \wedge \beta_2\rangle$$

where $\forall t \geq 0 : (\beta_1 \wedge \beta_2)(t) = \min(\beta_1(t), \beta_2(t)))$.

Lemma 3. *Given two triples $\langle v_0,p,\beta_1\rangle$ and $\langle v_0,p,\beta_2\rangle$ with*

1: $\Omega \leftarrow \{\langle u, 0, \beta \ominus S_{e(u),0}\rangle \mid u \in V(T)\}$
2: **while** $\exists \langle u, p, \gamma\rangle \in \Omega : p < t$ **do**
3: **for all** $\langle u, p, \gamma\rangle \in \Omega : p < t$ **do**
4: $\Omega \leftarrow \Omega \setminus \{\langle u, p, \gamma\rangle\}$
5: **for all** edges $(u, w) \in G(T)$ **do**
6: $\langle w, p', \gamma'\rangle = \langle u, p, \gamma\rangle \diamond w$
7: **for all** $\langle w, p', \delta\rangle \in G(T)$ **do**
8: $\Omega \leftarrow \Omega \setminus \{\langle w, p', \delta\rangle\}$
9: $\Omega \leftarrow \Omega \cup \{\langle w, p', \delta\rangle \wedge \langle w, p', \gamma'\rangle\}$
10: **end for**
11: **end for**
12: **end for**
13: **end while**
14: **return** $\bigwedge_{\langle u,p,\gamma\rangle\in\Omega}\{\gamma\}$

Figure 6. Algorithm for computing the remaining service curve up to interval size t.

the same u and p, let $\pi = \{v_0, v_1, \cdots, v_l\}$. Let

$$\langle u_1, p_1, \beta_1'\rangle = \langle u, p, \beta_1\rangle \diamond v_1 \diamond \cdots \diamond v_l$$
$$\langle u_2, p_2, \beta_2'\rangle = \langle u, p, \beta_2\rangle \diamond v_1 \diamond \cdots \diamond v_l$$
$$\langle u_3, p_3, \beta_3'\rangle = \langle u, p, \beta_1 \wedge \beta_2\rangle \diamond v_1 \diamond \cdots \diamond v_l$$

then it holds (i) $u_1 = u_2 = u_3$, (ii) $p_1 = p_2 = p_3$, and (iii) $\beta_3' \le \beta_1' \wedge \beta_2'$.

Proof: By the definition of $\diamond$ operator it is easy to see $u_1 = u_2 = u_3$ and $p_1 = p_2 = p_3$. In the following we prove $\beta_3' \le \beta_1' \wedge \beta_2'$. To save space, we only prove the special case of $\pi = \{v_0, v_1\}$. The general case of $\pi = \{v_0, v_1, \cdots, v_l\}$ can be proved by induction using similar reasoning.

For simplicity, let $\alpha = S_{e(v_1),p+p(v_0,v_1)}$ then we can write $\beta_1' = \beta_1 \ominus \alpha$, $\beta_2' = \beta_1 \ominus \alpha$ and $\beta_3' = (\beta_1 \wedge \beta_2) \ominus \alpha$. We have

$$\begin{aligned}\beta_3'(t) &= \sup_{0\le\lambda\le t}\{\min(\beta_1(\lambda), \beta_2(\lambda)) - \alpha(\lambda)\}\\ &\le \min(\sup_{0\le\lambda\le t}\{\beta_1(\lambda) - \alpha(\lambda)\}, \sup_{0\le\lambda\le t}\{\beta_2(\lambda) - \alpha(\lambda)\})\\ &= \min(\beta_1'(t), \beta_2'(t))\end{aligned}$$

Proved. ■

The pseudo-code of the overall algorithm to compute the remaining service curve up to interval size t is shown in Figure 6. The algorithm first collects the triples $\langle u, 0, \beta \ominus S_{e(u),0}\rangle$ for each single-job path $\{u\}$ in Ω, where $\beta \ominus S_{e(u),0}$ represents the remaining resource after serving the a job instance of u. Then the algorithm iteratively extends each triple in Ω with the successor of its ending job type using the $\diamond$ operator (line 6), and merges triples $\langle u, p, \gamma\rangle$ with the same u and p using the $\wedge$ operator (line 8 and 9). The procedure continues until all the triples $\langle u, p, \gamma\rangle$ in Ω have path length $p \ge t$. Finally the point-wise minimum of γ of all triples $\langle u, p, \gamma\rangle$ in Ω is the desired remaining service curve (up to interval size t).

The time complexity of the algorithm in Figure 6 is polynomial with respect to t. Similar to the algorithm in 4 in last section, we only need to generate the remaining service curve up to a pseudo-polynomially large number t, so the overall time complexity of computing the remaining service curve is pseudo-polynomial.

Note that using the $\wedge$ operator to merge multiple triples during the path exploration procedure in general may lose precision (in Lemma 3 we have $\beta_3' \le \beta_1' \wedge \beta_2'$ rather than $\beta_3' = \beta_1' \wedge \beta_2'$). However, in practice the analysis precision with merging triples is almost as good as without merging triples (see details in Section VI).

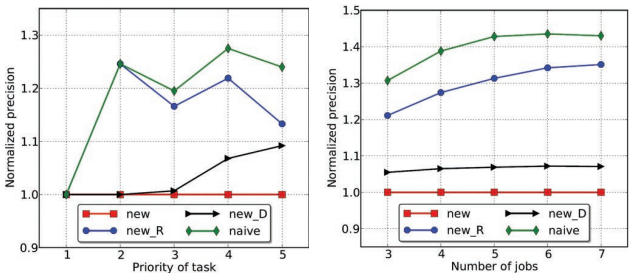
VI. Experimental Evaluations

We evaluate the delay analysis precision of our proposed approach, denoted by "**new**" in Figure 7, which uses the delay analysis algorithm in Section IV plus the remaining resource computation algorithm in Section V. We compare "**new**" with the naive approach directly combining PAT and RTC in Section III, denoted by "**naive**". Furthermore, we add two intermediate approach between "**new**" and "**naive**" into the comparison: "**new_D**" uses the new delay analysis algorithm in Section IV and the naive remaining resource computation in Section III and "**new_R**" uses the new remaining resource computation algorithm in Section V and the naive delay bound in Section III, to evaluate the contribution to analysis precision by each part of our new approach.

2000 task sets are randomly generated in the experiments of Figure 7-(a). Each task set has 5 tasks and each task contains 5 job types. The output degree of vertex is randomly chosen in the range $[1, 3]$. Edges are randomly placed but it is guaranteed that the whole graph is strongly connected. The WCET of each job type is randomly chosen in $[1, 4]$ and the inter-release separation between two job types is randomly chosen in $[10, 15]$. Figure 7-(a) shows the normalized analysis precision of the different approaches against the delay bound obtained by **new**. A point in the figure with x-axis of n denotes the average *normalized analysis precision* of all jobs in all n^{th}-highest priority task of all the generated task sets by the corresponding analysis approach. For example, if the delay bounds derived by "**new**" are $\{10, 16, 16, 20, 20\}$ for the second highest priority task, and the delay bounds obtained by "**new_R**" are $\{20, 24, 32, 30, 40\}$, then the average *normalized analysis precision* of the second highest priority task in this task set is $(\frac{20}{10} + \frac{24}{16} + \frac{32}{16} + \frac{30}{20} + \frac{40}{20})/5 = 1.8$. The average *normalized analysis precision* of the overall experiment depicted in the figure is the average of that of all generated task sets.

From Figure 7-(a) we can see that "**new**" is typically 20% to 30% more precise than "**naive**". The precision improvement by our new remaining service curve computation algorithm is larger with lower-priority tasks. This is because in "**naive**" the pessimism of remaining service curve computation accumulates along the analysis flow from high priority to low priority. We also observe that the precision improvement by our new delay analysis algorithm drops down for the lowest-priority tasks. This can be explained as followed. The pessimism of the delay bound in "**naive**" comes from the difference of delays of different job types. In the lowest-priority tasks the delays are usually very large, so the relative difference ratio among them may becomes smaller. For the highest-priority task, the delay of each job type simply equals its WCET, so the results returned by all the methods are regarded as the same.

In the experiments of Figure 7-(b), we also randomly generate 2000 task sets. Each task set has 3 tasks and the



(a) (b)

Figure 7. Experiment results

number of job types contained in each task is different (with different x-axis values in the figure). The WCET of each job type is randomly chosen in $[1, 6]$ and the inter-release separation between two job types is randomly chosen in $[10, 15]$. From Figure 7-(b) we can see that the precision improvement becomes more significant as the size of task graph scales up. This is because the a larger task typically has more diverse job types, which results in larger pessimism due to the delay bounds in "**naive**" that are not able to distinguish different job types.

VII. Conclusions and Future Work

In this paper we study the delay analysis problem for DRT task systems without the constrained-deadline limitation. We first show that directly combining path abstraction technique (PAT) in real-time scheduling theory and real-time calculus (RTC) can provide safe delay bounds, but the results are typically over-pessimistic. Then we propose new algorithms to efficiently and precisely solve the delay analysis problem. In the future, we plan to analyze the *backlog* of the DRT task model, and study the representation and analysis of the output workload maintaining the structural information.

Acknowledgement

This work is partially supported by NSF of China (Grant No. of 61300022, 61370076) and the Fundamental Research Funds for the Central Universities of China (Grant No. N130504008).

References

[1] S. K. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium (RTSS)*, 1990.

[2] Sanjoy Baruah. Dynamic-and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[3] Sanjoy Baruah. The non-cyclic recurring real-time task model. In *Proceedings of the IEEE 31st Real-Time Systems Symposium (RTSS)*, pages 173–182, Nov 2010.

[4] Sanjoy K. Baruah, Deji Chen, Sergey Gorinsky, and Aloysius K. Mok:. Generalized multiframe tasks. *Real-Time Systems*, 1999.

[5] J. L. Boudec and P. Thiran. Network calculus - a theory of deterministic queuing systems for the internet. In *LNCS 2050*. Springer Verlag, 2001.

[6] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.

[7] R. I. Davis, T. Feld, V. Pollex, and F. Slomka. Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In *in Proc. of RTAS*, 2014.

[8] Nan Guan and Wang Yi. Finitary real-time calculus: Efficient performance analysis of distributed embedded systems. In *Procedings of the 34th IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[9] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM*, 1973.

[10] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *IEEE Transactions on Software Engineering*, 1997.

[11] Martin Stigge. Real-time workload models: Expressiveness vs. analysis efficiency. In *Ph.D. Dissertation, Uppsala University*, 2014.

[12] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 71–80. IEEE, 2011.

[13] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. On the tractability of digraph-based task models. *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[14] Martin Stigge, Nan Guan, and Wang Yi. Refinement-based exact response-time analysis. *27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[15] Martin Stigge and Wang Yi. Combinatorial abstraction refinement for feasibility analysis. In *Procedings of the 34th IEEE Real-Time Systems Symposium (RTSS)*, pages 340–349, 2013.

[16] Ernesto Wandeler. Modular performance analysis and interface-based design for embedded real-time systems. In *PhD thesis, ETHZ*, 2006.

Appendix: Proof of Lemma 2

Proof: Let $\alpha_0 = S_{e(v_0),0}$ and $\alpha_x = S_{e(v_x),\sum_{i=0}^{x-1} p(v_i,v_{i+1})}$ for $x \geq 1$, then we can write $\gamma = \beta \ominus \alpha_0 \ominus \alpha_1 \ominus \cdots \ominus \alpha_l$ and $rf_\pi = \alpha_0 + \alpha_1 + \cdots + \alpha_l$. In the following we will prove

$$\beta \ominus \alpha_0 \ominus \alpha_1 \ominus \cdots \ominus \alpha_l = \beta \ominus (\alpha_0 + \alpha_1 + \cdots + \alpha_l) \tag{4}$$

To save space, we only prove it for the special case of $\pi = \{v_0, v_1\}$, i.e.,

$$\beta \ominus \alpha_0 \ominus \alpha_1 = \beta \ominus (\alpha_0 + \alpha_1) \tag{5}$$

and the general case can be proved by induction using similar reasoning. Let $\beta' = \beta \ominus (\alpha_0 + \alpha_1)$ and $\beta'' = \beta \ominus \alpha_0 \ominus \alpha_1$. Let $P = p(v_0, v_1)$, then $\alpha_1(t) = 0$ for $t \leq P$ and $\alpha_1(t) = e(v_1)$ for $t > P$. We first prove (5) for $0 \leq t \leq P$:

$$\begin{aligned}\beta''(t) &= \sup_{0\leq\theta\leq t}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-\alpha_1(\theta)\}\\ &= \sup_{0\leq\theta\leq t}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}\} = \sup_{0\leq\lambda\leq t}\{\beta(\lambda)-\alpha_0(\lambda)\}\end{aligned}$$

On the other hand, we have

$$\beta'(t) = \sup_{0\leq\lambda\leq t}\{\beta(\lambda)-\alpha_0(\lambda)-\alpha_1(\lambda)\} = \sup_{0\leq\lambda\leq t}\{\beta(\lambda)-\alpha_0(\lambda)\}$$

so $\beta'(t) = \beta''(t)$ holds for $0 \leq t \leq P$. Then we prove for the case of $t \geq p$.

$$\begin{aligned}\beta''(t) &= \sup_{0\leq\theta\leq t}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-\alpha_1(\theta)\}\\ &= \max(\sup_{0\leq\theta\leq P}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-\alpha_1(\theta)\},\\ &\qquad \sup_{P\leq\theta\leq t}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-\alpha_1(\theta)\})\end{aligned}$$

For simplicity, we define $\beta''(t) = \max(x_1, x_2)$, where

$$\begin{aligned}x_1 &= \sup_{0\leq\theta\leq P}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-\alpha_1(\theta)\}\\ &= \sup_{0\leq\theta\leq P}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-0\} = \sup_{0\leq\lambda\leq P}\{\beta(\lambda)-\alpha_0(\lambda)\}\\ x_2 &= \sup_{P\leq\theta\leq t}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-\alpha_1(\theta)\}\\ &= \sup_{P\leq\theta\leq t}\{\sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)\}-e(v_1)\}\\ &= \sup_{0\leq\lambda\leq\theta}\{\beta(\lambda)-\alpha_0(\lambda)-e(v_1)\}\end{aligned}$$

On the other hand, we have

$$\begin{aligned}\beta'(t) &= \sup_{0\leq\lambda\leq t}\{\beta(\lambda)-\alpha_0(\lambda)-\alpha_1(\lambda)\}\\ &= \max(\sup_{0\leq\lambda\leq P}\{\beta(\lambda)-\alpha_0(\lambda)-\alpha_1(\lambda)\}, \sup_{P\leq\lambda\leq t}\{\beta(\lambda)-\alpha_0(\lambda)-\alpha_1(\lambda)\})\\ &= \max(\sup_{0\leq\lambda\leq P}\{\beta(\lambda)-\alpha_0(\lambda)\}, \sup_{P\leq\lambda\leq t}\{\beta(\lambda)-\alpha_0(\lambda)-e(v_1)\})\end{aligned}$$

so $\beta'(t) = \beta''(t)$ for $t \geq P$. Proved. ■