Abstract—Demand bound function (DBF) is a powerful abstraction to analyze feasibility/schedulability of real-time tasks. Computing DBF for expressible system models, such as graph-based tasks is typically very expensive. In this paper, we develop new techniques to drastically improve the DBF computation efficiency for a representative graph-based task model, digraph real-time tasks (DRT). First, we apply the well-known quick processor-demand analysis (QPA) techniques, which was originally designed for simple sporadic tasks, to the analysis of DRT. The challenge is that existing analysis techniques of DRT have to compute the demand for each possible interval size, which is contradictory to the idea of QPA that aims to aggressively skip the computation for most interval sizes. To solve this problem, we develop a novel ILP-based analysis technique for DRT, to which we can apply QPA to significantly improve the analysis efficiency. Second, we improve the task utilization computation (a major step in DBF computation for DRT) efficiency from pseudo-polynomial complexity to polynomial complexity. Experiments show that our approach can improve the analysis efficiency by dozens of times.

Index Terms—DRT, Feasibility analysis, Processor demand criteria analysis, Acceleration, Linear programming.

I. INTRODUCTION

Real-time systems are often implemented by a number of concurrent tasks sharing hardware resources, in particular the execution processors. Traditionally, a real-time task system is modeled as a collection of periodically or sporadically repeating computation requests [1], [2]. With the increased complexity of real-time embedded software, complex control flow structures such as mode switches, local loops, and if-else branches cannot be fully captured by these simple periodic and sporadic task model. The first graph-based real-time task model is the recurring branching task model which is proposed to formulate some restricted forms of conditional real-time process code [3]. Over years, more and more expressive graph-based task models are proposed to precisely describe complex embedded real-time systems [3]–[10], and the digraph real-time (DRT) task model [10] is an representative one among them, which generalizes most known real-time task models.

When scheduling a real-time task system on a target execution platform, we concern with a fundamental analysis problem, called the feasibility analysis, i.e., “how do we determine whether the tasks can be scheduled in such a manner that all jobs complete by their deadlines?”. In this paper, we restrict our attention to preemptive scheduling (i.e., a job executing on the processor can be interrupted at any instant in time, and its execution resumed later with no cost or penalty) of the DRT tasks. The demand bound function analysis technique is a standard methodology for the feasibility-analysis of real-time tasks under preemptive scheduling, which is centered upon the idea of a demand bound function (DBF) for tasks: this quantifies the maximum amount of processor time that all the jobs generated by the task can require in an interval of specified size, and attempt to determine whether there is an interval size for which the demand bound summed over all tasks in the system exceeds the processor capacity (i.e., the overflow occurs). Briefly, the computation of DBF for DRT tasks are conducted in the following two steps:

Step 1. Compute the utilization (e.g., the amount of processor demand required by a task per time unit) for each task, and determine an interval-size bound \( l_{\text{mt}} \) by using the utilization summed over all tasks.

Step 2. For each interval size less than \( l_{\text{mt}} \), compute the DBF for each task, and then sum the DBF over all tasks, and check whether the summed DBF (i.e., the maximum processor demand required by all tasks) exceeds this interval size (i.e., the processor capacity).

The DBF methodology is originally proposed for analyzing the simplest sporadic tasks (e.g., [2]), and further is adapted to analyze the DRT task model [10]. When analyzing the simplest sporadic tasks, the DBF method is quite efficient, i.e., the bound \( l_{\text{mt}} \) and the DBF for any specified interval size can be computed within a constant time. The only problem left is that the bound \( l_{\text{mt}} \) has a pseudo-polynomial scale, and by following the traditional DBF methodology, it is inefficient to check all the interval sizes (less than \( l_{\text{mt}} \)). To this end, Zhang et al. [11] propose an efficient technique, called the quick processor-demand analysis (QPA), to accelerate the traditional DBF analysis for the simplest sporadic tasks. The main benefit of QPA technique is that QPA does not require to do analysis for every interval size, but only need to check few interval sizes for which the overflow may occur. With the help of QPA, the accelerated DBF analysis for simplest sporadic tasks even almost keeps a constant complexity [11].

However, when the DBF method is applied to analyze the DRT task model, it suffers a sharply increased complexity, and it is challenging to directly adapt QPA (for simple sporadic tasks) to accelerate the DBF analysis of DRT tasks, due to the fact that the parameters (e.g., task utilization and DBF) that are frequently calculated during the analysis progress are more complicated to compute, and more specifically:

- **DRT task’s utilization is more complicated to compute.** In the simplest sporadic task model, the utilization of a task can be computed within \( O(1) \) time, but in the DRT task model, computing the task utilization is equivalent to solving the densest cycle of the task graph. The traditional method in [10] solves this problem in pseudo-polynomial time.
- **DRT task’s DBF is more complicated to compute, and the traditional method in [10] cannot fit QPA.** The DBF for a simple sporadic task can be represented as a...
formula, which is computed within $O(1)$ time. However, computing the demand bound function for a DRT task is equivalent to searching an optimal path on the task graph with the maximum workload and with the bounded length. The traditional method in [10] use a dynamic programming (DP) based approach to deal with the exponential path explosion during the graph search progress, which relies on sequentially checking all interval sizes less than lmt and thus is contradictory with the idea of QPA.

**Contributions.** In this paper, we aim to solve the above challenging problems, and as the main contribution of this paper, we propose efficient acceleration techniques for the DBF analysis of the DRT task model. More specifically, we accelerate the DBF analysis in two parts.

- We integrate the QPA framework into the analysis of DRT tasks. Instead the traditional DP method, we propose an integer linear programming (ILP) model to directly solve the demand bound function, which does not need to sequentially check all the interval sizes, and thus this makes it possible to apply the QPA framework for analyzing DRT tasks. To make our ILP model more efficient to solve, we divide the constraints of our ILP model into two parts: the easy constraints and the complex constraints. We develop a row generation algorithm to iteratively solve our ILP model which initially only involve the easy constraints, and dynamically adds part of the complex constraints when necessary.

- We find that the task utilization computation that is solved by traditional method in [10] within pseudo-polynomial time actually has a polynomial time complexity. We develop a linear programming approach to solve the densest cycle of a task graph, and thus derive the task utilization within polynomial time, which clearly outperforms the traditional method and further accelerates the analysis progress.

We conduct experiments with randomly generated task systems to evaluate the efficiency of our proposed method. The experimental results show that our method is much faster than the traditional method in [10].

The motivation for accelerating the DBF analysis is twofold. The first requirement comes from online systems. During the runtime of a real-time system, new tasks may arrive, and need to be added to the current task set. The system must reanalyze feasibility online to decide whether to allow the new tasks to enter into the system or not. Such online admission control gives a stronger requirement on the efficiency of the feasibility test as the decisions have to be made in a very short time. Second, efficient analysis is very useful in design space exploration of real-time system, in which many different parameter profiles need to be checked. An automated search may even be undertaken as part of the architectural definition of the system. An efficient but accurate feasibility scheme is therefore needed.

**Organization.** The remainder of this paper is organized as follows. Sec. II introduces the formal digraph-based real-time task model and the execution semantics of the DRT task model. Sec. III reviews the DBF methodology for the feasibility analysis of the DRT task model and the QPA acceleration framework. Sec. IV integrates the QPA framework into the DBF analysis of the DRT task model, and presents the acceleration techniques especially designed for the DRT task model. Sec. V shows the simulation experiments that we have conducted to evaluate our method. Sec. VI reviews the related work about real-time task graph models. Finally, Sec. VII concludes the paper.

II. TASK MODEL

We consider a task system $T$ consisting of a set of $n$ independent DRT tasks, i.e., $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$. A DRT task $\tau$ is represented by a directed graph $G(\tau) = (V(\tau), E(\tau))$, where $V(\tau)$ denotes the set of vertices in $G(\tau)$, and $E(\tau)$ denotes the set of edges in $G(\tau)$. The vertices in $V(\tau)$ represent the types of all jobs that can be released by $\tau$. Each vertex $v$ of $V(\tau)$ has the worst-case execution time (WCET) $e(v) \in \mathbb{N}$, and the relative deadline $d(v) \in \mathbb{N}$. The edges in $E(\tau)$ represent the order in which $\tau$’s job instances are released.

Fig. 1 shows an example to illustrate the semantics of a DRT task system $T$ is defined as the set of job sequences it may generate. In the following, we introduce the job sequences generated by $T$ in more details. Before going into details, we first give some useful notations as follows. A job instance of type $v$ is represented by a tuple $(r, e, d)$ consisting of an absolute release time $r$, an execution time $e$ and an absolute deadline $d$. The job sequence $\sigma = [(r_0, e_0, d_0), (r_1, e_1, d_1), \cdots]$ is a sequence of job instances generated by $\tau$, which corresponds to a path $\pi = (v_0, v_1, \cdots)$ in $G(\tau)$ such that the $k$-th tuple $(r_k, e_k, d_k)$ of $\sigma$ corresponds to the $k$-th vertex $v_k$ of $\pi$, and thus, the following equalities hold, i.e., $r_{k+1} - r_k \geq p(v_k, v_{k+1})$, $d_k = r_k + d(v_k)$ and $e_k \leq e(v_k)$, for all $k \geq 0$. Combining the job sequences $\sigma$ of individual tasks $\tau \in T$ results in a job sequence $\Sigma_T$ of the whole task system $T$, i.e., $\Sigma_T = \{\sigma|\tau \in T\}$.

The semantics of a DRT task system $G(\tau)$, i.e., the task $\tau$ releases its first job instance of any job type. The released job sequence corresponds to a path through $G(\tau)$. We consider the job sequence $\sigma = [(5, 2, 14), (23, 5, 31), (37, 4, 46)]$ which corresponds to path $\pi = (v_5, v_1, v_2)$ in $G(\tau)$ (the edges in $\pi$ are marked in red). Note that not
every job instance in \( \sigma \) is released as earliest as possible, e.g.,
the second job instance (associated with \( v_1 \)) is released 2
units later than its earliest possible release time.

### III. Feasibility Analysis

The main focus of this work on the DRT task model is to
solve the associated feasibility problem defined as follows.

**Definition 1** (Feasibility). A task set \( T \) is preemptive uniprocessor feasible, if and only if all job sequences \( \sum T \) generated by \( T \) can be executed on a preemptive uniprocessor platform such that all jobs meet their deadlines.

In particular, we say a job \((r, e, d)\) of type \( v \) is successfully scheduled to meet its deadline, if there is an accumulated duration of \( e \) time units where the job \( v \) executes exclusively on the processor within the time interval \([r, d] \). It is known that earliest deadline first (EDF) is an optimal algorithm for scheduling real-time tasks on a preemptive uniprocessor. Thus, the feasibility problem is equivalent to EDF schedulability.

In the following, we first introduce a general feasibility-analysis methodology. After that, we give a brief introduction to the relevant prior work about such a methodology.

**A. Demand Bound Function Methodology**

As stated in Sec. I, the demand bound function technique is a standard methodology for the feasibility-analysis of real-time tasks. For DRT tasks, the notion of demand bound function (DBF) expresses accumulated execution time that a task set can demand from the processor within any time interval of a specified size. Formally:

**Definition 2** (Demand Bound Function). For any task \( \tau_k \in T \) and an interval size \( t \), \( DBF_k(t) \) denotes the maximum cumulative execution requirement of tasks with both release time and deadline within the specified interval size \( t \), over all task sequences generated by \( \tau_k \). More precisely,

\[
DBF_k(t) = \max\{e(\pi) | \pi \in G(\tau_k) \land l(\pi) \leq t\} \tag{1}
\]

where for any path \( \pi = (v_1, v_2, \cdots, v_h) \) of \( G(\tau_k) \) with \( v_i \) is the \( i \)-th vertex visited along the path \( \pi \), \( l(\pi) \) is the length of path \( \pi \), i.e., \( l(\pi) = \sum_{i=1}^{h-1} p_{v_i,v_{i+1}} + d_{v_i} \), and \( e(\pi) \) is the workload of path \( \pi \), i.e., \( e(\pi) = \sum_{i=1}^{h} e_{v_i} \). Further, for a task system \( T \) and a specified interval size \( t \), the demand bound function \( DBF(t) \) of \( T \) is defined as

\[
DBF(t) = \sum_{k: \tau_k \in T} DBF_k(t) \tag{2}
\]

Note that the definition of \( DBF(t) \) as the sum of \( DBF_k(t) \) of all tasks \( \tau_k \) relies on their independence of each other. The definition of DBF is tight since for each interval size \( t \), there is a job sequence \( \sum T \) generated by task set \( T \) in which some jobs actually require a processor demand of \( DBF(t) \) within an interval of \( t \) time units. Although we assume the dense time, i.e., demand bound function \( DBF(t) \) is defined for all \( t \in \mathbb{R}_{\geq 0} \), changes clearly only occur at integers.

**Example 1.** Consider again job sequence \( \sigma = [(5, 2, 14), (23, 5, 31), (37, 4, 46)] \) which corresponds to

path \( \pi = (v_5, v_1, v_2) \), generated by task \( \tau_k \) from Fig. 1. This sequence \( \sigma \) shows that in a time interval \( t = 41 \), task \( \tau_k \) may generate a demand of 11 on the processor as follows: The first job is released at \( t_1 = 5 \) and the third job has its deadline at \( d_3 = 46 \). Thus, all three jobs of the sequence have both their release time and deadline within time interval \([5,46]\) of length 41. Together, their execution time is \( 2 + 5 + 4 = 11 \). In fact, there are no job sequences generated by \( \tau_k \) with a higher demand within an interval of length 41. We can conclude that \( DBF_k(41) = 11 \) for our example task \( \tau_k \). The DBF calculated for \( \tau_k \) in Fig. 1 is given in Fig. 2.
(See in Sec. III-B.§1). The point-to-point checking procedure usually is time-consuming, and there is a lot of space for optimization and improvement. For the simplest sporadic task model, Zhang et.al [11] design a very efficient framework to accelerate the traditional DBF analysis, and we will review Zhang’s framework in Sec. III-B, and discuss how to adapt their technique to accelerate the analysis of DRT tasks.

Fig. 3. Illustration for point-to-point checking procedure.

Alg. 1 only gives the basic analysis framework, but leave two main computation problems as follows.

1) How do we compute the interval-size bound \( l_{\text{mt}} \)?
2) How do we compute DBF\((t)\) for a specified interval size \( t \)?

Stigge et.al [10] propose a dynamical programming based approach to solve the above problems, and we will briefly introduce their work in Sec. III-B, exploring why Zhang’s acceleration technique (for the simplest sporadic task model) is difficult to be integrated into Stigge’s method for DRT tasks.

B. Relevant Prior Research

we now review some prior work that studies how the processor demand criteria methodology in Alg. 1 is applied to analyze the DRT task model, and which studies the possible techniques to accelerate the methodology in Alg. 1. Some notations proposed in the prior work are also used in the methods we derive in this paper.

§1. Stigge’s Method for DRT Tasks

Stigge et.al [10] apply the DBF methodology in Alg. 1 to analyze the DRT task model. Their main contribution is that they propose a dynamic programming based approach to solve the DBF computation problem, and derive a pseudo-polynomial-scale bound \( l_{\text{mt}} \). We will introduce their work in these two aspects.

DBF computation. For a task \( \tau_k \) and a specified interval size \( t \), computing the demand bound function DBF\(_k\)(\(t\)) is equivalent to find an optimal path \( \pi^* \) such that \( \pi^* \) has the maximum workload and the length \( l(\pi^*) \) is no more than \( t \), according to (1). Stigge et.al [10] develop a dynamic programming algorithm to compute DBF\(_k\)(\(t\)) by solving the optimal path problem on task graph \( G(\tau_k) \). The main idea of Stigge’s method is to use the notion of demand triples as an abstraction of concrete paths through the task graph, preventing exponential path explosion. More specifically, for any vertex \( v \), the demand triple of vertex \( v \) is denoted as \((e, d, v)\) representing a finite path \( \pi \) that ends at \( v \) and has the workload equal \( e \), i.e., \( e(\pi) = e \), and the length of \( \pi \) is no more than \( d \), i.e., \( l(\pi) \leq d \). Using all demand triples \((e, d, v)\), we can calculate DBF\(_k\)(\(t\)) as follows.

\[
\text{DBF}_k(t) = \max \{e | (e, d, v) \text{ demand triple with } d \leq t \} \tag{3}
\]

Now the remain problem is how to compute all demand triples. In [10], the demand triple computation is applied as an iterative procedure as follows.

- **Base case.** Initially, the demand triples corresponding to all zero-length paths are computed and stored.
- **Iterative case.** For any stored demand triple \((e, d, u)\), consider all successor vertices \( v \) of \( u \). For each such \( v \), one can use the demand triple \((e, d, u)\) to compute a new demand triple \((e', d', v)\) corresponding to a path that has been extended by \( v \) with a heavier workload \( e' \) and a longer length \( d' \). Each newly computed demand triple \((e', d', v)\) is stored if it is not stored yet and \( d' \leq t \).

The above step is repeated until there are no new demand triples. Note that under the iterative procedure, the demand triples with shorter length are used to compute the demand triple with longer length. Moreover, during the runtime of the DBF analysis of Alg. 1, for any interval sizes \( t \) and \( t' \) (with \( t < t' \)), the demand triples used to derive DBF\(_k\)(\(t'\)) must rely on the demand triples used to derive DBF\(_k\)(\(t\)). This restricts the DBF analysis to be a forward checking procedure.

Bound computation. We now introduce the pseudo-polynomial interval-size bound \( l_{\text{mt}} \) proposed by Stigge et.al [10]. Before going into details, we first introduce some useful notations.

**Definition 3** (density). For any cycle \( \pi = (v_1, \ldots, v_h, v_1) \), its density \( u(\pi) \) is calculated as

\[
u(\pi) = \frac{\sum_{i=1}^{h} e_i}{\sum_{i=1}^{h} P_i v_i + 1} \tag{4}\]

For example, in Fig. 1, the density of cycle \((v_1, v_2, v_3, v_1)\) is \( \frac{10}{17} \), and the density of cycle \((v_2, v_3, v_4, v_2)\) is \( \frac{7}{39} \).

**Definition 4** (utilization). For any DRT task \( \tau_k \), its utilization \( u_k \) is calculated as follows.

\[
u_k = \max \{u(\pi) | \pi \text{ is a cycle in } G(\tau_k) \} \tag{5}\]

where \( u(\pi) \) is the density of the cycle \( \pi \). Moreover, we say the cycle in \( G(\tau_k) \) that has the density \( u_k \) is the densest cycle.

For example, in Fig. 1, the densest cycle is \((v_1, v_2, v_3, v_1)\), and thus, the utilization is \( \frac{10}{17} \).

The following lemma (proposed in [10]) give the upper bound of the interval size \( t_f \) that violates DBF\(_f\)(\(t_f\)) \( \leq t_f \).

**Theorem 2.** [10] If task system \( T \) is not EDF-schedulability upon preemptive uniprocessor, there is a \( t_f \) with DBF\(_f\)(\(t_f\)) \( > t_f \) such that:

\[
t_f < l_{\text{mt}} = \frac{\sum_{k=1}^{N} \omega_k}{1 - \sum_{k=1}^{N} u_k} \tag{6}\]

where \( \omega_k \) is the total workload of \( \tau_k \), i.e., \( \omega_k = \sum_{i \in G(\tau_k)} e_i \), and \( u_k \) is the utilization of \( \tau_k \).

From Thm. 2, the interval-size bound \( l_{\text{mt}} \) used in Alg. 1 is derive by (6), and the key problem for solving \( l_{\text{mt}} \) is to calculate the task utilization \( u_k \) for each task \( \tau_k \). Stigge et.al [10] point out that the task utilization \( u_k \) can be solved by enumerating all simple cycles of the task graph \( G(\tau_k) \). Since explicit cycle enumeration is still an exponential procedure, Stigge et.al [10] reuse their path abstraction framework which
was already used to reduce the complexity of the DBF computation, and they state that their method clearly has a pseudo-polynomial time complexity, e.g., $O(C^2n)$, where $n$ is the number of vertices in $G(\tau_k)$, and $C = \sum_{(u,v)\in E} P_{uv}$ is the summation of the periods of all edges in $G(\tau_k)$ (Please refer to Page 346 of [10]).

\section{Quick Processor-demand Algorithm}

As shown in Alg. 1, the traditional DBF methodology is usually implemented as a point-to-point checking procedure, such that all interval sizes in $[1, lmt]$ should be checked one by one. Note that the interval-size bound $lmt$ usually is a very large number. Zhang et.al [11] design an efficient framework, called quick processor-demand analysis (QPA), for accelerating the DBF analysis. Zhang’s work only available for the simplest sporadic task system consisting of $n$ tasks (where each sporadic task $\tau_k$ has a worst-case execution time $C_k$, a relative deadline $D_k$ and a period $T_k$. The utilization $u_k$ of $\tau_k$ equals $\frac{C_k}{T_k}$, and we let $U = \sum_{k=1}^{n} u_k$), and their result is summarized by the following theorem.

**Theorem 3.** [11] A task set is EDF-schedulable if and only if $U \leq 1$, and the result of the following iterative algorithm is $h(t) \leq d_{min}$, where $d_{min} = \min\{D_i\}$.

Algorithm 2: Quick processor-demand analysis.

1. $t := \max\{d_k | d_k < lmt\}$
2. while $h(t) \leq t \land h(t) > d_{min}$ do
   3. if $h(t) < t$ then
      4. $t := h(t)$
   5. else
      6. $t := \max\{d_k | d_k < t\}$
   7. if $h(t) \leq d_{min}$ then
      8. the task set is schedulable
   9. else
      10. the task set is not schedulable

Here $d_k$ is the absolute deadline of a job of $\tau_k$, $h(t)$ is the processor demand function (or equally, the demand bound function) of $\tau_k$ that is calculated as follows.

$$h(t) = \sum_{k=1}^{n} \max\{0, 1 + \left\lfloor \frac{t - D_k}{T_k} \right\rfloor \} C_k$$ \hspace{1cm} (7)

and the interval-size bound $lmt$ is calculated as

$$lmt = \max\{D_1, \ldots, D_n, \sum_{k=1}^{n} (T_k - D_k) u_k \} \frac{1}{1 - U}$$ \hspace{1cm} (8)

Different from the traditional analysis framework of Alg. 1, QPA applies the DBF analysis by using a backward checking procedure as shown in Lines 1 to 6 of Alg. 2. Most importantly, Alg. 2 does not check all the interval sizes less than $lmt$, but only check few interval sizes when necessary (Lines 4 and 6). This is the key to sharply reducing the computation complexity. Moreover, the parameters (e.g., processor demand function $h(t)$ and the interval-size bound $lmt$) can be computed by the formulas (7) and (8), which has a very low complexity (e.g., polynomial time). By applying QPA technique, the DBF analysis (for the simplest sporadic task model) even keeps a constant complexity in practice.

**Difficulties to adapting QPA to DRT tasks.** In this paper, we aim to integrate the QPA framework into the DBF analysis for the DRT task model. However, the traditional method for DRT tasks (proposed in [10]) does not fit the QPA framework well. The reason is that QPA uses a backward checking procedure, and aims to prune a lot of interval sizes for which overflow will not occur. However, the traditional method in [10] applies a forward checking procedure, i.e., they use a dynamic programming based algorithm to compute the demand bound functions, which relies on sequentially checking all interval sizes less than $lmt$, and does not allow interval-size pruning. This clearly is contradictory with the idea of QPA. In order to use QPA for accelerating the analysis of DRT tasks, we will propose some new techniques to solve the demand bound function (instead the DP technique), which could fit the QPA framework better (See in Sec. IV).

**IV. ACCELERATION TECHNIQUES**

In this section, we propose some efficient techniques to accelerating the processor demand criteria analysis of the DRT task model. The basic analysis framework is borrowed from the QPA method, which applies a backward checking procedure, and only checks the interval sizes for which the overflow may occur, as shown in Alg. 3.

Algorithm 3: Acceleration framework for DRT tasks.

1. for each $k = 1, \ldots, n$ do
   2. compute the utilization $u_k$ of task $u_k$
   3. compute $lmt$ by (6), and let $t = lmt$
   4. while $t \geq 0$ do
      5. for each $k = 1, \ldots, n$ do
         6. compute $DBF_k(t)$
         7. compute $DBF(t)$ by (2)
      8. if $t > DBF(t)$ then
         9. $t := DBF(t) - 1$
      10. else
          11. return "infeasible"
   12. return "feasible"

In Alg. 3, we first compute the interval-size bound $lmt$ according to Thm. 2, and initialize $t$ to be $lmt$ (Lines 1 to 3). We check the interval sizes $t$ in a backward checking way: for any interval size $t$ of interest, we check whether the demand bound function $DBF(t)$ is less than $t$. If this is the case, then we set $t := DBF(t) - 1$ (Line 9). Otherwise, an overflow occurs, and we return "unfeasible" (Line 11). If no overflow occurs during checking progress, we return "feasible" (Line 12).

The main benefit of using this method is the following. We do not need to check interval sizes in a point-to-point way. Instead, once we complete the check of interval size $t$, and find that $\Delta = t - DBF(t) \geq 0$, we then can directly “jump” to the smaller interval size $t' = t - \Delta - 1$ which may be far from $t$, and start the check procedure for the new interval size...
t' \). The correctness of this method is based on the observation described in the following lemma.

**Lemma 1.** For any interval size \( t \), if \( \text{DBF}(t) < t' \), then \( \text{DBF}(t') \leq t' \), \( \forall t' \in [\text{DBF}(t), t] \).

**Proof.** Since the demand bound function is non-decreasing when time increases, for any \( t' \in [\text{DBF}(t), t] \), \( \text{DBF}(t') \leq \text{DBF}(t) \). Moreover, since \( t' \geq \text{DBF}(t) \), we have \( \text{DBF}(t') \leq t' \). \( \square \)

According to Lem. 1, for any interval size \( t \) such that \( \text{DBF}(t) < t \), there is no interval size within \( [\text{DBF}(t), t] \) violating the condition of Lem. 1. As illustrated in Fig. 4, the DBF line is always below the threshold line \( DBF(t) = t \) during \( [\text{DBF}(t), t] \). Therefore, it does not need to check the interval sizes in \( [\text{DBF}(t), t] \). In Sec. V, experimental results show that during the schedulability test, only a few interval sizes should be checked.

Fig. 4. The basic idea of our acceleration technique.

So far, we only introduce the framework of our method, but leave two main problems (e.g., demand bound function (DBF) computation and utilization computation) that should be solved. Certainly, one choice is to solve the above problems by directly applying Stigge’s method in [10]. As stated in Sec. III-B, Stigge’s DP approach does not fit the acceleration framework (of Alg. 3). In the following, we revisit the two computation problems, and develop more efficient techniques to solve these problems. In more details,

- **utilization computation.** We observe that the utilization computation that was solved by Stigge in pseudo-polynomial time actually has a polynomial time complexity, and therefore, we propose a linear program (LP) for computing the utilization as shown in Sec. IV-A.

- **DBF computation.** We propose an integer linear program (ILP) to formulate the DBF computation (see in Sec. IV-B). By using the ILP technique, we can directly solve \( \text{DBF}_k(t) \), which does not restrict to a point-to-point forward checking procedure, and thus, it fit the acceleration framework of Alg. 3 better. The empirical result shows that our ILP-based method is much faster than Stigge’s DP algorithm.

### A. Utilization Computation

According to (5), we know that computing \( \tau_k \)’s utilization \( u_k \) means to find the densest cycle of \( G(\tau_k) \), where the density \( u(\pi) \) of a cycle \( \pi \) is defined in (4). We use linear programming (LP) techniques to solve the densest cycle problem. We first give some useful notations and variables as follows.

For any vertex \( v_i \in G(\tau_k) \), we use \( \text{PRED}(v_i) \) to denote the set of \( v_i \)’s predecessors, i.e., \( \text{PRED}(v_i) = \{ v_j \mid (v_j, v_i) \in E(\tau_k) \} \), and we use \( \text{SUCCE}(v_i) \) to denote the set of \( v_i \)’s successors, i.e., \( \text{SUCCE}(v_i) = \{ v_j \mid (v_j, v_i) \in E(\tau_k) \} \). For any edge \( (v_i, v_j) \) of \( G(\tau_k) \), we denote a variable \( x_{ij} \leq 1 \) such that \( x_{ij} > 0 \) if the densest cycle travels \( (v_i, v_j) \). Otherwise, \( x_{ij} = 0 \). The LP model is given as follows.

\[
\text{MODEL I:} \quad \max u_k = \sum_{(v_i, v_j) \in E(G(\tau_k))} e_{ij} x_{ij} \quad \text{subject to} \quad \sum_{v_j \in \text{PRED}(v_i)} x_{ij} - \sum_{v_j \in \text{SUCCE}(v_i)} x_{ij} = 0, \quad v_i \in G(\tau_k) \quad (10)
\]

Constraint (10) ensures that the flows coming into and going out of the vertex \( v_i \) are balanced, which formulates the cycle \( \pi \) of \( G(\tau_k) \). Constraint (11) ensures that \( \sum_{(v_i, v_j) \in \pi} e_{ij} x_{ij} \) equals the density \( u(\pi) \) of cycle \( \pi \). Objective function (9) maximizes cycle \( \pi \)’s dense, and thus obtain the utilization \( u_k \) of \( G(\tau_k) \).

**Correctness.** In the following, we discuss whether our LP model is capable to solve the densest cycle of \( G(\tau_k) \). We start with the condition that a densest cycle needs to satisfy (see Lem. 2).

**Lemma 2.** The densest cycle is a simple cycle\(^1\).

**Proof.** Suppose that the densest cycle \( \pi \) is not a simple cycle, and thus, \( \pi \) is constructed by several simple cycles. Without loss of generality, as illustrated in Fig. 5, we let \( \pi \) consist of two simple cycles \( \pi_1 \) and \( \pi_2 \) with different densities, i.e., \( \pi : u(\pi_1) > u(\pi_2) \).

By (4), the densities of cycles \( \pi_1 \) and \( \pi_2 \) are calculated as \( u(\pi_1) = \frac{e(\pi_1)}{l(\pi_1)} \) and \( u(\pi_2) = \frac{e(\pi_2)}{l(\pi_2)} \), where \( e(\pi_i) \) and \( l(\pi_i) \) are the workload and the length of the cycle \( \pi_i \) respectively (for \( i = 1, 2 \)). From \( u(\pi_1) > u(\pi_2) \), we have

\[
\frac{e(\pi_1)}{l(\pi_1)} > \frac{e(\pi_2)}{l(\pi_2)}
\]

\[
\iff e(\pi_1) l(\pi_2) > e(\pi_2) l(\pi_1) > e(\pi_2) l(\pi_1) + e(\pi_1) l(\pi_1)
\]

\[
\iff e(\pi_1) > e(\pi_1) + e(\pi_2)
\]

\[
\iff \frac{e(\pi_1)}{l(\pi_1)} > \frac{e(\pi_1) + e(\pi_2)}{l(\pi_1) + l(\pi_2)}
\]

and since \( e(\pi) = e(\pi_1) + e(\pi_2) \) and \( l(\pi) = l(\pi_1) + l(\pi_2) \), we have \( \frac{e(\pi)}{l(\pi)} > \frac{e(\pi_1)}{l(\pi_1)} \). By (4), we eventually derive \( u(\pi_1) > u(\pi_2) \), i.e., the simple cycle \( \pi_1 \) is denser than \( \pi \), which contradicts to the assumption that \( \pi \) is the densest cycle. \( \square \)

Fig. 5. The cycle \( \pi \) contains two simple cycles \( \pi_1 \) and \( \pi_2 \).

We use the following lemma to show that the solution space of our LP model is restricted to simple cycles of \( G(\tau_k) \).

**Lemma 3.** MODEL 1’s solution is a simple cycle in \( G(\tau_k) \).

**Proof.** Constraint (10) ensures that the solution of MODEL I is a set of simple cycles as illustrated in Fig. 6. Without loss of generality, we assume that the solution \( X = \{ x_{ij} \mid (v_i, v_j) \in E(\tau_k) \} \) is a set of simple cycles in \( G(\tau_k) \).

\(^1\)A cycle \( \pi \) is simple if \( \pi \) travels each vertex of \( G(\tau_k) \) at most once, except the original vertex of \( \pi \).
G(τ_i) \) of MODEL I consists of two simple cycles \( π_1 \) and \( π_2 \), and moreover, we let \( u(π_1) > u(π_2) \). We rewrite Constraint (11) as \( \sum_{(v_i,v_j) \in π_1} p_{ij}x_{ij} + \sum_{(v_i,v_j) \in π_2} p_{ij}x_{ij} = 1 \), where \( x_{ij} \in X \) for each edge \( (v_i,v_j) \in G(τ_k) \). We modify the solution \( X \) as follows.

- For each edge \( (v_i,v_j) \in π_2 \), we let \( x'_{ij} = 0 \).
- For each \( (v_i,v_j) \in π_1 \), we let \( x'_{ij} = x_{ij} + Δ \), where \( Δ \) is a constant defined as follows.

\[
Δ = \frac{1 - \sum_{(v_i,v_j) \in π_1} p_{ij}x_{ij}}{\sum_{(v_i,v_j) \in π_1} p_{ij}}
\]

(12)

- For each edge \( (v_i,v_j) \in G(τ_k) - (π_1 \cup π_2) \), we let \( x'_{ij} = 0 \).

We now obtain the new solution \( X' = [x'_{ij}; (v_i,v_j) \in G(τ_k)] \) that only contains simple cycle \( π_1 \). In the following, we prove that the solution \( X' \) satisfies the constraints of MODEL I.

**Satisfaction of Constraint (10).** On the one hand, since \( X \) satisfies Constraint 10, for any vertex \( v_i \) visited along the cycle \( π_1 \), we know that for \( v_i \in π_1 \),

\[
\sum_{v_j \in \text{pred}(v_i)} x_{ji} = \sum_{v_j \in \text{succ}(v_i)} x_{ij}
\]

\( \Leftrightarrow \)

\[
\sum_{v_j \in \text{pred}(v_i)} (x_{ji} + \Delta) = \sum_{v_j \in \text{succ}(v_i)} (x_{ij} + \Delta)
\]

\( \Leftrightarrow \)

\[
\sum_{v_j \in \text{pred}(v_i)} x'_{ji} = \sum_{v_j \in \text{succ}(v_i)} x'_{ij}
\]

(13)

On the other hand, for any vertex \( v_i \) that is not in cycle \( π_1 \), the edges \( (v_i,v_j) \) going into \( v_i \) correspond to zero variables \( x_{ji} = 0 \), and the edges \( (v_i,v_j) \) going out of \( v_i \) also correspond to zero variables \( x_{ij} = 0 \), i.e., for \( v_i \in G(τ_k) - π_1 \),

\[
\sum_{v_j \in \text{pred}(v_i)} x'_{ji} = \sum_{v_j \in \text{succ}(v_i)} x'_{ij} = 0
\]

(14)

- **Satisfaction of Constraint (11).** Since only the edges \( (v_i,v_j) \) visited along \( π_1 \) have positive variable, we have

\[
\sum_{(v_i,v_j) \in G(τ_k)} p_{ij}x'_{ij} = \sum_{(v_i,v_j) \in π_1} p_{ij}x'_{ij}
\]

and since \( x'_{ij} = x_{ij} + Δ \), for each \( (v_i,v_j) \in π_1 \), we have

\[
\sum_{(v_i,v_j) \in G(τ_k)} p_{ij}x'_{ij} = \sum_{(v_i,v_j) \in π_1} p_{ij}(x_{ij} + Δ)
\]

and by (12), we know that

\[
\sum_{(v_i,v_j) \in G(τ_k)} p_{ij}x'_{ij} = \sum_{(v_i,v_j) \in π_1} p_{ij}x_{ij} + 1 - \sum_{(v_i,v_j) \in π_1} p_{ij}x_{ij} = 1
\]

This indicates that the solution \( X' \) satisfies Constraint (11).

In sum, the newly obtained solution \( X' \) satisfies the constraints of MODEL I and consists of a simple cycle.

According to Lem. 3, any solution of MODEL I corresponds to a simple cycle of \( G(τ_k) \). Moreover, by the objective function 9, we know that MODEL I must solve the densest simple cycle of \( G(τ_k) \). Moreover, according to Lem. 2, we conclude the correctness of MODEL I in Corollary 1.

**Corollary 1.** MODEL I solves the densest cycles in \( G(τ_k) \).

**Complexity.** The following proposition indicates that our LP model is solved within polynomial time.

**Proposition 1.** There are \( m \) variables and \( n + 1 \) constraints in MODEL I, where \( n \) is the number of vertices in \( G(τ_k) \), and \( m \) is the number of edges in \( G(τ_k) \).

**Proof.** Since each edge \( (v_i,v_j) \) of \( G(τ_k) \) corresponds to a variable \( x_{ij} \), MODEL I contains \( m \) variables.

For each vertex \( v_i \in G(τ_k) \), we construct a constraint of (10). Moreover, there is only one constraint of (11) MODEL I contains \( n + 1 \) constraints in total.

Recall that Stigge’s method for computing utilization \( u_k \) has the pseudo-polynomial time complexity, e.g., \( O(C^2n) \), where \( n \) is the number of vertices in \( G(τ_k) \), and \( C = \sum_{(v_i,v_j) \in G(τ_k)} p_{ij} \) is the summation of the periods of all edges in \( G(τ_k) \). Clearly, our LP model outperforms Stigge’s method from the perspective of computation complexity.

**B. DBF Computation**

For a given \( t \), computing \( DBF_k(t) \) of \( τ_k \) is to find the optimal path \( π \) of \( G(τ_k) \) such that \( π \) has the maximum workload and \( π \)’s length is bounded by \( t \) according to (1). We formulate this problem as an integer linear program (ILP). Before going into details, we first add two auxiliary vertices \( v_{src} \) and \( v_{snk} \) (without execution time) to \( G(τ_k) \). For any vertex \( v_i \), we add the edges \( (v_i,v_{src}) \) and \( (v_i,v_{snk}) \) with the periods \( p_{src,i} = 0 \) and \( p_{i,snk} = d_i \). We denote by \( G'(τ_k) \) the digraph with these auxiliary vertices and edges. Any path of \( G(τ_k) \) is equivalent to a path of \( G'(τ_k) \) that starts with the source vertex \( v_{src} \) and ends at the sink vertex \( v_{snk} \). For any edge \( (v_i,v_j) \) of \( G'(τ_k) \), we denote an integer variable \( z_{ij} \) to represent the times we travel the edge \( (v_i,v_j) \). The ILP model is given as follows.

**MODEL II**

\[
\text{max } DBF_k(t) = \sum_{(v_i,v_j) \in G'(τ_k)} e_{ij}z_{ij}
\]

\( s.t. \)

\[
\sum_{v_j \in \text{pred}(v_i)} z_{src,i} = 1
\]

\( \sum_{v_i \in \text{pred}(v_{snk})} z_{i,snk} = 1 \)

\[
\sum_{v_j \in \text{pred}(v_i)} z_{ij} - \sum_{v_j \in \text{pred}(v_i)} z_{ij} \leq 0, v_i \in G'(τ_k) \)

(18)

\[
\sum_{v_j \in \text{pred}(v_i)} z_{ij} \leq t
\]

(19)

Objective function (15) maximizes the workload of the path \( π \) of \( G'(τ_k) \). Constraints 16 and 17 respectively enforce the
path $\pi$ to start with the source vertex $v_{src}$ and ends at the sink vertex $v_{snk}$. Constraint 18 ensures that for each vertex $v_i$ of $G(\tau_k)$, the flows coming into $v_i$ and the flows going out of $v_i$ are the same, which is necessary to formulate a path. Here $\text{PRE}(v_i)$ contains all the predecessors of $v_i$ (including $v_{src}$), and $\text{SUCC}(v_i)$ contains all the successors of $v_i$ (including $v_{snk}$). Constraint (19) ensures that the length of the path $\pi$ is bounded by $t$.

It should be noted that by now we cannot use MODEL II to exactly solve $\text{DBF}_G(t)$ but we can only provide an upper bound of $\text{DBF}_G(t)$ by using MODEL II, since MODEL II may have some feasible solutions that are not the paths in $G(\tau_k)$. We give a counterexample as follows.

![Fig. 7. The counterexample for MODEL II.](image)

**Example 2.** We give a DRT task graph $G(\tau_k)$ with additional vertices $v_{src}$ and $v_{snk}$ as shown in Fig. 7. For each vertex $v_i \in G(\tau_k)$, there are two edges ($v_{src}, v_i$) and ($v_i, v_{snk}$). For the sake of convenience, we do not show all edges associated with $v_{src}$ and $v_{snk}$ in Fig. 7. For a given $t = 100$, we use MODEL II to solve the maximum workload of the path in $G(\tau)$, and obtain the solution $X$ as described as follows: $x_{src, 1} = 1$, $x_{12} = 1$, $x_{23} = 1$, $x_{34} = 1$, $x_{4, snk} = 1$, $x_{56} = 1$, $x_{67} = 1$ and $x_{75} = 1$. The solution $X$ corresponds to a path $\pi_1$ from $v_{src}$ to $v_{snk}$ and a cycle $\pi_2$ that contains vertices $v_5$, $v_6$ and $v_7$, as marked red in Fig. 7. The total length of $\pi_1$ and $\pi_2$ is 70 ($\leq 100$), and the total workload solved by MODEL II is $4 + 3 = 7$. Therefore, the demand bound function solved by MODEL II is $\text{DBF}_G(100) = 7$.

Actually, the maximum workload among all paths of $G(\tau_k)$ with length no more than 100 is 4. For example, there are two types of possible paths whose lengths are bounded by 100 and which have the maximum workloads. The first path is $\pi_3 = (v_{src}, v_1, v_2, v_3, v_4, v_{snk})$ with length 4 and workload 4. The second path is $\pi_4 = (v_{src}, v_5, v_6, v_7, v_5, v_{snk})$ with length 100 and workload 4. Therefore, the actual demand bound function $\text{DBF}_G(100)$ equals 4.

Example 2 shows that the demand bound function solved by MODEL II may be much larger than the actual value, e.g., the ratio of the gap between them is about 75%. Therefore, MODEL II cannot solve the demand bound function exactly. The main reason is that the solution of MODEL II may contain isolated cycles. To preclude isolated cycles, we will add some additional variables and constraints into MODEL II as follows.

For each vertex $v_i \in G(\tau_k)$, we define a boolean variable $y_i$ such that $y_i = 1$ if $v_i$ is traveled along the path $\pi$. Otherwise, $y_i = 0$. The following constraints show the relation between the variables $z_{ij}$ and $x_i$. For each vertex $v_i \in G(\tau_k)$, using the big number $B = \infty$,

$$\sum_{v_j \in \text{SUCC}(v_i)} z_{ij} \geq y_i \quad (20)$$

$$By_i \geq \sum_{v_j \in \text{SUCC}(v_i)} z_{ij} \quad (21)$$

Constraint (20) ensures that if the vertex $v_i$ is traveled (e.g., $y_i = 1$), then at least one edge goes out of $v_i$, i.e., $\sum_{v_j \in \text{SUCC}(v_i)} z_{ij} \geq 1$. Constraint (21) ensures that if some edge going out of $v_i$ is traveled (e.g., $\sum_{v_j \in \text{SUCC}(v_i)} z_{ij} \geq 1$), then the vertex $v_i$ must be traveled (e.g., $y_i = 1$).

In the following, we will propose the constraint to preclude isolated cycles. Before going into details, we first introduce some useful notations.

**Definition 5 (cut set).** For vertex $v_i \in G(\tau_k)$, the cut set $\text{CUT}(v_i)$ is the set of edges such that

- The vertex $v_i$ is not reachable from the source vertex $v_{src}$ if the edges of $\text{CUT}(v_i)$ are all removed.
- For any subset $E \subset \text{CUT}(v_i)$, the vertex $v_i$ is reachable from the source vertex $v_{src}$ if the edges of $E$ are removed.

Moreover, we denote the minimum cut set $\text{CUT}_{\text{min}}(v_i)$ as the cut set of $v_i$ that has the minimum edge number, i.e.,

$$\text{CUT}_{\text{min}}(v_i) = \min_{\text{CUT}(v_i) \in G(\tau_k)} |\text{CUT}(v_i)|$$

For example, in Fig. 7, the cut set $\text{CUT}(v_6)$ of $v_6$ may be $\{(v_5, v_6)\}$ or $\{(v_{src}, v_5), (v_2, v_5)\}$. The minimum cut set $\text{CUT}_{\text{min}}(v_6) = \{(v_5, v_6)\}$. From Def. 5, we know that if a path starting with $v_{src}$ travels $v_i$, then the path $\pi$ must travel the edge in $\text{CUT}(v_i)$. This is formulated by the following constraint. For each $v_i \in G(\tau)$, using big number $B = \infty$,

$$B(1 - y_i) + \sum_{(v_j, v_i) \in \text{CUT}(v_i)} z_{ij} \geq 1 \quad \forall \text{CUT}(v_i) \in G(\tau_k)$$

The correctness of Constraint (22) is illustrated as follows. If the vertex $v_i$ is traveled (e.g., $y_i = 1$), then Constraint (22) becomes $\sum_{(v_j, v_i) \in \text{CUT}(v_i)} z_{ij} \geq 1$, indicating that the edge of $\text{CUT}(v_i)$ must be traveled. If the vertex $v_i$ is not traveled (e.g., $y_i = 0$), then Constraint (22) becomes $B \geq 1$, indicating that it is not necessary to travel the edges of $\text{CUT}(v_i)$.

We combine the newly added variables $y_i$ (for all vertices $v_i \in G(\tau_k)$) and Constraints (20) to (22) into MODEL II, and obtain the following ILP model, denoted as MODEL III, i.e.,

**MODEL III**

Objective function (15)

s.t. Constraints (16) to (22)

Clearly, MODEL III can find the path with the length no more than $t$ and with the maximum workload, and thus, exactly derives $\text{DBF}_G(t)$. However, MODEL III may have exponential number of constraints in (22), since the number of cut sets $\text{CUT}(v_i)$ for a vertex $v_i$ may be exponential in $m$, the number of edges in $G(\tau_k)$. It is very hard to enumerate all constraints of (22), and thus, MODEL III cannot be directly solved. In the following, we propose a row generation algorithm to solve MODEL II efficiently.
Row Generation Algorithm

We solve \textsc{Model III} in an iterative way. The main idea is that we do not enumerate all constraints of \textsc{Model III}, and instead, we solve \textsc{Model III} with part of its constraints, and iteratively add the constraints only when necessary. As we know that in the operational research (OR) community, the variables in the ILP are usually called as \textit{columns}, and the constraints are called as the \textit{rows} of the ILP. Accordingly, our approach that iteratively adds new constraints is called the \textit{row generation algorithm} as given in Alg. 4.

\begin{algorithm}
\caption{Row generation algorithm.}
\begin{algorithmic}[1]
\STATE solve \textsc{Model II}, and obtain the solution $Z^*$
\WHILE{the solution $Z^*$ contains no isolation cycles}
\FOR{each isolation cycle $\pi$ of $G(t_k)$}
\STATE $\text{CUT} := \arg\min_{\ell_i\in\pi} |\text{CUT}_{\min}(\ell_i)|$
\STATE add $B(1 - y_{ij} + \sum_{(v_i,\ell_j)\in\text{CUT}} z_{ij}) \geq 1$ into \textsc{Model II}
\ENDFOR
\STATE solve \textsc{Model II} (with newly added constraints), and obtain $Z^*$
\RETURN $\text{DBF}_k(t) = \sum_{(v_i,\ell_j)\in G(t_k)} e_i z_{ij}$
\end{algorithmic}
\end{algorithm}

In Alg. 4, we solve \textsc{Model II}, the initial ILP model with (15) to (19), and obtain the solution $Z^*$. If $Z^*$ indicates a feasible path, then the objective function of $Z^*$ is returned as $\text{DBF}_k(t)$. Otherwise, there are some isolation cycles in the solution $Z^*$. For each isolation cycle $\pi$, we find the minimum cut set $\text{CUT}$ among all the vertices of $\pi$ (Lines 3 to 4). Based on the minimum cut set $\text{CUT}$, we add a constraint of (22) into \textsc{Model II} as shown in Line 5. Then we solve the updated model, and obtain the solution $Z^*$. If $Z^*$ contains the isolation cycles, then we start the computation process (Lines 2 to 6) in the next iteration. This process repeats until the solution contains no isolation cycle.

\section{V. Evaluations}

This section reports the comparison between Stigge’s method $A_{\text{STG}}$ in [10] and our method $A_{\text{SUN}}$. We randomly generate the DRT task sets consisting of $n$ DRT tasks and with the total utilization $u$. Each DRT task has 5 to 9 vertices. The execution time of each vertex ranges from 1 to 4, and the period of each edge ranges from 100 to 200. We implement our models by using the Z3 solver embedded in python 3.5. The code runs on a PC with Intel Core i5-6300U CPU at 2.4GHz with 8G RAM. Fig. 8 to 10 conduct experiments with different combinations of parameters. The values of the configurations are written in the figure caption. For each data point, 1000 random experiments have been run. The longitudinal axes are log10 transformed to adjust for the wide range of data.

In Fig. 8, we record the numbers of interval sizes checked by $A_{\text{STG}}$ and $A_{\text{SUN}}$, denoted as $\Gamma_{\text{STG}}$ and $\Gamma_{\text{SUN}}$ respectively. The checked interval-size number $\Gamma_{\text{SUN}}$ is 5.73 on average, which does not significantly change when the task number $n$ increases, and slightly increases when the utilization $u$ increases. $\Gamma_{\text{STG}}$ is 513.43 on average, 90 times larger than $\Gamma_{\text{SUN}}$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig8.png}
\caption{Evaluate the number of checked interval sizes.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig9.png}
\caption{Evaluate computation time for schedulability analysis.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig10.png}
\caption{Evaluate computation time for solving utilization.}
\end{figure}

We evaluate the run time for the schedulability analysis. We use $t_{\text{SUN}}$ to denote the run time for solving \textsc{Model I} and \textsc{Model II}, and use $t_{\text{STG}}$ to denote the run time of Stigge’s schedulability analysis method. As shown in Fig. 9, the run time $t_{\text{SUN}}$ of our method is 0.93 second on average, and the run time $t_{\text{STG}}$ of Martin’s method is 21.04 seconds on average. The run time $t_{\text{STG}}$ (as well as $t_{\text{SUN}}$) increases when the task number $n$ and the total utilization $u$ increases. Our method is 23 times faster than Martin’s method on average.

We also evaluate the run time for solving the utilization. We use $t_1$ to denote the run time for solving \textsc{Model I}, and use $t_u$ to denote the run time of Stigge’s utilization computation method. As shown in Fig. 10, \textsc{Model I} can be solved within 48ms, and Stigge’s method exceeds 1100ms on average. The run time $t_1$ increases when the vertex number $|V_k|$ and the utilization $u_k$ of $G(t_k)$ increases.

\section{VI. Related Work}

Graph-based task models include recurring branching [3], recurring [4], non-cyclic recurring [5], GMF [6], and so on. One of the most expressive graph-based task models is the Digraph Real-Time (DRT) task model [10] using arbitrary directed graphs for modeling task activations. This section mainly summarizes the related work for DRT tasks. Stigge et al. [10] first propose the DRT task model, and develop a pseudo-polynomial time approach to analyze the schedulability of DRT tasks under the dynamic priority scheduling (e.g., EDF). In [8], they further clarify the tractability of schedulability analysis of DRT tasks (under EDF). For static priority (SP) schedulers, Stigge et al. [12] show the intractability of the schedulability analysis of DRT tasks, and in [13], they propose an iterative approach to efficiently cope with the combinatorial
explosion in the analysis process. Using the similar technique, Stigge et al. [14] solve the response time analysis (RTA) problem for DRT task under SP scheduling. Guan et al. [15] propose an approximation algorithm to solve the RTA problem for DRT tasks under SP scheduling, and in [16], they use the real-time calculus for the delay analysis of each vertex in DRT tasks. Gu et al. [17] use graph transformation methods to improve the schedulability of DRT tasks under SP scheduling. For both dynamic and static priority schedulers, Zeng et al. [18] use the max-plus algebra technique to improve the efficiency of the schedulability analysis for the strongly connected DRT tasks. Please see the survey [19] for details.

DRT task models are extended into different application scenarios. Guan et al. [20] and Abdullah et al. [21] study the scheduling strategies for the DRT tasks with resource sharing constraints. Ekberg et al. [22] use DRT task model to describe the mixed criticality system. Mohaqeqi et al. [23], Fradet et al. [24] and Xu et al. [25] study the dependent DRT tasks, i.e., there are dependency constraints between DRT tasks. Ben et al. [26] study the DRT tasks with the probabilistic worst-case execution time. Sun et al. [27], Zahaf et al. [28], and Houssam et al. [29] extend the DRT task model to support the inner parallel tasks. Mohaqeqi et al. [30] use the DRT task model to formulate the data flow graphs. Abdullah et al. [31] and Mohaqeqi [32] use DRT tasks to formulate Ada code.

VII. CONCLUSION

The traditional feasible analysis for the DRT task model uses a dynamic programming based approach to deal with the exponential explosion, but is hard to be accelerated. In this paper, we provide efficient techniques to accelerate the analysis of DRT tasks. By using LP techniques, we reduce utilization computation’s complexity from pseudo-polynomial time to polynomial time. By using ILP techniques, we propose an efficient method to compute the demand bound function, which facilitates the implementation of acceleration techniques. Experimental work shows that our method has a much shorter run time than the traditional method.

REFERENCES