A General Scheduling Framework for Real-time DAG Task Systems

Anonymous

***Abstract*—In this paper, we consider the real-time scheduling problem for communicating tasks modeled with directed acyclic graphs (DAG). Different from traditional methods, we model it as a nonlinear program (NLP) and use gradient-based methods for efficient solutions. Different requirements can be freely added as constraints, which makes the proposed method effective for a wide range of problems. The optimization method also gives a new necessary and sufficient schedulability analysis method which are easy to verify. Furthermore, the proposed NLP scheduling method can utilize any available scheduling algorithms as the initialization method, and improve their performance even further. Experimental evaluation shows big advantages of the proposed approach against simple heuristic methods in complicated problems.**

I. Introduction

As modern cyber-physical systems (CPS) become more and more complicated, efficient and safe scheduling methods gradually become a critical issue. Different from traditional computation platforms where both computation tasks and requirements are simple and straightforward, recent applications such as drones and automotive usually feature with heterogeneous computation resources, a large number of computation tasks with complicated dependency relationship, non-sustainable scheduling requirements, etc. Traditional scheduling analysis and algorithms usually focus on only one or two aspects of these requirements, which make them less applicable in real systems. Furthermore, by making pessimistic assumptions on the scheduling model, complicated schedulability analysis is proposed to find a safe scheduling method at the cost of pseudo polynomial or even exponential complexity. In reality, many pessimistic assumptions usually cause a big waste on hardware resources, or may be too complicated to be implemented in real systems reliably, making the scheduling methods less practical.

In this paper, we propose a novel, general scheduling framework which aims to schedule complicated computation systems with multiple scheduling requirements in a as less pessimistic as possible way. Inspired by Baruah [1], we model the scheduling problem as an optimization problem, where the start time of each job instance is the variable. However, we propose to solved it by trust-region methods rather than using integer linear programming (ILP) solvers. Trust-region methods are originally proposed to solve nonlinear least square (NLLS) problems by Levenberg [2]. After several decades of development, it becomes very popular in many areas and achieves great success because of its excellent performance and high efficiency. By switching to trust-region methods, our scheduling method does not require constraints to be linear anymore as in Baruah [1], which adds great flexibility in modeling. A large portion of scheduling constraints can be incorporated into our scheduling framework freely, and a safe scheduling method will be found with only polynomial complexity ($O(N^2)$ in average, $O(N^3)$ in worst case, where N is the number of variables). This is a major improvement on run-time speed than Baruah [1] where ILP solver is used to solve a problem with $O(N^3)$ variables. Such fast speed is achieved not only by gradient-based optimization method, which are usually much faster than traditional methods such as meta-heuristic or branch-and-bound (BnB), but also by exploiting problem-specific structure with orders of speed-up possible.

The proposed scheduling framework is "compatible" with most available scheduling methods because of its flexible initialization mechanism. Usually, nonlinear programming (NLP) algorithms including trust-region methods require an initial solution to begin with. Most scheduling algorithms (such as RM in our case) can be used to provide an initial solution no matter whether they can satisfy all the scheduling requirements or not. If not, then the proposed algorithm is very likely to improve it and return a better solution.

Considering that many scheduling problems are NP-hard, a general, exact scheduling algorithm with polynomial complexity and no pessimistic assumption is usually very difficult to find. As such, this paper does not aim to find a general solver with polynomial complexity for NP problems. Our argument is that a sufficient scheduling algorithm with less pessimism is always possible with a good optimizer, which is the main focus of this paper.

II. Related Work

A. Dependency tasks scheduling

It is very common that modern computation tasks have complicated dependency relationship, which poses a big challenge in real-time scheduling. Baruah *et al*. [3] first propose to use directed acyclic graph (DAG) to describe such dependency relationship upon multi-processor computation platforms. In this model, all the nodes of one DAG share the same period and deadline parameter, which simplifies the scheduling problem greatly. However, even so, general DAG scheduling problem is shown to be NP-hard in the strong sense [4], and so most work seeks for approximate safe solutions instead. Following this direction, both fixed-task priority (FTP, such as deadline monotonic) and dynamic-priority (such as earliest deadline first, or EDF) have been studied. Several schedulability test based on some pessimistic assumptions are also proposed, such

as worst-case response time (WCRT) analysis [3], speedup bound test, work function test [5], etc. More recent work gradually improves WCRT analysis bound under different model assumptions, such as arbitrary deadline [6], constrained deadline [7]. Some work tries to analyze response time at a more fine-grained level, and considers carry-in, body and carry-out to provide a tighter analysis model. Melani *et al*. [8] utilize Graham *et al*. [9]'s analysis bound and provide a better analysis that considers inter-task interference for both global EDF and FTP. Pathan *et al*. [10] consider a two-level FTP scheduler at task-level and sub-task level. They also propose a less pessimistic analysis model on interference compared with [8]. Later, the interference model is even improved by Fonseca *et al*. [11] which considers workload distribution and topology of DAG.

More complicated DAG models are also considered for scheduling. Limited preemption DAG [12], [13] assumes only DAG task at node boundaries can be preemptive. Federated scheduling [14], [15] considers task based on their utilization. Higher utilization tasks are assigned dedicated cores, and the lower utilization tasks are scheduled under shared cores. Conditional DAG [16], [17] allows different execution path based on *if-else* or *switch* conditions. Although not considered in this paper, the proposed framework does have the potential to schedule these variations.

As a special case of DAG model, cause-effect chains only consider DAG tasks with only one possible path. Different from DAG model, cause-effect chain usually assumes different nodes may have their own periods, which makes traditional DAG analysis not applicable. Abdullah [18] considers WCRT in the context of non-blocking situation and proposes a latency analysis method. Gunzel *et al*. [19] provide reaction time and data age analysis method for asynchronous distributed cause-effect chains at fine-grained level.

B. Demand bound function in scheduling

Demand bound function (DBF) has been proposed in schedulability analysis by Baruah *et al*. [20] long time ago. Later, Jeffay *et al*. also use it in EDF schedulability analysis under constrained situations [21] and give a sufficient schedulability test. Recently, Baruah [1] propose to apply DBF in DAG scheduling with given processor assignment. Utilizing insights from operations research, Baruah formulate the scheduling problem as an integer linear programming problem. Similar as before, DAG model in this paper only has a single period, which makes the variable space quite small, as such solvable by ILP solver. Different from [1], our paper considers a much more complicated DAG model where different nodes have different periods, inspired from RTSS 2021 Industry Challenge. The whole hyper-period must be explored to provide a safe schedule, and so makes original ILP solver highly intractable.

C. Optimization techniques

Optimization including nonlinear programming (NLP) has been widely applied in real-time systems, robotics and many other areas. Usually, it is very difficult to directly obtain solutions for general NLP, and so most popular methods follow an iterative manner [22]. Given an initial solution, a new update is formulated based on heuristic algorithms. Many famous meta-heuristic algorithms such as simulated-annealing [1], [23], genetic algorithms [24] are proposed based on inspirations from physical world. Such algorithms usually can be applied to solve a very large range of problems because they almost do not pose specific requirements on either objective function or constraints. However, such generality makes them very easy to ignore problem specific structure to explore, and so suffer from either slow speed or poor performance [25].

III. Review on Nonlinear programming

In this section, we give a brief review about some classical optimization techniques that are used in this paper. Considering an unconstrained nonlinear least-square problem as an example:

$$\min_x \|f(x) - b\|_2^2 \tag{1}$$

where $f(x) : R^n \to R^m$ is a nonlinear function. We want to find optimal x that minimizes objective function 1.

Different from meta-heuristics algorithms, gradient-based methods usually have faster convergence speed based on local information and also better performance for continuous optimization. Inside iterations, steepest descent (or gradient descent) updates the variables based on first-order gradient (usually denoted as Jacobian matrix J), while Gauss-Newton (GN) provides faster convergence speed based on second-order (usually denoted as Hessian matrix H) information if available. The updating steps Δ of these two algorithms are given as follows:

$$\Delta_{sd} = \alpha J^T b \tag{2}$$

$$J^T J \Delta_{gn} = J^T b \tag{3}$$

where the step size α is chosen by algorithms such as line search, and the Jacobian matrix J is defined as:

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{4}$$

These two basic algorithms are improved to be applied in more general NLP problems by trust-region methods. The basic idea is to update the variables based on how well the current approximation performs. It only accepts one update if it improves the objective function, otherwise, a new step is formulated to shrink the searching space surrounding the current variable. Inside the trust region, the objective function can be updated by different methods. Levenberg-Marquardt (LM) algorithm [2], [26] introduces a damping factor λ to the Hessian matrix ($J^T J$ in function 1), which not only makes sure the perturbed Hessian to be positive definite when encountering negative curvature, but also could take advantages of both SD and GN by approximating their updating steps under different situations. For example, the update step of LM is given by Δ_{lm}:

$$(J^T J + \lambda \ \mathrm{diag}(J^T J))\Delta_{lm} = J^T b \tag{5}$$

Another representative trust-region algorithm is Powell's dogleg [27] (DL) which is a more efficient alternative to LM. Unlike LM, DL computes SD and GN steps separately, therefore these step information can be reused if one tentative step is rejected. However, DL is less reliable than LM when encountering numerically poorly constrained systems or under-constrained systems.

IV. System model

The system model that we are considering is mostly based on the scheduling problem posed by RTSS 2021 industry challenge [28]. We consider a single directed acyclic graph (DAG) model $\mathcal{G} = (V, E)$ to describe computation tasks. Each node v_i has their own period T_i, worst-case execution time c_i, relative deadline d_i. An edge E_{ij}, or (v_i, v_j), going from node v_i to node v_j, means v_j's input depends on v_i's output. The hyper-period, i.e., least common multiple of periods for all nodes in $\mathcal{G}$, is denoted as H. Within a hyper-period, the k-th instance of node v_i starts execution at time s_{ki} *non-preemptively*, and finishes at $f_{ki} = s_{ki} + c_i$. Bold font **s** represents start time vector for all task sets within a hyperperiod. We do not consider offset for simplicity, which actually can be incorporate easily. For a node v_i with precedence constraints, we denote all its source tasks as $\mathit{pre}(v_i)$, and all its successor as $\mathit{suc}(v_i)$. A path in DAG is described by a node sequence $\lambda_{be} = \{v_b, ..., v_e\}$, which starts at node v_b and ends at node v_e, and is connected in sequence, i.e., $(v_i, v_{i+1}) \in E$. The DAG model is processed by a multi-processor system. We assume all the computation tasks are mapped to a known single computation resource such as CPU or GPU, and do not consider processor assignment.

There are several differences between our DAG model and classical DAG model [4], [29]. At first, each node in our DAG model may have its own period which are not necessarily the same as other nodes. Following [28], we assume each computation node with dependency has a buffer that stores the incoming information. Secondly, the overall DAG graph $\mathcal{G}$ is not necessarily fully connected so that multiple fully connected DAG tasks can be described by a single DAG model. The DAG model under consideration is not only more consistent with the real computation systems [19], [28], but also allows us to mathematically describe more scheduling requirements, which the proposed scheduling framework can incorporate and optimize. Finally, we mainly consider non-preemptive scheduling problems in this paper. However, the proposed method can be easily generalized to preemptive scheduling systems by introducing extra finish time variables for each node instance.

Although the proposed framework is designed to be as general as possible, there is no guarantee that the method will work well with all kinds of constraints. It is possible that some special constraints are highly nonlinear and our optimizer has to be modified accordingly. Regardless of which, there is still a large range of scheduling requirements that we are able to optimize well, including but are not limited to:

- **Schedulability**. All the instances of each task should be schedulable and finish before their deadline:

$$\forall \tau_i, \forall k \in [0,\ \frac{H}{T_i}),\ s_{ik} + c_i \leq (k-1)T_i + d_i \tag{6}$$

- **Period constraints**. All the instances of each task cannot start earlier than the beginning of their periods:

$$\forall i, k,\ s_{ik} \geq T_i(k-1) \tag{7}$$

- **Computation resource bounds**. All the computation resources R_i (e.g., CPU, GPU) are not overloaded for any time interval from t_i to t_j. The mathematical description is given by demand bound function (DBF) [4] as follows:

$$\forall R_i, \text{DBF}(t_i, t_j) \leq t_j - t_i \tag{8}$$

  A more efficient, but equivalent description for this requirement will be given in the methodology section.
- **Sensor fusion bound**. Let's use v_i to denote a task which requires multiple source sensor's output, and $\{v_l | v_l \in \mathit{pre}(v_i)\}$ represents all its source sensor tasks. For any job instances v_{ik} with start time at s_{ik}, the time difference of all the source data tokens v_{lj} must be smaller than pre-defined threshold Θ_s:

$$\forall s_{ik} : s_{lj} \in \{0 \leq j < \frac{H}{T_l} | s_{lj} \leq s_{ik} < s_{l(j+1)}\} : \tag{9}$$

$$\max_l s_{lj} - \min_l s_{lj} \leq \Theta_s \tag{10}$$

- **Sensor freshness** Using the same notation above, for any instance of task v_{ik} and its immediate previous source sensor task instance $\{v_{lj} | v_l \in \mathit{pre}(v_i)\}$, the time period from j-th instance of task v_l finishes processing at $s_{lj} + c_l$ to node v_{ik} begins execution at s_{ik} should be upper bounded by a pre-defined threshold Θ_{sf}:

$$\forall s_{ik}, \forall s_{lj} \in \{0 \leq j < \frac{H}{T_l} | s_{lj} \leq s_{ik} < s_{l(j+1)}\} : \tag{11}$$

$$s_{ik} - s_{lj} - c_l \leq \Theta_{sf} \tag{12}$$

- **DAG dependency**. Each node cannot start until all its dependency tasks have finished.

$$\forall v_i, \forall l \in \mathit{pre}(v_i), s_{l0} + c_l \leq s_{i0} \tag{13}$$

Considering that different nodes have different frequency, over-sampling or under-sampling [18] would be unavoidable. As such, we only explicitly require DAG dependency constraint for the first instance of each node in constraint 13. However, DAG dependency can be enforced for each job instance if explicit dependency between them can be specified.

V. Methodology

We model the scheduling problem as an optimization problem, and propose to use trust-region optimizer to solve it for efficiency. The main framework is shown in Fig. 1.

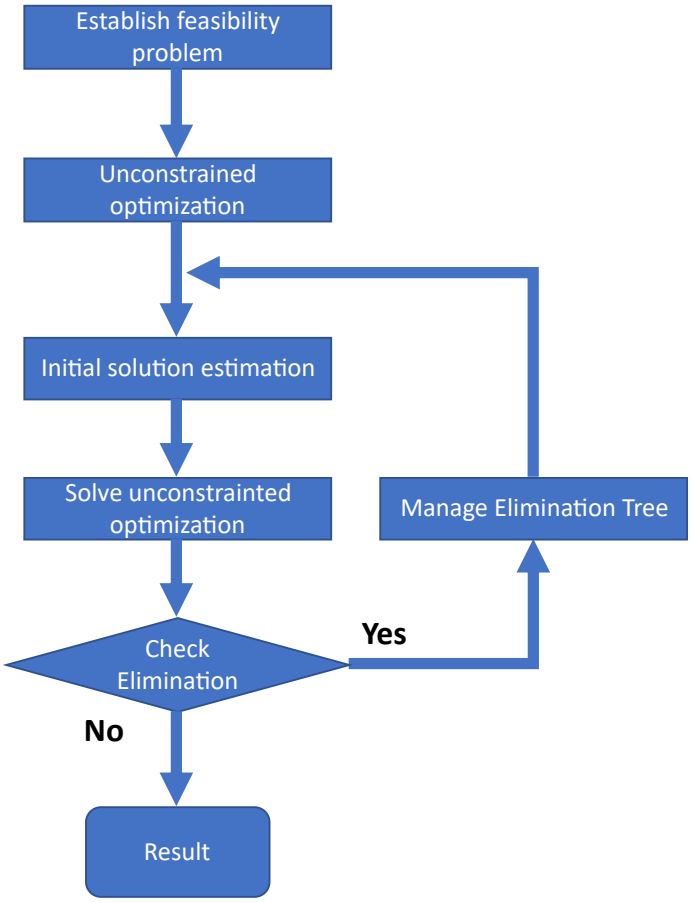

Figure 1: Main optimization framework. After formulating the feasibility problem, we first transform it into an unconstrained least-square optimization problem. We then iteratively estimate an initial solution and solve it. At the end of each iteration, we check and add qualified variables into elimination forest, and formulate a new initial solution accordingly. The loop continues until there are no new variables to eliminate.

A. More efficient schedulability analysis based on DBF

In this section, we propose a new method to analyze computation resource bound constraint 8 for non-preemptive situations based on the ILP formulation proposed by Baruah [1]. Although originally proposed for preemptive scheduling systems, DBF test in [1] can be easily modified to be applied in non-preemptive systems by making the finish time of each task instance depend on its start time. However, this method is very slow with $O(N^3)$ complexity. As such, we propose a new algorithm which yields the same results as the DBF test in [1], but with worst computation cost of only $O(N^2)$ ($O(N\log(N))$ in average) as opposed to $O(N^3)$ in [1].

1) Interval overlapping test: The basic idea is simple: we sort task execution time intervals before calculating DBF. In non-preemptive situations, an equivalent schedulability analysis as DBF analysis can be obtained by calculating "interval" overlapping, where an "interval" is defined as a task instance's execution time interval. For example, the k-th instance of task v_i's execution interval starts at s_{ik}, and ends at $s_{ik} + c_i$. If all the intervals are not overlapped with each other, then the system is schedulable. By sorting all these intervals before detecting possible overlapping, we can avoid redundant overlapping detection and improve overall speed. The pseudo-code is provided by Algorithm 1. As will be proved later, the original computation resource bound constraint 8 is satisfied if Algorithm 1 returns 0.

Complexity for Algorithm 1 is much lower than DBF test($O(N^3)$) in [1] in non-preemptive situations. The most expensive step in Algorithm 1 is line 3, where sort usually requires $O(N\log(N))$. In most cases, intervals are spread evenly within a hyper-period, as such line 7 usually breaks early, and the two for-loop only requires $O(N)$. At the worst case where all the intervals are overlapped with each other, the overall complexity would be $O(N^2)$, which is usually rare.

Algorithm 1: Interval overlapping test

Input: start time vector **s** of all task instances, task sets $\{\tau_i\}$
Output: Interval overlapping error for $\{\tau_i\}$

```
1  OverlapError = 0
2  IntVec = CreateIntervalVector(s, τ_i)
   // Sort based on the start time
3  Sort(IntVec)
4  for I_i in IntVec do
5      EndTime = I_i.start + I_i.length
6      for I_j after I_i do
7          if I_j.start ≥ EndTime then
8              break;
9          end
10         OverlapError+ = Overlap(I_i, I_j)
11     end
12 end
13 return OverlapError
```

2) Interval overlapping test: The following lemmas and theorem give a formal proof that Algorithm 1 always returns the same test result as constraint 8. In non-preemptive situations, we consider two schedulability tests given by demand bound function (DBF test) [1] and interval overlapping function (IO test), respectively. For notation simplicity, we use $[s_i, f_i]$ to denote any task instance's execution interval, the duration of which is c_i, the execution time of the task. Before presenting the proof, we first describe DBF test (from [1]) and IO test as follows:

Definition V.1 (DBF test)**.** *For all intervals $[s_i, f_j]$ such that both jobs i and j are assigned to the same processor under consideration, the task set is schedulable on this processor if*

$$\sum_k c_{ijk} \leq f_j - s_i \tag{14}$$

$$c_{ijk} \geq \begin{cases} c_k, & \text{if } s_i \leq s_k \text{ AND } f_k \leq f_j \\ 0, & \text{otherwise} \end{cases} \tag{15}$$

Definition V.2 (IO test)**.** *We consider any two job instance intervals $[s_i, f_i]$ and $[s_j, f_j]$ where $f_j \geq s_i$ and both job i and j are assigned to the same processor under consideration. The task set is schedulable on this processor if*

$$\forall i, j : f_j - s_i \geq c_i + c_j \tag{16}$$

Lemma 1. *If IO test returns false, then DBF test returns false.*

Proof. If IO test returns false, that means, there exists two different task execution intervals i, j (possibly belong to different tasks):

$$f_j - s_i < c_j + c_i \tag{17}$$

In that case, DBF test will return false because

$$f_j - s_i \leq c_j + c_i \leq \sum_k c_{ijk} \tag{18}$$

■

Lemma 2. *If IO test returns true, then DBF test returns true.*

Proof. Since tasks are non-preemptive, we have

$$\forall p, f_p = s_p + c_p \tag{19}$$

If IO test returns true, that means,

$$\forall i, j : f_j - s_i \geq c_i + c_j \tag{20}$$

In that case, if DBF test returns false, then there must be an interval: $[s_i, f_j]$, such that

$$f_j - s_i \leq \sum_k c_{ijk} \tag{21}$$

Without loss of generality, we assume there are m task instances that contribute into the right hand side of Equation 21. For all of these m task instances, which we label as $0, ..., p, ..., m-1$ following ascending order on its start time, then we have:

$$\sum_k c_{ijk} = \sum_{p=0}^{m-1} c_p \tag{22}$$

Since IO test returns true, we must have

$$f_{p+1} - s_p \geq c_p + c_{p+1} \tag{23}$$

or equivalently,

$$s_{p+1} - s_p \geq c_p \tag{24}$$

Adding all these m tasks together, we have

$$s_{m-1} - s_0 \geq \sum_{p=0}^{m-1} c_p \tag{25}$$

or equivalently,

$$f_{m-1} - s_0 \geq \sum_k c_{ijk} \tag{26}$$

Since this is in contradiction with Equation 21, we know the lemma is true. ■

Theorem 1. *In non-preemptive situations, schedulability test given by demand bound function (DBF test) always returns the same results as schedulability test given by interval overlapping function (IO test).*

Proof. This conclusion is easily derived from Lemma 2 and Lemma 1. ■

B. Optimization problem formulation

1) Feasibility problem: In this paper, we decide to formulate the scheduling problem as an optimization problem. Inspired from [1], the variables $\mathbf{s}$ are the start time of all the node instances in the DAG graph $\mathcal{G}$ within a hyper-period, and the objective is to find a feasible solution that satisfies all the requirements specified by Equation eqs. (6) to (13). For simplicity, we denote these constraints together as:

$$f(\mathbf{s}) \leq 0 \tag{27}$$

where $f(x) : R^n \rightarrow R^m$. Such modeling method is very flexible and can easily describe most constraints without pessimistic assumptions. The optimization problem that we are considering is general, in the sense that different constraints can be added or removed freely, including but are not limited to those mentioned in Section IV.

Directly solving this feasibility problem is very difficult: At first, the problem size is very big. Depending on task periods, the optimization problem proposed could have a very large number of variables and constraints. Common methods such as BnB or meta-heuristics methods usually run very slow in this case, and cannot be expected to return a good solution within reasonable time. Secondly, the objective function is binary, which means either feasible or not. In that case, most iterative algorithms will fail because they do not know whether a new update could improve the solution or not.

2) Least-square formulation: Inspired from barrier function [30], we propose to transform the original feasibility problem into an unconstrained optimization problem as follows:

$$\min_{\mathbf{s}} \sum_{m=1}^{M} \text{Barrier}^2(f_i(\mathbf{s})) \tag{28}$$

where the barrier function is defined as

$$\text{Barrier}(x) = \begin{cases} 0, & x \leq 0 \\ g(x), & \text{otherwise} \end{cases} \tag{29}$$

where $g(x)$ is a punishment function, such as

$$g(x) = -f(x) \tag{30}$$

Different from the binary feasibility problem, the least-square formulation 28 is much easier to solve because classical trust-region methods can be applied to solve 28. As illustrated before, these gradient-based methods usually have better performance and faster speed than traditional meta-heuristics [31] or BnB(usually has exponential complexity) in many situations. What's more, problem structures such as sparsity can be utilized to speed up computation even further, which we will discuss in more detail.

Example 1. We show how to formulate the barrier function for *computation resource bounds* constraint according to IO test

in this example. At first, IO test constraint 16 is transformed to negative inequality in the form of constraint 27:

$$f_{DBF}(\mathbf{s}) = \sum_i \sum_j c_i + c_j - (f_j - s_i)|_{f_j - s_i \leq 0} \tag{31}$$

This constraint can then be transformed to objective function with the barrier function as follows:

$$\text{Barrier}(\mathbf{s}) = \begin{cases} 0, & f_{DBF}(\mathbf{s}) \leq 0 \\ -f_{DBF}(\mathbf{s}), & \text{otherwise} \end{cases} \tag{32}$$

Obviously, the function is not differentiable at the schedulability region boundary. We will show how to handle it later.

3) Schedulability analysis: Since the Barrier function always gives a positive error if some constraints are violated, the least-square formulation establishes a *necessary and sufficient* condition for schedulability analysis:

Theorem 2. *A schedule* $\mathbf{s}$ *is schedulable for a given task sets with scheduling requirements specified by constraints 27 if and only if:*

$$\sum_{m=1}^{M} \textit{Barrier}^2(f_i(\boldsymbol{s})) = \boldsymbol{0} \tag{33}$$

Proof. The proof is straightforward. If one schedule $\mathbf{s}$ is schedulable, then all the barrier function will be 0 according to the definition of barrier function in 29. If $\mathbf{s}$ is not schedulable, at least one of the barrier function equals a positive error. Since barrier function cannot return negative error, objective function 28 must be non-zero. Therefore, equation 33 establishes a necessary and sufficient condition for schedulability. ■

C. Gradient-based optimization method

After formulating the least-square problem in 28, we can use gradient-based trust-region methods [31], [32] to solve it, just as shown in the review section 1. However, several issues must be noticed before proceeding:

1) Numerical Jacobian evaluation: Since many constraints are not always differentiable, numerical methods are used to estimate Jacobian matrix at those non-differentiable points:

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(x_1, .., x_j + h, .., x_n) - f_i(x_1, ..., x_j - h, ..., x_n)}{2h} \tag{34}$$

The $h > 0$ parameter above should be *reasonably* small to properly estimate Jacobian matrix. If h is large, then the numerical Jacobian would not be useful for trust-region methods; however, if it is too small, then the numerical Jacobian could become very large and so misleading when x_j is non-differentiable. In our case, we use $h = 0.1$.

Although numerical Jacobian is simple, analytic Jacobian should always be preferred whenever possible because it would save lots of computation cost. Making appropriate use of chain rule could improve speed in this part by an order.

Example 2. We show how to formulate analytic Jacobian for *computation resource bounds* constraint according to IO test in this example. Other constraints' Jacobian matrix are estimated in a similar way.

Similar as evaluating IO test error, we sort all the execution time intervals according to their start time at first. Then we find all the intervals that are possible to overlap from each other. For each of these two pairs of intervals, we use numerical Jacobian to estimate numerical gradient, and then record the results. After adding elimination in the next sections, we will use chain rule to derive partial gradient of true start time vector with respect to eliminated start time vector, and then multiply it together:

$$J_{IO}^{elimination} = J_{IO}^{true} J_{true}^{elimination} \tag{35}$$

2) Sparsity for speed-up: Although the least-square problem could be a very large-scale optimization problem, efficient solutions actually can be achieved by exploiting sparsity. All constraints in $f(\mathbf{s})$ are only related to a few variables instead of n variables of $\mathbf{s}$, which means the Jacobian matrix J evaluated from objective function 28 would be a very sparse matrix. Such sparsity combined with elimination is well exploited in many robotics and optimization problems such as simulated localization and mapping (SLAM) [22], [33], motion planning [32], where thousands or even millions of variables are optimized together with fast speed.

To be more specific, advantages in computation cost are shown in the following aspects:

- By only considering non-zero elements, sparse matrix multiplication is usually much faster than its dense version which usually requires $O(N^3)$
- One of the major computation cost in LM lies in solving backward substitution for the Hessian matrix $J^T J$. With proper ordering heuristic, sparse Cholesky or QR decomposition can not only speed up backward substitution, but also reduce fill-in afterwards, and so further speed-up [22].
- Much faster Jacobian evaluation. By identifying non-zero elements at first, only a small portion of elements have to be evaluated in the Jacobian matrix, which achieves computation saving in one or two order.

As such, although there may be thousands or tens of thousands variables and constraints in problem 28, almost all the operations during optimization process only have $O(N)$ or $O(N^2)$ complexity.

3) Vanishing gradient problem: Vanishing gradient problem is a common problem met in the machine learning area. We borrow the same name there because some functions such as IO test constraints suffer from similar issues. In other words, we observe that during optimization, some points have 0 gradient but with non-zero error. An example situation is shown in Fig. 2. To handle this issue, a simple idea would be increasing the granularity, i.e., h in Equation 34, until the gradient is not zero if the interval overlap error is not 0. However, this idea requires using numerical Jacobian instead of analytic Jacobian, and is very expensive in large-scale systems. More effective ideas such as random walk will be exploited in the near future.

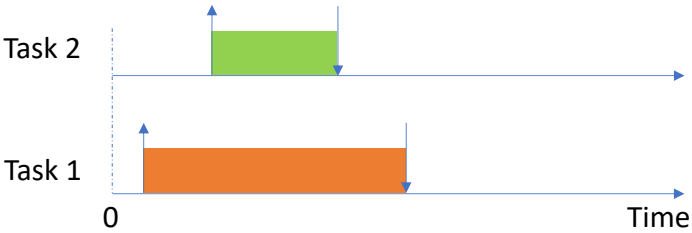

Figure 2: When task 1's *interval*(see section V-A) fully covers task 2's interval, gradient for both the start time of task 1 and task 2 would become 0, even though the system is not schedulable.

4) Getting stuck at a non-local optimal point: Two definitions are provided for the ease of presentation:

Definition V.3 (Local optimal point(LOP))**.** *A point x_0 is a local optimal point for unconstrained optimization $\min_x f(x)$ if*

$$\forall \Delta \in \{R^n | \ |\Delta| = 1\}, \ \delta \to 0 : \\ f(x_0) \leq f(x_0 + \Delta\delta) \tag{36}$$

where Δ indicates an updating direction, and δ is step size.

Definition V.4 (Pseudo-local optimal point (PSOP))**.** *A point x_0 for unconstrained optimization $\min_x f(x)$ is called pseudo-local optimal point if*

- *It is judged as a local optimal point from Jacobian J, Hessian H or their variants:*

$$\delta \to 0 : f(x_0) < f(x_0 + \Delta(J, H)) \tag{37}$$

  where Δ is given by the common gradient based methods such as steepest descent, Gauss-Newton, Levenberg-Marquardt (LM) [2], [26], Dogleg(DL) [27], etc.
- *It is not a local optimal point, i.e.,*

$$\exists \Delta \in \{R^n | \ |\Delta| = 1\}, \ \delta \to 0 : \\ f(x_0 + \Delta\delta) \leq f(x_0) \tag{38}$$

In our experience, we find that it is very common that the trust-region optimizer return a PSOP rather than LOP, which is mostly because the objective function is discrete in some area. To solve this issue, we propose a new algorithm named *elimination forest* in the next section.

D. Building and managing elimination forest

1) Leaving PSOC by elimination: One of the reasons that the optimization algorithm is stuck at a pseudo-local optimal point is because gradient cannot guide the optimization well at points where the Jacobian is not continuous. For example, consider the following situation that the computation resource bound constraint could become almost 0 but still feasible:

$$f_i(\mathbf{x}) = \delta < 0 \tag{39}$$

In this case, $f_i(\mathbf{x})$ is very close to its schedulability boundary and there is no much space to optimize anymore. However, numerical gradient is not well defined at this point and cannot lead the optimizer to stop optimizing toward this direction. As such, to prevent it from interfering other variables continuing optimization, it would be useful if we manually stop optimizing $f_i(\mathbf{x})$ and keep $f_i(\mathbf{x})$ at this almost-zero point. This intuition promotes us to consider objective function elimination. To lock $f_i(\mathbf{x})$ at this point, we need to add one more constraint $f_i(\mathbf{x}) = \delta$ back to the system. For the convenience of optimization, we do this by transforming Equation 39 to

$$x_j = f_i^{-1}(x_0, ..., x_{i-1}, x_i, ..., x_{n-1}, \delta) \tag{40}$$

to "eliminate" x_j.

To summarize, when we need to perform elimination, we go through each component of objective function $f(x)$ to find one dimension i:

$$f_i(x) = \delta < 0 \tag{41}$$

where δ is smaller than a threshold. If some variables related to dimension i have not been eliminated, then we will choose one variable and perform elimination, i.e., use other variables to replace it following Equation 41. In our current implementation, we only included IO test to build elimination forest. However, other kinds of constraints, if only they will possibly limit performance by getting stuck at PSOC, should be included into elimination forest.

Example 3. In case of resource bound constraint, although the overall error is a scalar for the convenience of programming, we need to check for elimination condition for each pair of intervals with non-zero overlapping error. As such, following Algorithm 1, we create and sort intervals. After it, we evaluate numerical Jacobian following Equation 34 for each pair of intervals with non-zero overlapping error, or

$$c_i + c_j - (f_j - s_i) \leq h \tag{42}$$

where h is a small positive number as in Equation 34 that controls granularity of numerical Jacobian. The overall worst-case complexity of detecting and adding elimination is $O(N^2)$.

2) Building elimination forest: As the iteration loop continues, more and more variables may be eliminated and could bring confliction in constraints. Inspired from Bayes tree proposed by Kaess *et al.* [34], we propose elimination forest, an efficient algorithm for both managing elimination relationship and avoiding constraints conflict.

Pseudo-code for creating and managing elimination forest is given in Algorithm 2. First we create a node (root for a tree) for each variables. Each time a variable is eliminated, we add dependency edges from the eliminated variables to the dependency variables. Since all the nodes in a tree have fixed relative value (given by equation 41), adding a new dependency may possibly make some nodes in those two trees violate some constraints. In that case, confliction check must be performed to all the nodes in the two related trees. An example is given in Fig. 3. Please notice that although Fig. 3 only shows elimination trees for two-variable constraints, the algorithm is applicable to constraints with more variables.

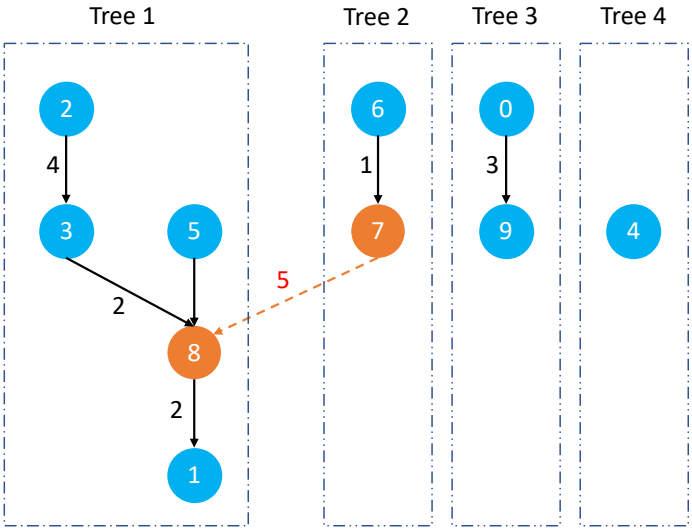

Figure 3: Managing elimination forest: After adding an edge from node 7 to node 8, all the nodes in tree 1 and tree 2 depend on node 1. This may indirectly cause some nodes to always violate some constraints. For example, adding the new edge from 7 to 8 will make node 6 and node 2 both start at the same time because the relative distance to node 8 are all 6, and so the system would always become unschedulable in single-core platform. As such, confliction check must be performed to prevent indirect confliction before adding edges.

Building elimination forest is also helpful in improving algorithm efficiency because all the nodes inside an elimination tree are already checked to be confliction-free. When an edge is considered to be added between two nodes, we only need to check confliction for nodes between the two trees. It will save more time and effort when performing optimization with respect to a large number of variables.

E. Initial solution estimation

Although our scheduling optimization problem 28 is an unconstrained optimization problem, which means any variable estimation can serve as an valid initialization, in reality, it is highly suggested to provide a reasonable initialization method, preferably from an available scheduling algorithm. Totally random initialization usually cannot give very good performance in our experience.

Finding a good initialization method is actually very simple, because almost all available scheduling algorithms can serve as a good initialization. Such flexible initialization policy makes NLP compatible with most available scheduling algorithms to achieve further performance boost. If only a scheduling algorithm could satisfy parts of requirements, then it can be used as initialization and the proposed algorithm is expected to give a even better schedule. For example, classical Rate Monotonic(RM) is used in our experiments.

Algorithm 2: Managing elimination forest inside one iteration

Input: variables **s** after performing unconstrained optimization, elimination forest G
Output: elimination forest G

if *G is empty* **then**
  Add each s as independent root nodes to G
end
// iterate over objective function
for $(i = 0;\ i < m;\ i = i + 1)$ **do**
  if $\theta \leq f_i(\boldsymbol{x}) \leq 0$ **then**
    $s_0 =$ Select eliminated variable
    $S =$ Select dependency variables
    Trees = $\{\}$
    for $(s_j \in \{s_0, S\})$ **do**
      Trees.**add**(**ExtractTree**(G, s_j))
    end
    if ***CheckNoConfliction***$(Trees)$ **then**
      G. **AddEdge**(x_0, X)
    end
  end
end
return G

VI. Experiment results

A. Task sets generation

The scheduling algorithm is tested on simulated random task sets. Task dependency is generated randomly following He *et al.* [35]'s experiments. After creating vertices in DAG, a parallel factor is selected. Then we visit each pair of vertices and generate a random number, if it is smaller than the parallel factor, then a dependency edge is added. As such, the larger the parallel factor, the more edges will there be in the DAG. The processor that a task is assigned is generated randomly from the range of available processor.

Two different task sets of different scale are generated to test performance. One task set comes from an automotive benchmarks in the industry [36], where periods are randomly selected in the range $[1, 2, 5, 10, 20, 50, 100, 200, 1000]$. Task sets generated in this way usually have several thousands of variables to optimize. In the second task sets, we choose task periods from a smaller range $[100, 200, 300, 400, 500, 600, 800, 1000, 1200]$. These task sets usually have several hundreds of variables to optimize. Although we do not test larger task sets with tens of thousands variables, the proposed algorithm is expected to work well within longer but acceptable time because overall complexity is polynomial.

Each task's execution time is generated by Uunifast [37], while the deadline could be implicit or random from the task's execution time to its period. In each of the following, experiments, 1000 random task sets are used for evaluation. Other settings that will change in different experiments will be described later.

A schedule is accepted if the sum of final error of all the constraints within a hyperperiod mentioned in Section IV is smaller than a small threshold.

Finally, we want to mention that although the proposed algorithm shows a low accept rate in some situations, that is mostly because many task sets are not schedulable in that case. The first task set is consisted of tasks with big difference between task set periods. If dependency edges are generated totally randomly, then it is highly likely that some tasks with small periods depend on tasks with large large execution time, which means the task with small periods will almost always miss its deadline. Therefore, we suggest readers to focus on performance improvement achieved rather than absolute accept rate reported.

B. Automotive task sets

Experiments in this section are conducted on the automotive task sets. We may control the range that random period parameters can generate from for faster evaluation. However, this change mostly only influences speed rather than performance, and the performance advantages that NLP can achieve over initial algorithms remain the same.

1) Constrained deadline: In this experiment, each task's deadline is generated randomly from its execution time to its period. Two cores are used to process each task sets, and the processor assignment is totally random. Sensor fusion bound is set as 50, and sensor freshness bound is set as 100000(not used, actually). Parallel factor is 0.2, which means in average there are 2 edges in each DAG. Schedule accept threshold is 10. Overall, the experiment settings are relatively easy to satisfy by many algorithms. To perform experiments faster, we exclude period 1 from the random period selection range in this section. The results of both the proposed algorithm (NLP) and initialization algorithm are shown in Fig. 4. It can be seen an obvious improvement under different system utilization even though the potential improvement space is not very big.

2) Multi-core platforms: We also tested proposed algorithm on multi-core systems. All the task sets are using implicit deadline with different per-core utilization and processing cores. Besides, sensor freshness and sensor fusion bound are both set as 50, schedule accept threshold is 1.0. For faster evaluation, we only select periods from $[2, 5, 10, 20, 50, 100, 200]$. Overall, this is a very difficult task sets for scheduling comparing with the first experiment. The results are reported in Fig. 5, where accept rate before and after optimization is drawn by dashed and solid lines, respectively. As task sets utilization increases, the NLP optimization improves accept rate by 5 times in average, 10 times at the maximum.

3) Run-time speed: We report the run-time speed of the proposed algorithm in this section. The task sets are executed on two cores, and all the task sets have implicit deadline. The task periods are generated from all the possible ranges in $[1, 2, 5, 10, 20, 50, 100, 200, 1000]$, which means there are several thousands of variables to optimize in each task set. Sensor fusion bound is set as 500, while sensor freshness bound is set as 100000(no use). Schedule accept threshold is 10. Run-time speed and accept rate are reported in Fig. 6 and 7, respectively. A very good quadratic curve fitting is shown in the Fig. 6 by dashed lines, which verifies previous $O(N^2)$ complexity analysis. To avoid possible confusion, it should be highlighted that the complexity is more related to the number of job instances within a hyperperiod rather than the number of tasks in a task set. However, Fig. 6 shows statistical results, which makes the run-time also follows a good quadratic relationship with the number of tasks in each task set.

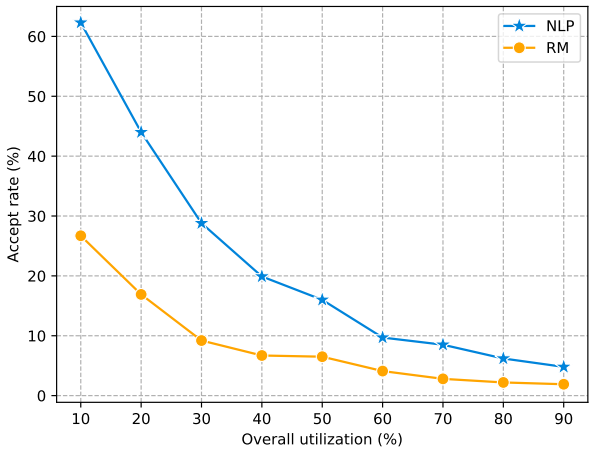

Figure 4: Comparison of initial (RM) and optimization algorithms on automotive task sets with random deadline, two-core system. The performance is improved by 100% in average.

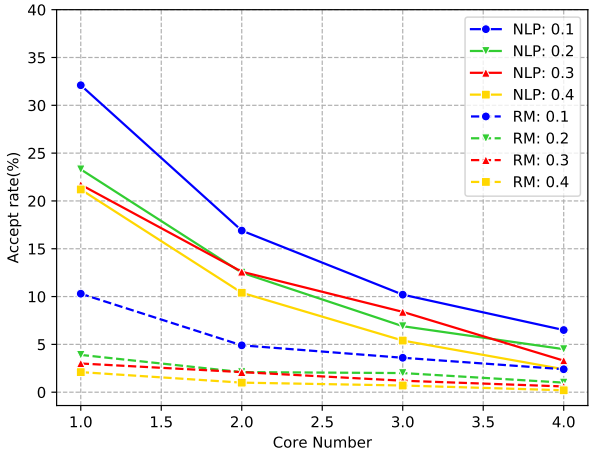

Figure 5: Comparison of accept rate of initial (RM) and optimization algorithms. Results with same utilization are drawn using the same color. In average, accept rate is improved by 5 times after optimization.

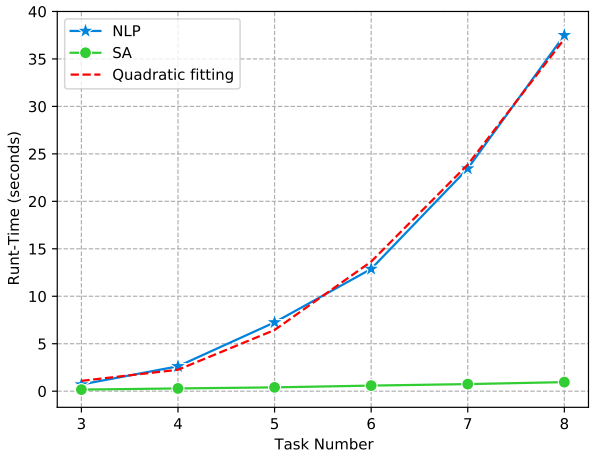

Figure 6: Run-time speed in automotive task sets. The proposed optimization algorithm has a $O(N^2)$ complexity as shown by the quadratic fitting (dashed curve).

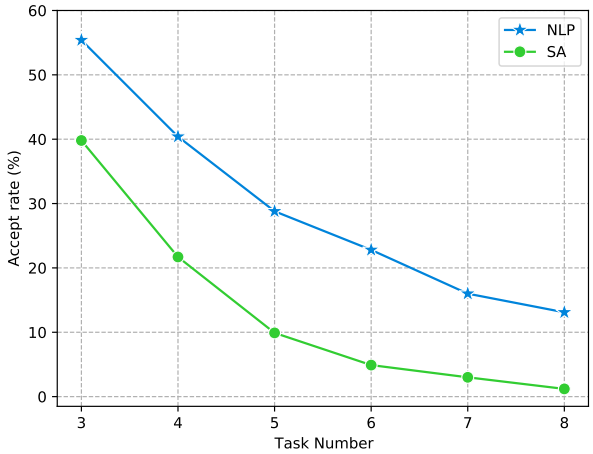

Figure 7: Comparison of performance with simulated annealing, both algorithms are initialized with RM. 40% to 1000% improvements in accept rate is shown as task number increases.

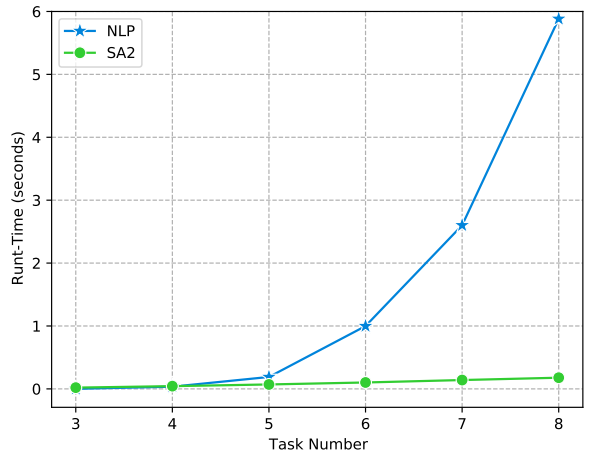

Figure 8: Comparison of performance with simulated annealing, both algorithms are initialized with RM. Since there are fewer variables to optimize, the run-time speed is faster than automotive task sets.

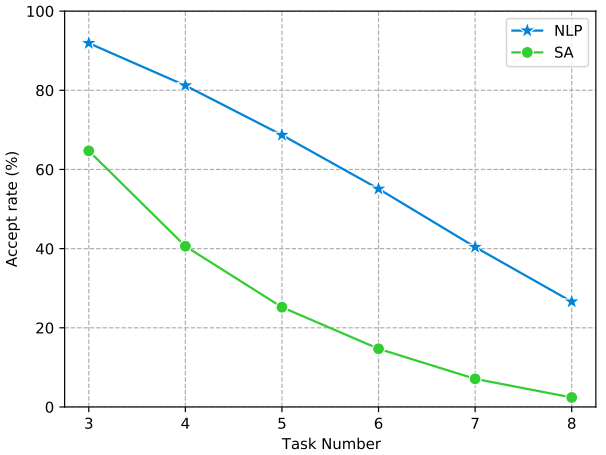

Figure 9: Comparison of performance with simulated annealing, both algorithms are initialized with RM. An obvious advantage in accept rate is shown with larger task numbers.

C. Simulated task sets

In this section, we test algorithm performance on a smaller task sets where several hundreds of variables are being optimized. Most task set settings remain the same as above except that periods are limited to a smaller range $[100, 200, 300, 400, 500, 600, 800, 1000, 1200]$. Still, "small" there means the number of variables rather than the absolute value. All the experiments within one table have the same initialization(RM), and all the experiments are evaluated on the same task sets with same scheduling requirements. The results are summarized in Table I and Fig. 8, 9. In Table I, the first row reports initialization method's performance, second row reports baseline optimization method simulated annealing (SA), third row are the proposed NLP algorithm. It can be seen that the proposed algorithm successfully reduce initial error by 70%. Considering that a large portion of task sets are not schedulable, which means the error cannot be reduced to 0, the proposed algorithm would also be very good in serving as a soft real-time scheduler.

D. Summary on experiments

Several conclusions can be drawn from these experiments:

- The proposed algorithm improves around 50% to 1000% acceptance rate in the experiments, which proves the validity and effectiveness of the algorithm.

Table I: More precise data on optimization performance

Algorithm	Accept rate (Error<1)	Accept rate (Error<0.1)	Average Initial Error (time units)	Average Optimized Error (time units)	Average time (seconds)
RM_Initialize	25.9%	25.4%	410.861	-	1e-4
SA_RM_Optimize	25.9%	25.4%	410.861	168.94	0.072
NLP_RM_Optimize	**66.5%**	63.2%	410.861	122.181	0.218

- The proposed algorithm can easily schedule systems with a large range of constraints and requirements.
- Although relying on a reasonable initialization method, the proposed algorithm does not require a very good initialization, which is critical in practice. Furthermore, the proposed algorithm can usually utilize a better initialization method for better performance.
- It may take some time to optimize a system with a very large number of variables. However, the $O(N^2)$ complexity analysis makes sure that the analysis can be finished within reasonable amount of time.

VII. Conclusions and Future Work

The scheduling problem which combines so many complicated constraints is actually very difficult, and there is not much related work to consult. Since for many real-time system models the exact schedulability analysis of real-time systems is NP-hard (such as that of fixed priority scheduling for Liu-Layland task model [38]), it is very likely that our problem is an NP-hard problem if we want to find the global optimal solution. However, by exploiting problem-specific structure and classical NLP algorithms, the proposed algorithm provides a systematic way to solve this scheduling problem very efficiently and effectively. There are also limitations in the proposed method, for example, it is not able to solve vanish gradient issue effectively but can only rely on very heuristic method. As such, that would be one of our main focus in the future. Besides, we also hope to find ways to improve the algorithm complexity even further so that the proposed algorithm can be applied in online situations.

References

[1] S. Baruah, "Scheduling dags when processor assignments are specified," *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, 2020.

[2] K. Levenberg, "A method for the solution of certain non – linear problems in least squares," *Quarterly of Applied Mathematics*, vol. 2, pp. 164–168, 1944.

[3] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," *2012 IEEE 33rd Real-Time Systems Symposium*, pp. 63–72, 2012.

[4] S. Baruah, M. Bertogna, and G. Buttazzo, "Multiprocessor scheduling for real-time systems," in *Embedded Systems*, 2015.

[5] S. Baruah, L. E. Rosier, and R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, pp. 301–324, 2005.

[6] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, "Feasibility analysis in the sporadic dag task model," *2013 25th Euromicro Conference on Real-Time Systems*, pp. 225–233, 2013.

[7] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet, "Global edf scheduling of directed acyclic graphs on multiprocessor systems," in *RTNS '13*, 2013.

[8] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," *2015 27th Euromicro Conference on Real-Time Systems*, pp. 211–221, 2015.

[9] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[10] R. M. Pathan, P. Voudouris, and P. Stenström, "Scheduling parallel real-time recurrent tasks on multicore platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 915–928, 2018.

[11] J. C. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic dag tasks for global fp scheduling," *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017.

[12] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill, "Parallel real-time scheduling of dags," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, pp. 3242–3252, 2014.

[13] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones, "Response-time analysis of dag tasks under fixed priority scheduling with limited preemptions," *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1066–1071, 2016.

[14] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. D. Gill, and A. Saifullah, "Analysis of federated and global scheduling for parallel real-time tasks," *2014 26th Euromicro Conference on Real-Time Systems*, pp. 85–96, 2014.

[15] S. Baruah, "The federated scheduling of constrained-deadline sporadic dag task systems," *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1323–1328, 2015.

[16] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho, "A multi-dag model for real-time parallel applications with conditional execution," *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015.

[17] S. Baruah, V. Bonifaci, and A. Marchetti-Spaccamela, "The global edf scheduling of systems of conditional sporadic dag tasks," *2015 27th Euromicro Conference on Real-Time Systems*, pp. 222–231, 2015.

[18] J. Abdullah, G. Dai, and W. Yi, "Worst-case cause-effect reaction latency in systems with non-blocking communication," *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1625–1630, 2019.

[19] M. Günzel, K.-H. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J.-J. Chen, "Timing analysis of asynchronized distributed cause-effect chains," *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 40–52, 2021.

[20] S. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, 1990.

[21] K. Jeffay and D. L. Stone, "Accounting for interrupt handling costs in dynamic priority task systems," *1993 Proceedings Real-Time Systems Symposium*, pp. 212–221, 1993.

[22] F. Dellaert and M. Kaess, "Factor graphs for robot perception," *Found. Trends Robotics*, vol. 6, pp. 1–139, 2017.

[23] K. Tindell, A. Burns, and A. Wellings, "Allocating hard real-time tasks: An np-hard problem made easy," *Real-Time Systems*, vol. 4, pp. 145–165, 2004.

[24] A. Hamann, M. Jersak, K. Richter, and R. Ernst, "Design space exploration and system optimization with symta/s - symbolic timing analysis for systems," *25th IEEE International Real-Time Systems Symposium*, pp. 469–478, 2004.

[25] Y. Zhao, R. Zhou, and H. Zeng, "An optimization framework for real-time systems with sustainable schedulability analysis," *2020 IEEE Real-Time Systems Symposium (RTSS)*, pp. 333–344, 2020.

[26] D. Marquardt, "An algorithm for least-squares estimation of nonlinear

parameters," *Journal of The Society for Industrial and Applied Mathematics*, vol. 11, pp. 431–441, 1963.
[27] M. POWELL, "A new algorithm for unconstrained optimization," in *Nonlinear Programming* (J. Rosen, O. Mangasarian, and K. Ritter, eds.), pp. 31–65, Academic Press, 1970.
[28] PerceptIn, "2021 rtss industry challenge." http://2021.rtss.org/industry-session/, 2021.
[29] M. Verucchi, *A comprehensive analysis of DAG tasks: solutions for modern real-time embedded systems*. PhD thesis, Department of Physiscs, Informatics and Mathematics, University OF Modena And Reggio Emilia, 2020.
[30] S. P. Boyd and L. Vandenberghe, "Convex optimization," *IEEE Transactions on Automatic Control*, vol. 51, pp. 1859–1859, 2006.
[31] S. Wang, J. Chen, X. Deng, S. Hutchinson, and F. Dellaert, "Robot calligraphy using pseudospectral optimal control in conjunction with a novel dynamic brush model," *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6696–6703, 2020.
[32] M. Mukadam, J. Dong, X. Yan, F. Dellaert, and B. Boots, "Continuous-time gaussian process motion planning via probabilistic inference," *The International Journal of Robotics Research*, vol. 37, pp. 1319 – 1340, 2018.
[33] F. Dellaert and M. Kaess, "Square root sam: Simultaneous localization and mapping via square root information smoothing," *The International Journal of Robotics Research*, vol. 25, pp. 1181 – 1203, 2006.
[34] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert, "isam2: Incremental smoothing and mapping using the bayes tree," *The International Journal of Robotics Research*, vol. 31, pp. 216 – 235, 2012.
[35] Q. He, M. Lv, and N. Guan, "Response time bounds for dag tasks with arbitrary intra-task priority assignment," in *ECRTS*, 2021.
[36] A. H. Simon Kramer, Dirk Ziegenbein, "Real world automotive benchmarks for free," 2015.
[37] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, pp. 129–154, 2005.
[38] P. Ekberg and W. Yi, "Fixed-priority schedulability of sporadic tasks on uniprocessors is np-hard," in *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pp. 139–146, IEEE Computer Society, 2017.