Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks

Sayra Ranjha[1,2], Geoffrey Nelissen[1], Mitra Nasri[1]
[1] *Eindhoven University of Technology, The Netherlands* [2] *Delft University of Technology, The Netherlands*

Abstract—**The temporal correctness of safety-critical systems is typically guaranteed via a response-time analysis (RTA). However, as systems become complex (e.g., parallel tasks running on a multicore platform), most existing RTAs either become pessimistic or do not scale well. To make a trade-off between accuracy and scalability, recently, a new reachability-based RTA, called *schedule-abstraction graph* (SAG), has been proposed. The analysis is at least three orders of magnitude faster than other exact RTAs based on UPPAAL.**

One fundamental limitation of the SAG analysis is that it suffers from state-space explosion when there are large uncertainties in the timing parameters of the input jobs, which may impede its applicability to some industrial use cases. In this paper, we improve the scalability of the SAG analysis by introducing *partial-order reduction* (POR) rules that avoid combinatorial exploration of all possible scheduling decisions. An empirical evaluation shows that our solution is able to reduce the runtime by *five orders of magnitude* and the number of explored states by 98%, at a negligible cost of an over-estimation of 0.1% on the tasks' worst-case response-time (WCRT). We applied our solution on an automotive case study showing that it is able to scale to realistic systems made of hundreds of tasks for which the original analysis fails to finish.

I. Introduction

Guaranteeing temporal correctness of safety-critical real-time systems is typically done via a response-time analysis (RTA) whose goal is to determine the *worst-case response time* (WCRT) of a set of input jobs scheduled by a given scheduling policy on a computing resource. A real-time system is said to be *schedulable* if the WCRT of each job is smaller than its deadline. However, most variations of the RTA problem for periodic tasks scheduled by a *job-level fixed-priority* (JLFP) policy such as *fixed-priority* (FP) or *earliest-deadline first* (EDF) are NP-hard even for a uniprocessor platform [1].

Related work. Known *exact* WCRT analyses fall into two general categories: (i) fixed-point iteration-based analyses [2,3] that have pseudo-polynomial time complexity and (ii) *reachability-based analysis* (RBA) [4]–[10]. Fixed-point iteration-based analyses are typically faster than RBAs, however, they are exact only in special cases, e.g., when analyzing sporadic (or periodic) tasks scheduled by the FP or EDF scheduling policies on a uniprocessor platform [2,3,11,12]. RBAs, on the other hand, are often exact in more general cases, e.g., preemptive [4] or non-preemptive [6] periodic and sporadic tasks scheduled by global FP policies. However, a majority of them are known to have very poor scalability w.r.t. the number of tasks and processors or period values [4]–[6].

Recently, a new reachability-based RTA, called *schedule-abstraction graph* (SAG), has been proposed [7]–[10,13]. It surpasses the scalability limitations of the timed-automata based RBAs by at least three orders of magnitude [6,9]. However, as we will show in Sec. III, this analysis still suffers from state-space explosion when there are large uncertainties in the timing parameters of the input tasks or jobs, e.g., large release jitter or execution-time variations.

Schedule-abstraction graph. SAG explores the space of possible decisions that a JLFP scheduler can take when dispatching a set of jobs on processing resources. This decision space is explored by building a graph whose vertices represent the *state* of the resource (e.g., processor) *after the execution of a set of jobs* and whose edges represent possible scheduling decisions that evolve the system states. SAG has been designed for non-preemptive jobs [7]–[10], hence, a scheduling decision is to determine "*a next job that can possibly be dispatched*" after a system state. An example job set along with its schedule are provided in Figs. 1(a) and 1(b), respectively. The graph made by the SAG for this job set is shown in Fig. 1(c). While building the graph, the response time of each job dispatched on an edge is tracked. Hence, when the graph is fully constructed, the method outputs the smallest and largest response time of each job in all scenarios (edges) that involve that job.

To defer the state-space explosion, Nasri et al. [7] have introduced two main techniques: (i) powerful *interval-based abstractions* to aggregate similar system states (or equivalently, schedules that have a similar impact on the final system state), and (ii) *state-merging rules* to combine system states whose future can be explored together. These techniques allowed their solution to be at least 3000 times faster than other exact RBAs based on generic formal verification tools such as UPPAAL [6], and to scale to large system sizes.

Current limitation of SAG. Despite the current success in scalability, the schedule-abstraction-based analysis still faces one big fundamental limitation: *each edge can only account for a single scheduling decision* (i.e., dispatching of one job) [7]–[10]. As a result, as soon as there are large uncertainties in the release time or execution time of the jobs in the input job set, the number of states generated by the SAG grows exponentially because the analysis will try to explore all (valid) combinations of ordering between jobs. This may impede applicability of the SAG analysis to large industrial use cases.

This paper. The goal of our work is to improve the scalability of the schedule-abstraction-based analysis by introducing *partial-order reduction* (POR) rules that allow combining multiple scheduling decisions on one edge and hence avoiding combinatorial exploration of all possible orderings between jobs in cases where there are large uncertainties.

As our goal is to demonstrate how to apply POR on SAG-based analyses, in this work we will focus on a simpler (yet NP-hard) schedulability analysis problem, i.e., analyzing a set of non-preemptive jobs (or periodic tasks) scheduled by a work-conserving JLFP scheduling policy on a uniprocessor platform. A recent survey on industrial real-time systems shows that more than 80% of real-time systems have periodic activities and about 40% of them include single-core platforms [14]. Its findings show that this problem is not only relevant now but also in the next ten years as indicated by over 30% of industrial practitioners that filled-in the survey.

We show that our solution is able to reduce the runtime of the SAG analysis by *five orders of magnitude* and the number of explored states by 98%, while remaining an *exact* schedulability analysis. This achievement comes at a negligible cost of an over-estimation of only 0.1% on the WCRT. Furthermore, we applied our solution to a large case study from an automotive use case with more than 710 runnables and tens of thousands of jobs per hyperperiod. This shows that our solution is suitable to be used as an exact schedulability analysis in industrial-grade design-space exploration tools.

II. System model and assumptions

A. Job and system model

We consider the problem of scheduling a finite set of non-preemptive jobs $\mathcal{J}$ on a uniprocessor platform. A job $J_i = ([r_i^{min}, r_i^{max}], [C_i^{min}, C_i^{max}], d_i, p_i)$ is characterized by its earliest release time r_i^{min} (a.k.a. arrival time [2]), latest release time r_i^{max}, best-case execution time (BCET) C_i^{min}, worst-case execution time (WCET) C_i^{max}, absolute deadline d_i, and priority p_i. We assume that the job timing parameters are integer multiples of the system clock.

Job J_i non-deterministically releases at a time instant $r_i \in [r_i^{min}, r_i^{max}]$ and executes for an *a priori* unknown amount of time $C_i \in [C_i^{min}, C_i^{max}]$. Because of this uncertainty, we say that J_i is *possibly released* at time t if $r_i^{min} \leq t < r_i^{max}$ and *certainly released* if $t \geq r_i^{max}$. We say a job is *ready* at time t if it is released but did not start executing before t.

Because we assume that every job J_i is non-preemptive, a job J_i that starts executing *at* time t finishes its execution *by* time $t + C_i$, continuously occupying the processor during the interval $[t, t + C_i)$. We denote the finish time of J_i by f_i. Once J_i finishes its execution by f_i, the processor becomes available again and the next job may start. The response time of J_i is the length of the interval between its earliest release time r_i^{min} and its finish time f_i, i.e, $f_i - r_i^{min}$. We assume that the deadline of a job is set based on the arrival time and hence is not affected by the release jitter.

Priorities are integer numbers. A smaller value indicates a higher priority, namely, if $p_i < p_j$, then the job J_i has a higher priority than the job J_j. Priority ties are broken arbitrarily but consistently, i.e., we assume that the "$<$" operator implicitly uses this tie-breaking rule. The priority of a job is assigned by a job-level fixed-priority (JLFP) scheduling policy. For example, under EDF, a job's priority is its absolute deadline.

We use $\langle\ \rangle$ to refer to an ordered set (or a sequence) and $\{\ \}$ to refer to a non-ordered set. Neither contains repeated items. We use $\min_\infty\{X\}$ and $\max_0\{X\}$ over a set of positive integers X to indicate that the minimum of an empty set is infinity and the maximum of an empty set is 0.

B. Scheduler model

We consider all non-preemptive job-level fixed-priority (JLFP) scheduling algorithms. Despite the original schedule-abstraction graph from Nasri et al. [7] supporting both work-conserving and non-work-conserving scheduling policies, we only focus on work-conserving policies in this paper, i.e., policies that do not leave the processor idle as long as there is a ready job in the system. Like the original analysis, we solely focus on schedulers that are priority-driven and deterministic, i.e., schedulers that only schedule a job if it is the highest-priority ready job in the system and always produce the same schedule for a given execution scenario, where an execution scenario is defined as a mapping of the jobs to release times and execution times as follows.

Definition 1. (from [7]) An execution scenario $\gamma = (C, R)$ for a set of jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_m\}$ is a sequence of execution times $C = \langle C_1, C_2, \ldots, C_m \rangle$ and release times $R = \langle r_1, r_2, \ldots, r_m \rangle$ such that, $\forall J_i \in \mathcal{J}$, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$

We consider a set of jobs $\mathcal{J}$ to be *schedulable* under a given scheduling policy A if there exists no execution scenario of $\mathcal{J}$ that results in a deadline miss when scheduled by A.

III. Motivation

When a given job set has timing uncertainties, e.g., due to release jitter or execution time variation, the exact release or execution times of the jobs are not known. Since an online JLFP scheduler takes its decisions by looking at the *released jobs* in the ready queue when the processor is available, multiple schedules might be generated depending on the release order of jobs and the completion time of already running jobs (called *execution scenarios*).

The current SAG analysis [7] processes one scheduling decision at a time as can be seen on the labels of the edges in Fig. 1(c). It starts from an idle system (state v_1) and looks for all *next* possible scheduling decisions. Here, J_1 has no release jitter. As the first and only job in the ready queue in v_1 and, under a work-conserving scheduling policy, J_1 will always be dispatched before other jobs (hence, there is only one edge from v_1 to v_2). However, due to the non-deterministic execution time of J_1, the processor may become available for other jobs at any time from 7 to 13 (note that 7 is the BCET and 13 the WCET of J_1). The SAG abstracts these uncertainties in an *uncertainty interval* $[A_1^{min}(v_2), A_1^{max}(v_2)] = [7, 13]$ on state v_2 which means that the processor may *possibly* be available for other jobs at time 7 and will *certainly* be available at time 13.

Representing a system state using an uncertainty interval allows SAG to fairly reduce the number of states needed to

(a)

J_i	release time earliest	latest	BCET	WCET	deadline	priority
J_1	0	0	7	13	20	1 (high)
J_2	6	9	1	2	20	2
J_3	7	10	1	2	20	3
J_4	18	20	1	3	25	4
J_5	5	9	1	2	20	5 (low)

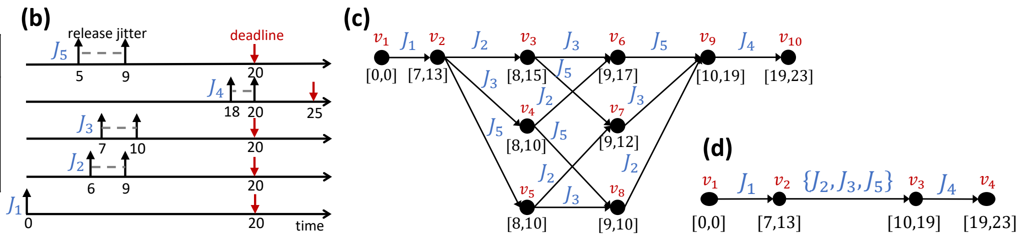

(b)
release jitter
deadline
J_5 5 9 20
J_4 18 20 25
J_3 7 10 20
J_2 6 9 20
J_1 0 20 time
(c)
v_1 [0,0] J_1 v_2 [7,13]
v_3 [8,15] v_4 [8,10] v_5 [8,10]
v_6 [9,17] v_7 [9,12] v_8 [9,10]
v_9 [10,19] J_4 v_{10} [19,23]
(d)
v_1 [0,0] J_1 v_2 [7,13] $\{J_2,J_3,J_5\}$ v_3 [10,19] J_4 v_4 [19,23]

Fig. 1: An example showing the difference between the SAG constructed by the original analysis and our new POR-based analysis. (a) An example job set with release jitter. (b) Visual representation of release intervals and deadlines. (c) SAG constructed by the analysis from Nasri et al. [7]. (d) SAG constructed by our POR-based analysis for the same system.

keep track of different schedules. However, it also forces the analysis to work with uncertainties. Namely, now to decide how v_2 will change after the next scheduling decision, SAG must consider any job that can be *dispatched next* in this state. In our example, due to the release jitter, either J_2, J_3, or J_5 can be dispatched next (or can be released next), after which any of the two remaining jobs can be dispatched. For example, if J_5 is released at time 5 and J_2 and J_3 are released at time 9 and 8, respectively, and the execution time of J_1 is not more than 7, then the first job that will be dispatched by the scheduler is J_5 and then job J_3 and finally job J_2. Alternatively, we could see a scenario in which J_2 is dispatched first (e.g., when it is released at time 6). The SAG captures all these scenarios in the graph by adding out-going edges to the states for each possible scheduling decision. After building the graph, the smallest and largest observed finish time of a job (on any edge with the label of that job) is reported as the BCRT and WCRT of the job, respectively.

It becomes clear from the above discussion that, when there are large uncertainties in the timing parameters, the analysis will face a state-space explosion due to the combinatorial increase in the number of possible orderings between jobs. Our key observation to improve the performance of the SAG analysis is that *the exploration of many of these job execution orderings is not relevant to asserting schedulability of the job set, as none of these orderings may lead to a deadline miss*. For example, the graph in Fig. 1(c), has 10 nodes and 14 edges. Yet, none of the possible job execution orderings of J_2, J_3, and J_5 considered in the graph may lead to a deadline miss. Ideally, we would want to skip over these three jobs without exploring every individual job execution ordering. By avoiding the exploration of such scenarios, we could get a performance gain in the analysis and have a much smaller graph like the one shown in Fig. 1(d).

IV. Partial-order reduction (POR)

Our key idea is to identify subsets of jobs for which the combinatorial exploration of all orderings is *irrelevant* to schedulability of the job set. Exploring these combinations is irrelevant when **(i)** all scenarios lead to the same system state and **(ii)** none of the jobs observe a deadline miss. Dispatching such jobs can be considered in a single step (that combines all those scheduling decisions), which further defers the state-space explosion. The POR technique proposed in this work allows us to identify such job orderings and treat them as a *batch of scheduling decisions on a single edge in the SAG*.

Fig. 1(d) shows the SAG constructed using our proposed POR technique, where J_2, J_3, and J_5 are combined and assigned to a single edge, removing the need to enumerate all possible scenarios between these jobs and shrinking the resulting SAG. Note that the interval recorded in v_9 in Fig. 1(c) is the same as the interval recorded in v_3 in Fig. 1(d), even though the latter did not explore all different scenarios.

As shown earlier, when *expanding* a state v_p, the original SAG analysis of Nasri et al. [7] simply adds a new vertex (state) for each job that is a *direct successor* of v_p.

Definition 2. (From [9]) Let $\mathcal{J}^P$ be the set of jobs already dispatched until reaching state v_p. A job $J_j \in \mathcal{J} \setminus \mathcal{J}^P$ is a *direct successor* for path P ending in vertex v_p iff (if and only if) there exists an execution scenario in which job J_j is dispatched *after* state v_p and *before* any other job.

In this work, we use POR to determine whether a set of future (i.e., not yet dispatched) jobs can be "reduced" to a single edge that encompasses all orderings of the original jobs without explicitly exploring all these orderings. We call the candidate job set $\mathcal{J}^S$ considered for a reduction to a single edge the *candidate reduction set*. If the candidate reduction set meets the conditions for a *safe* POR (as will be defined in Definition 5), it is added to the graph using a single edge and vertex. Otherwise, the graph is expanded with a new vertex for each direct successor as per the original SAG.

A. Problem definition

An important property of the original schedulability analysis by Nasri et al. [7] is that it is both an exact schedulability and an exact (tight) response-time analysis.

Definition 3. An analysis is an *exact schedulability analysis* iff **(i)** for all job sets that are deemed schedulable by the analysis, there is no execution scenario that may result in a deadline miss, and **(ii)** for all job sets deemed unschedulable by the analysis, there is at least one execution scenario that results in a deadline miss.

Definition 4. An analysis is an *exact response-time analysis* (namely, produces *tight* response-time bounds) iff **(i)** there is no execution scenario such that any job has a response-time smaller than the best-case response time (BCRT) or larger than the WCRT returned by the analysis for that job, and **(ii)** for each job, there must be an execution scenario where the job experiences a response time equal to the computed BCRT and another where it experiences the computed WCRT.

However, since the goal of POR is to eliminate the need to explore excessive job execution orderings, we need to make a trade-off between the exactness of the analysis and its scalability. In this work, *we propose an exact schedulability analysis that returns safe but not tight response-time bounds* (i.e., our lower bound might be smaller than the actual BCRT of a job and our upper bound might be larger than the actual WCRT of the job) in return for a reduced state-space. That is, we present an analysis that satisfies both conditions **(i)** and **(ii)** of Definition 3 but only condition **(i)** of Definition 4.

Thus, we define a safe partial-order reduction as follows.

Definition 5. A POR of a set of jobs $\mathcal{J}^S$ is *safe* iff it maintains both conditions of Definition 3 (i.e., provide an exact schedulability analysis) and satisfies condition (i) in Definition 4 (i.e., derives safe bounds on the response time).

The key to having a safe POR is that the jobs in the reduction set do not affect the response time of the jobs that are not contained in the reduction set. Therefore, we define a *safe reduction set* as follows.

Definition 6. Given a system state v_p, a set of jobs is a *safe reduction set* (denoted by $\mathcal{J}^M(v_p)$) iff there is no other job in $\mathcal{J} \setminus (\mathcal{J}^M(v_p) \cup \mathcal{J}^P)$ that can start executing *before all jobs in $\mathcal{J}^M(v_p)$ finish their execution.*

For simplicity, we omit to specify the state in the notation and simply denote a safe reduction set by $\mathcal{J}^M$ when it is apparent from the context which state is being referred to. We will discuss how to compute $\mathcal{J}^M$ in detail in Section IV-H.

Having defined the criteria that should hold for a safe POR with a safe reduction set, we are ready to formally introduce our problem as follows:

Problem 1. Given a system state v_p, find a safe reduction set $\mathcal{J}^M$ that satisfies the conditions of Definitions 5 and 6.

B. Graph generation using partial-order reduction

We follow a top-down approach to explain the high-level idea of our solution before going through its details. Algorithm 1 summarizes how to construct the SAG using POR. It is based on the SAG construction algorithm of Nasri et al. [9], with the addition of POR at lines 4-11.

We first create a safe reduction set $\mathcal{J}^M$ at line 4 (using Algorithm 4 presented later in Section IV-H). If $\mathcal{J}^M$ is not empty (line 5), Algorithm 1 applies POR. It computes a lower and upper bound, denoted $\overline{EFT}$ and $\overline{LFT}$, respectively, on the finish time of all jobs in $\mathcal{J}^M$ using Algorithms 2 and 3 introduced later in Section IV-C. A new vertex v_k is then created using $\overline{EFT}$ and $\overline{LFT}$ as bounds on the earliest and latest time by which the processor may become available after executing all jobs in $\mathcal{J}^M$. Vertex v_k is then added to the graph at line 7. After the application of the POR, the algorithm is identical to the original SAG analysis by applying the same merge phase as in [9] (lines 8-11).

Algorithm 1: Schedule-abstraction graph construction using partial-order reduction

```
Input  : Job set $\mathcal{J}$
Output: Schedule graph $G = (V, E)$
1  Initialize $G$ by adding a root vertex $v_1$ with label $[0, 0]$;
2  while $\exists$ path $P$ from $v_1$ to a leaf $v_p$ s.t. $|\mathcal{J}^P| < |\mathcal{J}|$ do
3      $P \leftarrow$ path with the smallest set $\mathcal{J}^P$ from $v_1$ to a leaf
         vertex $v_p$;
4      Create $\mathcal{J}^M$ using Algorithm 4;
5      if $\mathcal{J}^M \neq \emptyset$ then
6          Create $v_k$ with label
             $[\overline{EFT}(\mathcal{J}^M, v_p), \overline{LFT}(\mathcal{J}^M, v_p)]$ according to
             Algorithms 2 and 3;
7          Connect $v_p$ to $v_k$ by an edge with label $\mathcal{J}^M$;
8          while $\exists$ path $Q$ that ends with $v_q$ such that $v_k$ and
             $v_q$ can be merged [9] do
9              Merge $v_k$ and $v_q$ by updating $v_k$ (see [9]);
10             Redirect all incoming edges of $v_q$ to $v_k$;
11             Remove $v_q$ from $V$;
12     else
13         Expand $G$ with a vertex for each direct successor $J_j$
             of $v_p$ as in [9] ;
```

If Algorithm 4 returns an empty set $\mathcal{J}^M$ (i.e., no safe reduction set could be found), we continue with the original SAG analysis by applying the same expansion phase as in [9].

C. Constructing a safe reduction set

According to Definition 6, a candidate reduction set $\mathcal{J}^S$ is a safe reduction set iff there is no other job $J_x \in \mathcal{J} \setminus (\mathcal{J}^M(v_p) \cup \mathcal{J}^P)$ that may start to execute before the jobs in $\mathcal{J}^S$ complete. An interfering job for candidate reduction set $\mathcal{J}^S$ is thus defined as follows.

Definition 7. A job $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$ is an *interfering* job for $\mathcal{J}^S$ iff J_x can execute before any of the jobs in $\mathcal{J}^S$.

The conditions under which a job J_x can interfere with $\mathcal{J}^S$ depend on the scheduling policy used for the job set under analysis. Since this work analyses work-conserving JLFP scheduling algorithms, the interference conditions relate to the *work-conserving* and *priority-driven* properties of the scheduling algorithm.

Work-conserving interference condition. Let $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$ be a job that neither belongs to the set of already-dispatched jobs ($\mathcal{J}^P$) nor the candidate reduction set $\mathcal{J}^S$. The job J_x can execute between two jobs in $\mathcal{J}^S$ if J_x is released before the end of an idle interval between the execution of two jobs in $\mathcal{J}^S$. Consider the example in Figure 2, where $\mathcal{J}^S = \{J_1, J_2\}$ and J_x is J_3. In a scenario where J_1 is released at 6 and J_2 at 12, it is possible for a low-priority job such as J_3 to be dispatched between the two higher-priority jobs in $\mathcal{J}^S$, i.e., at time 11. This chance is provided for J_3 because it is released before (or at) the end of an idle interval.

In order to find out whether a job J_x may start to execute within an idle interval between the execution of two jobs in

(a)

J_i	r_i^{min}	r_i^{max}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	6	8	1	2	30	1
J_2	7	12	1	2	30	2
J_3	11	14	1	2	30	3

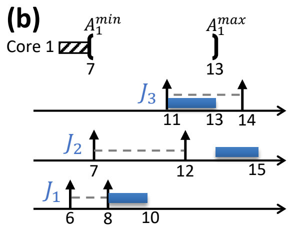


Fig. 2: An example where J_3 interferes with a candidate reduction set $\mathcal{J}^S = \{J_1, J_2\}$ because J_3 releases before the end of an idle time in $\mathcal{J}^S$.

(a)

J_i	r_i^{min}	r_i^{max}	C_i^{min}	C_i^{max}	d_i	p_i
J_1	6	8	1	2	30	1
J_2	7	10	1	2	30	2
J_3	12	13	1	3	35	3
J_4	5	9	1	2	30	4

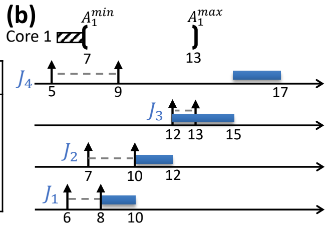


Fig. 3: An example where J_3 interferes with a candidate reduction set $\mathcal{J}^S = \{J_1, J_2, J_4\}$ because J_3 releases before a lower-priority job in $\mathcal{J}^S$ starts (in this case, J_4).

$\mathcal{J}^S$, we need to determine when idle intervals may happen. To do so, we compute two bounds: (i) the earliest time by which an idle interval may start prior to the execution of a job $J_i \in \mathcal{J}^S$, and (ii) the latest time by which such idle interval may end. Lemma 1 below computes a bound for the latter.

Lemma 1. *The latest idle interval before the execution of $J_i \in \mathcal{J}^S$ cannot end later than r_i^{max}.*

Proof. By contradiction. Suppose that there could be an idle interval before the execution of $J_i \in \mathcal{J}^S$ that ends at $r_i^{max}+1$. Then, it means that the processor remains idle until $r_i^{max} + 1$ even though J_i is certainly released. This contradicts the assumption that the scheduling policy is work-conserving, as a work-conserving scheduling policy never keeps the processor idle when there is a ready job. $\square$

To determine whether there is a potential idle interval ending at r_i^{max}, we must compute when that idle interval may start. To do so, we first define the set of jobs in $\mathcal{J}^S$ that are certainly released before time t as

$$\mathcal{C}(t, v_p) = \{J_j \in \mathcal{J}^S \mid r_j^{max} < t\} \tag{1}$$

Now, let $\overline{EFT}(\mathcal{C}(t, v_p), v_p)$ be a lower bound on the finish time of all jobs in $\mathcal{C}(t, v_p)$. If $\overline{EFT}(\mathcal{C}(t, v_p), v_p)$ is strictly smaller than t, then there may be an idle interval starting at $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ and ending by t. Otherwise, there cannot be any idle interval immediately before t.

Therefore, based on the above discussion, we define the set of jobs in $\mathcal{J}^S$ before which an idle interval may exist as

$$\mathcal{J}^\delta = \{J_i \mid J_i \in \mathcal{J}^S \wedge \overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p) < r_i^{max}\} \tag{2}$$

We formally prove the above properties in the two following lemmas.

Lemma 2. *Let J_i be a job in $\mathcal{J}^S$ that is released at r_i^{max}. If there is an idle interval just before the execution of J_i starts, then $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is a lower bound on the start time of that idle interval.*

Proof. Since by (1), all jobs in $\mathcal{C}(r_i^{max}, v_p)$ must have been released strictly before r_i^{max}, if there is an idle interval before J_i starts to execute, then all the jobs in $\mathcal{C}(r_i^{max}, v_p)$ must have completed their execution by r_i^{max} (by the work-conserving property).

Now, by contradiction, suppose that there is an idle interval that starts before $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$. This means that the jobs in $\mathcal{C}(r_i^{max}, v_p)$ can finish before $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$. This contradicts the fact that $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is the EFT of the set of jobs $\mathcal{C}(r_i^{max}, v_p)$ as proven in Lemma 6. $\square$

Lemma 3. *If $\mathcal{J}^\delta = \emptyset$, there exists no execution scenario such that there is an idle interval between the execution of two jobs in $\mathcal{J}^S(v_p)$ scheduled after v_p.*

Proof. If $\mathcal{J}^\delta = \emptyset$ it means that for each $J_i \in \mathcal{J}^S$, $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p) \geq r_i^{max}$. Since by definition, $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ is a lower bound on the finish time of $\mathcal{C}(r_i^{max}, v_p)$, the idle interval ending with r_i^{max} is empty for each r_i^{max}. Hence, there is no idle interval between any two arbitrary jobs in $\mathcal{J}^S$. $\square$

For instance, given a candidate reduction set $\mathcal{J}^S = \{J_1, J_2\}$ as in the example of Figure 2, $\mathcal{C}(r_1^{max}, v_p) = \mathcal{C}(8, v_p) = \emptyset$ and $\mathcal{C}(r_2^{max}, v_p) = \mathcal{C}(12, v_p) = \{J_1\}$. In this example, the earliest finish time of J_1 is at time 8 which happens when it is released at time 6, the processor becomes available at time 7 ($A_1^{min} = 7$), and J_1 has one unit of execution time (Sec. IV-D will explain how to obtain $\overline{EFT}$). Finally, from Equation (2), we get $\mathcal{J}^\delta = \{J_2\}$.

Now that we know when idle intervals between jobs in $\mathcal{J}^S$ may end, we can formulate the condition that should hold for a job to be able to interfere with the jobs in $\mathcal{J}^S$ due to the work-conserving property of the scheduling algorithm.

Lemma 4. *If a job $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$ can execute in an idle interval between two jobs in $\mathcal{J}^S$, then*

$$\mathcal{J}^\delta \neq \emptyset \wedge r_x^{min} < \delta_M(v_p), \tag{3}$$

where $\delta_M(v_p) = \max\{r_j^{max} \mid J_j \in \mathcal{J}^\delta\}$.

Proof. If $\mathcal{J}^\delta \neq \emptyset$, $\delta_M(v_p)$ is the latest end of an idle interval ending before the execution of the last job in $\mathcal{J}^S$. J_x is released before $\delta_M(v_p)$, so J_x will be a ready job at some idle instant before $\delta_M(v_p)$. As the scheduler is work-conserving it will not leave the processor idle when there is a ready job, and hence will schedule J_x. Thus, J_x can execute between two jobs in $\mathcal{J}^S$ and interferes with $\mathcal{J}^S$. $\square$

Going back to the example in Figure 2 with $\mathcal{J}^S = \{J_1, J_2\}$, we see that for the job J_3, (3) is 'true' because $\mathcal{J}^\delta = \{J_2\}$, $\delta_M(v_p) = 12$, and $r_3^{min} = 11 < 12$.

Priority-driven interference condition. A job $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$ can execute between two arbitrary jobs in $\mathcal{J}^S$ if it has a higher priority than a job $J_l \in \mathcal{J}^S$ and is released before J_l started to execute. Figure 3 shows an example of a higher-priority job (namely J_3) interfering with $\mathcal{J}^S = \{J_1, J_2, J_4\}$ because it is released before the time at which a lower-priority job in $\mathcal{J}^S$ (i.e., J_4) has started executing.

Let $\mathcal{J}^{high}$ be the set containing all the jobs not in $\mathcal{J}^S$ that have a higher priority than at least one job in $\mathcal{J}^S$, formally

$$\mathcal{J}^{high} = \{J_x \mid J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P) \ \wedge \ \exists J_l \in \mathcal{J}^S, p_x < p_l\} \quad (4)$$

Then, Lemma 5 provides a sufficient condition under which there is no interfering job for a candidate reduction set $\mathcal{J}^S$.

Lemma 5. *Let $\widehat{LST}_i(\mathcal{J}^S, v_p)$ be an upper bound on the start time of any job $J_i \in \mathcal{J}^S$. If there exists no $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$ such that* (3) *holds* ***and*** *$\forall J_y \in \mathcal{J}^{high}$, $\forall J_l \in \mathcal{J}^S$ such that $p_y < p_l$ we have $r_y^{min} > \widehat{LST}_l(\mathcal{J}^S, v_p)$, then there exists no interfering job for $\mathcal{J}^S$ and $\mathcal{J}^S$ is a safe reduction set.*

Proof. Under a non-preemptive work-conserving JLFP policy, a job $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$ can only start its execution before a job $J_i \in \mathcal{J}^S$ iff: **(i)** J_x has a higher priority than J_i and J_x is possibly released before J_i starts executing **or (ii)** if the processor is idle before the start of J_i, and J_x possibly releases before or during this idle time interval.

Lemma 4 proves that (3) must hold if J_x interferes with $\mathcal{J}^S$ by executing in an idle interval. Since by the lemma's assumption, (3) does not hold for any job $J_x \in \mathcal{J} \setminus (\mathcal{J}^S \cup \mathcal{J}^P)$, no job respects condition (ii). Therefore, if the claim does not hold, then there must be a set of jobs $\mathcal{J}^I$ not in $\mathcal{J}^S$ (i.e., $\mathcal{J}^I \cap \mathcal{J}^S = \emptyset$) that satisfies condition (i). That is, for all $J_x \in \mathcal{J}^I$, there exists $J_l \in \mathcal{J}^S$ such that $p_x < p_l$ and J_x is released before J_l starts executing.

By assumption, $\widehat{LST}_l(\mathcal{J}^S, v_p)$ is an upper bound on the start time of any job $J_l \in \mathcal{J}^S$ when no job interferes with $\mathcal{J}^S$. Hence, for the jobs in $\mathcal{J}^I$ to interfere with $\mathcal{J}^S$, there must be at least one job $J_y \in \mathcal{J}^I$ and a job $J_l \in \mathcal{J}^S$ such that $p_y < p_l$ and J_y is released before or at $\widehat{LST}_l(\mathcal{J}^S, v_p)$ (from condition (i)). This contradicts the lemma's assumption that $r_y^{min} > \widehat{LST}_l(\mathcal{J}^S, v_p)$. Therefore, the set $\mathcal{J}^I$ must be empty, thereby proving that no job can interfere with $\mathcal{J}^S$ and, by Definition 6, $\mathcal{J}^S$ is a safe reduction set. □

An upper bound $\widehat{LST}_i(\mathcal{J}^S, v_p)$ on the start time of any job $J_i \in \mathcal{J}^S$ can be computed using (8) presented in Section IV-F.

D. Computing $\overline{EFT}$

Lemmas 2 to 5 require a lower bound $\overline{EFT}(\mathcal{C}(r_i^{max}, v_p), v_p)$ on the finish time of any job in the job set $\mathcal{C}(r_i^{max}, v_p)$.

A lower bound on the EFT of any set of jobs $\mathcal{J}^X$ can be computed by scheduling all jobs in $\mathcal{J}^X$ as early as possible and assume that they execute for their BCET. Algorithm 2 shows how to obtain this lower bound, i.e., $\overline{EFT}(\mathcal{J}^X, v_p)$. Further in Lemma 6, we prove that $\overline{EFT}(\mathcal{J}^X, v_p)$ as calculated by Algorithm 2 is indeed a lower bound on the earliest time at which the processor can possibly become available after dispatching the jobs in $\mathcal{J}^X$ in any possible execution order following system state v_p.

Algorithm 2: Earliest finish time of $\mathcal{J}^X$ after state v_p

```
Input  : Job set $\mathcal{J}^X$, system state $v_p$
Output: Earliest finish time $\overline{EFT}(\mathcal{J}^X, v_p)$
1 Sort $\mathcal{J}^X$ by $r^{min}$ ascending, break ties by highest $p$;
2 $\overline{EFT} \leftarrow A_1^{min}(v_p)$;
3 for each job $J_x \in \mathcal{J}^X$ do
4     $\overline{EFT} \leftarrow \max\{\overline{EFT}, r_x^{min}\} + C_x^{min}$;
5 return $\overline{EFT}$ ;
```

Lemma 6. *The set of jobs $\mathcal{J}^X$ scheduled after state v_p cannot complete its execution earlier than $\overline{EFT}(\mathcal{J}^X, v_p)$ as returned at line 5 of Algorithm 2.*

Proof. At line 1 of Algorithm 2, the jobs in $\mathcal{J}^X$ are sorted in ascending order of r^{min}, and any ties are broken by highest priority. Let J_k denote the k^{th} job in the ordered set $\mathcal{J}^X$, and let $J_k^X = \{J_1, J_2, \ldots, J_k\}$ denote the re-ordered (re-indexed) set of jobs in $\mathcal{J}^X$. And finally, let $\overline{EFT}_k$ be the value computed at line 4 of Algorithm 2 after the k^{th} iteration of the for-loop.

We prove by induction that $\overline{EFT}_k$ is a lower bound on the finish time of all jobs in J_k^X assuming that no job $J_j \in \mathcal{J} \setminus (J^P \cup J_k^X)$ executes between the jobs in J_k^X. The base case considers the first job J_1. Job J_1 is the job with the earliest r^{min} in $\mathcal{J}^X$, and in case of a tie, the highest-priority one. We prove that line 4 of Algorithm 2 computes a lower bound on the EFT of J_1. Since a job cannot start executing before it is released and the processor is available, $\max\{A_1^{min}, r_1^{min}\}$ is a lower bound on the start time of J_1. Therefore, J_1 cannot finish before $\max\{A_1^{min}, r_1^{min}\} + C_1^{min}$ as computed at line 4 since C_1^{min} is the BCET of J_1.

In the induction step, $\overline{EFT}_{k-1}$ is a lower bound on the finish time of the jobs in $\{J_1, \ldots, J_{k-1}\}$ assuming no other job executes between them. We show that $\overline{EFT}_k$ as computed at line 4 of Algorithm 2 is the EFT of all jobs $\{J_1, \ldots, J_k\}$. We divide the proof into two cases depending on whether J_k starts its execution before or after the completion of the jobs in $J_{k-1}^X = \{J_1, \ldots, J_{k-1}\}$.

Case (i): Assume that J_k does not start its execution before the jobs in J_{k-1}^X have finished, namely, it does not start before $\overline{EFT}_{k-1}$. We know that J_k cannot start before r_k^{min} since J_k cannot start before it is released. Thus, $\max\{\overline{EFT}_{k-1}, r_k^{min}\}$ is a lower bound on the start time of J_k. Since C_k^{min} is the BCET of J_k, if J_k starts executing at $\max\{\overline{EFT}_{k-1}, r_k^{min}\}$ then it cannot finish before $\overline{EFT}_k = \max\{\overline{EFT}_{k-1}, r_k^{min} + C_k^{min}\}$ as computed at line 4. Furthermore, since J_k starts executing after all jobs in J_{k-1}^X completed their own execution, the finish time $\overline{EFT}_k$ of J_k is also a lower bound on the finish time of all the other jobs in $J_k^X = J_{k-1}^X \cup \{J_k\}$.

Case (ii): Assume that J_k starts executing before $\overline{EFT}_{k-1}$. Let s_k be the start time of J_k. By assumption, $s_k < \overline{EFT}_{k-1}$. Since J_k cannot start before it is released, $r_k^{min} \le s_k$. We show that $\overline{EFT}_k$ computed as $\overline{EFT}_k = \overline{EFT}_{k-1} + C_k^{min}$ by Algorithm 2 is a lower bound on the finish time of $J_k^X = J_{k-1}^X \cup \{J_k\}$. Since all jobs in J_{k-1}^X have their release before that of J_k by line 1 of Algorithm 2, the processor executes at least $\overline{EFT}_{k-1} - s_k$ time units of workload from the jobs in J_{k-1}^X *after* s_k. Thus, when including J_k, the processor must execute at least $\overline{EFT}_{k-1} - s_k + C_k^{min}$ time units of workload of J_k^X after s_k. Therefore, the jobs in J_k^X cannot complete before $s_k + \overline{EFT}_{k-1} - s_k + C_k^{min} = \overline{EFT}_{k-1} + C_k^{min}$, which is $\overline{EFT}_k$ as computed at line 4 of Algorithm 2.

Hence, if we apply the inductive step to all jobs in $\mathcal{J}^X$, then line 5 of Algorithm 2 returns a lower bound on the earliest time the processor can finish scheduling all jobs in $\mathcal{J}^X$. □

E. Exact finish time intervals of a safe reduction set

If Lemma 5 holds for a job set $\mathcal{J}^S$, then $\mathcal{J}^S$ is a safe reduction set (denoted by $\mathcal{J}^M$ from now on) that satisfies Definition 6. Algorithm 1 thus requires to compute a lower bound (EFT) and upper bound (LFT) on the finish time of all jobs in $\mathcal{J}^M$ (line 6 of Algorithm 1). In this section, we explain how to compute *exact bounds* on the finish time of a safe reduction set $\mathcal{J}^M$ where an exact EFT and LFT are defined as follows.

Definition 8. The EFT of a safe reduction set $\mathcal{J}^M$ is *exact* iff the set of jobs $\mathcal{J}^M$ cannot complete its execution earlier than EFT and there exists an execution scenario such that all jobs in $\mathcal{J}^M$ have completed exactly at EFT.

Definition 9. The LFT of a safe reduction set $\mathcal{J}^M$ is *exact* iff the set of jobs $\mathcal{J}^M$ cannot complete its execution later than LFT and there exists an execution scenario such that all jobs in $\mathcal{J}^M$ have completed exactly at LFT.

First, we prove using Lemma 7 and Corollary 1 that the bound $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 2 is an *exact bound on EFT* when $\mathcal{J}^M$ is a safe reduction set.

Lemma 7. *There exists an execution scenario such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 2.*

Proof. If each $J_i \in \mathcal{J}^M$ releases at r_i^{min} and executes for exactly C_i^{min} time units and the processor becomes available at A_1^{min}, then, the execution of $\mathcal{J}^M$ will complete exactly at $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 2 since this algorithm practically simulates the schedule of the jobs in $\mathcal{J}^M$ under the given execution scenario. Hence, there exists an execution such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{EFT}(\mathcal{J}^M, v_p)$. □

Corollary 1. *The $\overline{EFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 2 is the exact earliest finish time of $\mathcal{J}^M$.*

Proof. The set of jobs $\mathcal{J}^M$ cannot complete its execution before $\overline{EFT}(\mathcal{J}^M, v_p)$ (Lemma 6), and there exists an execution scenario such that the jobs in $\mathcal{J}^M$ complete at $\overline{EFT}(\mathcal{J}^M, v_p)$ (Lemma 7). Hence, $\overline{EFT}(\mathcal{J}^M, v_p)$ is exact. □

The LFT of a safe reduction set $\mathcal{J}^M$ can be determined by scheduling all jobs as late as possible and by assuming that they execute for their WCET. Algorithm 3 does exactly that, and Lemma 8, Lemma 9 and Corollary 2 prove that the bound $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 3 is an *exact bound on LFT* when $\mathcal{J}^M$ is a safe reduction set.

Algorithm 3: Latest finish time of $\mathcal{J}^M$ after state v_p

```
Input  : Job set $\mathcal{J}^M$, system state $v_p$
Output: Latest finish time $\overline{LFT}(\mathcal{J}^M, v_p)$
1 Sort $\mathcal{J}^M$ by $r^{max}$ ascending, break ties by highest $p$;
2 $\overline{LFT} \leftarrow A_1^{max}(v_p)$;
3 for each job $J_x \in \mathcal{J}^M$ do
4     $\overline{LFT} \leftarrow \max\{\overline{LFT}, r_x^{max}\} + C_x^{max}$;
5 return $\overline{LFT}$ ;
```

Lemma 8. *The set of jobs $\mathcal{J}^M$ scheduled after state v_p cannot complete their execution later than $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 3.*

Proof. At line 1 of Algorithm 3, the jobs in $\mathcal{J}^M$ are sorted in ascending order of r^{max}, and any ties are broken by highest priority. Let J_k denote the k^{th} job in the ordered set and let $J_k^M = \{J_1, \ldots, J_k\}$ denote the re-indexed set of jobs in $\mathcal{J}^M$ sorted by their r^{max} (in line 1 of the algorithm). Finally, let $\overline{LFT}_k$ be the value computed at line 4 of Algorithm 3 after the k^{th} iteration of the for-loop.

We prove by induction that $\overline{LFT}_k$ is an upper bound on the finish time of all jobs in J_k^M assuming that no job $J_j \in \mathcal{J} \setminus (J^P \cup J_k^M)$ executes between the jobs in J_k^M. The base case considers the first job J_1. J_1 is the job with the earliest r^{max} in $\mathcal{J}^M$, and in case of a tie, the highest priority one. We prove that line 4 of Algorithm 3 computes an upper bound on the LFT of J_1. Because the scheduler is work-conserving, a job cannot start later than the time by which it is certainly released and the processor is certainly available. Hence, $\max\{A_1^{max}, r_1^{max}\}$ is an upper bound on the start time of J_1. If J_1 starts at $\max\{A_1^{max}, r_1^{max}\}$ it cannot finish after $\max\{A_1^{max}, r_1^{max}\} + C_1^{max}$ as computed at line 4 of Algorithm 3 since C_1^{max} is the WCET of J_1.

In the induction step, $\overline{LFT}_{k-1}$ is an upper bound on the finish time of the jobs in $\{J_1, \ldots, J_{k-1}\}$ assuming no other job executes between them. We show that $\overline{LFT}_k$ as computed at line 4 of Algorithm 3 is the LFT of all jobs $\{J_1, \ldots, J_k\}$. We divide the proof into two cases depending on whether J_k starts its execution before or after the completion of the jobs in $J_{k-1}^M = \{J_1, \ldots, J_{k-1}\}$.

Case (i): If J_k does not start its execution before the jobs in J_{k-1}^M have finished, then, we know that $\overline{LFT}_{k-1}$ is the latest time at which the processor becomes available to other jobs including J_k. We also know that J_k will be released at the latest by time r_k^{max}. Hence, an upper bound on the start time of J_k is $\max\{\overline{LFT}_{k-1}, r_k^{max}\}$ because at that time, a work-conserving scheduler must dispatch a job (in this case, J_k). Since C_k^{max} is the WCET of J_k, if J_k

starts executing at $\max\{\overline{LFT}_{k-1}, r_k^{max}\}$ then it cannot finish after $\overline{LFT}_k = \max\{\overline{LFT}_{k-1}, r_k^{max} + C_k^{max}\}$ as computed at line 4. Furthermore, since J_k starts executing after all jobs in J_{k-1}^M completed their own execution, the finish time $\overline{LFT}_k$ of J_k is also an upper bound on the finish time of all the other jobs in $J_k^M = J_{k-1}^M \cup \{J_k\}$.

Case (ii): If J_k starts executing before $\overline{LFT}_{k-1}$, then let s_k be the start time of J_k. By assumption, $s_k < \overline{LFT}_{k-1}$. We show that $\overline{LFT}_k$ computed as $\overline{LFT}_k = \overline{LFT}_{k-1} + C_k^{max}$ by Algorithm 3 is an upper bound on the finish time of $J_k^M = J_{k-1}^M \cup \{J_k\}$. Since all jobs in J_{k-1}^M have their latest release (r^{max}) before that of J_k by line 1 of Algorithm 3, the processor executes at most $\overline{LFT}_{k-1} - s_k$ time units of workload from the jobs in J_{k-1}^M after s_k. Thus, the processor must execute at most $\overline{LFT}_{k-1} - s_k + C_k^{max}$ time units of workload of J_k^M after s_k. Therefore, the jobs in J_k^M cannot complete after $s_k + (\overline{LFT}_{k-1} - s_k + C_k^{max}) = \overline{LFT}_{k-1} + C_k^{max}$, which is $\overline{LFT}_k$ as computed at line 4 of Algorithm 3.

Hence, if we apply the inductive step to all jobs in $\mathcal{J}^M$, then line 5 of Algorithm 3 returns an upper bound on the latest time the processor can finish scheduling all jobs in $\mathcal{J}^M$. □

Lemma 9. *There exists an execution scenario such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 3.*

Proof. If each $J_i \in \mathcal{J}^M$ releases at r_i^{max} and executes for exactly C_i^{max} time units and the processor becomes available at A_1^{max}, then, the execution of $\mathcal{J}^M$ will complete exactly at $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 3. Hence, there exists an execution such that all jobs in $\mathcal{J}^M$ have completed exactly at $\overline{LFT}(\mathcal{J}^M, v_p)$. □

Corollary 2. *The $\overline{LFT}(\mathcal{J}^M, v_p)$ as returned by line 5 of Algorithm 3 is the exact latest finish time of $\mathcal{J}^M$.*

Proof. The set of jobs $\mathcal{J}^M$ cannot complete its execution after $\overline{LFT}(\mathcal{J}^M, v_p)$ (Lemma 8), and there exists an execution scenario such that the jobs in $\mathcal{J}^M$ complete at $\overline{LFT}(\mathcal{J}^M, v_p)$ (Lemma 9). Hence, $\overline{LFT}(\mathcal{J}^M, v_p)$ is exact. □

F. Exact schedulability analysis of a safe reduction set

In order to efficiently determine whether a deadline miss may happen for the jobs in $\mathcal{J}^M$, we use a sufficient schedulability test on $\mathcal{J}^M$. If $\mathcal{J}^M$ passes the sufficient test, i.e., it is certainly schedulable, we apply the POR and create a single edge for $\mathcal{J}^M$ in the SAG (lines 5 to 11 in Algorithm 1). Otherwise, we do not know whether a deadline miss could happen unless we explore all possible execution orderings of the jobs in $\mathcal{J}^M$. Therefore, we reject $\mathcal{J}^M$ and let the original SAG analysis explore the schedules that could be generated from these jobs for us (line 13 in Algorithm 1).

The sufficient schedulability test for $\mathcal{J}^M$ is defined as

$$\forall J_i \in \mathcal{J}^M, \widehat{LFT}_i(\mathcal{J}^M, v_p) \le d_i, \tag{5}$$

where $\widehat{LFT}_i(\mathcal{J}^M, v_p)$ is an upper bound on the finish time of $J_i \in \mathcal{J}^M$ when scheduled after state v_p.

In order to compute $\widehat{LFT}_i(\mathcal{J}^M, v_p)$, we need an upper bound on the start time of every job $J_i \in \mathcal{J}^M$.

As shown by Davis et al. [3], for a non-preemptive JLFP scheduling policy, the following execution scenario results in a late start time for any job J_i: a lower priority job starts its execution *just before* J_i is released and, subsequently, all higher-priority jobs interfere with J_i.

While the above-mentioned scenario is very pessimistic and often does not happen in practice, assuming that scenario allows us to easily compute an upper bound on the start time of J_i using a fixed-point iteration equation. Note that it is possible to compute a tighter upper bound on J_i's start time. However, it would require exploring more execution scenarios and thus generate more overhead. Consequently, we compute an upper bound $s_i(v_p)$ on the start time of any job $J_i \in \mathcal{J}^M(v_p)$ using the following recursive equations and stopping when $s_i^{(k)} = s_i^{(k-1)}$.

$$s_i^{(0)} = \max\{A_1^{max}, r_i^{max} - 1 + \max_{\forall J_j \in \mathcal{J}^M(v_p)} \{C_j^{max} \mid p_i < p_j\}\} \tag{6}$$

$$s_i^{(k)} = s_i^{(0)} + \sum_{\{J_j \mid J_j \in \mathcal{J}^M(v_p) \wedge r_j^{min} \le s_i^{(k-1)} \wedge p_j < p_i\}} C_j^{max} \tag{7}$$

Lemma 10. *The fixed-point iteration in* (7) *converges.*

Proof. $s_i^{(k)}$ increases or remains constant as only non-negative terms are added because $\forall J_j$, $C_j^{max} \ge 0$. Furthermore, $s_i^{(k)}$ increases when there exists $J_j \in \mathcal{J}^M(v_p)$ such that J_j has a higher priority than J_i and $s_i^{(k-2)} < r_j^{min} \le s_i^{(k-1)}$. If such a J_j does not exist, there is no more job that has a higher priority than J_i and is possibly released at $s_i^{(k-1)}$ and $s_i^{(k)} = s_i^{(k-1)}$. Therefore, as long as it does not converge, (7) must add at least one more job from $\mathcal{J}^M(v_p)$ at every iteration. Since the number of jobs in $\mathcal{J}^M(v_p)$ is finite, the number of iterations by (7) is upper bounded by $|\mathcal{J}^M(v_p)|$. This proves the lemma. □

Lemma 11. *$J_i \in \mathcal{J}^M(v_p)$ starts executing no later than $s_i(v_p)$.*

Proof. The proof is by contradiction. Assume that the processor starts executing $J_i \in \mathcal{J}^M(v_p)$ later than $s_i(v_p)$. Then, there should either be a larger *blocking by a lower-priority job* or a *larger interference by higher-priority jobs* than accounted for by $s_i(v_p)$. We divide the proof into two cases depending on whether there is a larger blocking or interference for J_i. We show that there cannot be a larger blocking or interference than what is already included in $s_i(v_p)$.

Case (i): There is a larger blocking for J_i. This means either the processor becomes possibly available later than A_1^{min}, or there exists a job with a lower priority that can execute for longer than what is used in (6). The former contradicts the assumption that A_1^{min} is the exact earliest time at which the processor becomes possibly available. The latter contradicts the assumptions that a lower-priority job cannot execute for longer than C_j^{max} before J_i starts, and that (6) uses the lower-priority job with the largest WCET.

Case (ii): There is a larger interference for J_i. (7) terminates when there is no more higher-priority job that possibly releases at $s_i^{(k-1)}$. If J_i can start later than $s_i(v_p)$, the jobs currently included in $s_i(v_p)$ can execute longer than their WCET, or there must be a higher-priority job that is released before J_i starts but is not included in $s_i(v_p)$. The former is not possible as C_j^{max} is the WCET of a job J_j. The latter means that there is still a J_j with $p_j < p_i$ and $r_j^{min} < s_i(v_p)$. But this contradicts the assumption that (7) has terminated.

Hence, when (7) terminates, $s_i^{(k)}$ is the latest time instant the processor may start scheduling $J_i \in \mathcal{J}^M$ after v_p. □

A second upper bound on the start time of J_i is given by $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$, as proven in Lemma 12.

Lemma 12. *$J_i \in \mathcal{J}^M(v_p)$ starts executing no later than $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$.*

Proof. By contradiction. Assume that the processor starts executing $J_i \in \mathcal{J}^M(v_p)$ later than $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}$, say at $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$, where $x > 0$. According to the definition of $\mathcal{J}^M(v_p)$, no job $J_j \in \mathcal{J} \setminus (\mathcal{J}^M(v_p) \cup \mathcal{J}^P)$ can start executing before all jobs in $\mathcal{J}^M(v_p)$ finish their execution. Hence, the processor is busy executing jobs in $\mathcal{J}^M(v_p) \setminus J_i$ until $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$. Consequently, at least one job in $\mathcal{J}^M(v_p) \setminus J_i$ finishes at $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$. Then, J_i starts at $\overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max} + x$. If J_i executes for its WCET C_i^{max}, it finishes at $\overline{LFT}(\mathcal{J}^M, v_p) + x$. This contradicts Corollary 1 that states that $\overline{LFT}(\mathcal{J}^M, v_p)$ is the latest time instant the processor may be busy executing jobs in $\mathcal{J}^M(v_p)$. □

Combining the two upper bounds provided by Lemmas 11 and 12, we get that the latest start time of any job $J_i \in \mathcal{J}^M(v_p)$ is upper bounded by:

$$\widehat{LST}_i(\mathcal{J}^M, v_p) = \min\{s_i(v_p), \overline{LFT}(\mathcal{J}^M, v_p) - C_i^{max}\} \quad (8)$$

Then, as proven in Lemma 13, the LFT of any job $J_i \in \mathcal{J}^M(v_p)$ is upper bounded by

$$\widehat{LFT}_i(\mathcal{J}^M, v_p) = \widehat{LST}_i(\mathcal{J}^M, v_p) + C_i^{max} \quad (9)$$

Lemma 13. *$J_i \in \mathcal{J}^M(v_p)$ cannot complete its execution later than $\widehat{LFT}_i(\mathcal{J}^M, v_p)$ as defined in (9).*

Proof. J_i cannot start later than $\widehat{LST}_i(\mathcal{J}^M, v_p)$ as it is an upper bound on the latest start time of J_i (by Lemmas 11 and 12). If J_i starts at $\widehat{LST}_i(\mathcal{J}^M, v_p)$, it cannot finish later than $\widehat{LST}_i(\mathcal{J}^M, v_p) + C_i^{max}$ as C_i^{max} is the WCET of J_i. □

Lemma 14. *No job in $\mathcal{J}^M(v_p)$ misses its deadline if* (5) *holds.*

Proof. Lemma 13 shows that $\widehat{LFT}_i(\mathcal{J}^M, v_p)$ is an upper bound on the LFT of $J_i \in \mathcal{J}^M(v_p)$. Hence, if it is smaller than the deadline of J_i, then it is certain that $J_i \in \mathcal{J}^M(v_p)$ cannot miss its deadline. □

G. Computing safe response-time bounds for jobs in a safe reductions set

Section IV-F already describes how to compute a safe upper bound on the LFT of any job $J_i \in \mathcal{J}^M$, so all that remains to derive is a safe lower bound on their EFT. As proven in Lemma 15, the EFT of $J_i \in \mathcal{J}^M$ is lower bounded by

$$\widehat{EFT}_i(\mathcal{J}^M, v_p) = \max\{A_1^{min}(v_p), r_i^{min}\} + C_i^{min} \quad (10)$$

Lemma 15. *$J_i \in \mathcal{J}^M(v_p)$ cannot complete its execution earlier than $\widehat{EFT}_i(\mathcal{J}^M, v_p)$ as defined in* (10).

Proof. J_i cannot start before r_i^{min} as a job cannot start before it is released, and J_i cannot start before A_1^{min} as A_1^{min} is the earliest time at which all jobs before J_i finish. Hence, $\max\{A_1^{min}, r_i^{min}\}$ is a lower bound on the start time of J_i. If J_i starts at $\max\{A_1^{min}, r_i^{min}\}$ it cannot finish before $\max\{A_1^{min}, r_i^{min}\} + C_i^{min}$ as C_i^{min} is the BCET of J_i. □

H. Algorithm to construct a safe reduction set

Algorithm 4 shows how to create the safe reduction set $\mathcal{J}^M$ for a system state v_p. In short, interfering jobs are added to the candidate reduction set $\mathcal{J}^S$ until the POR is either accepted because it is safe, or rejected because it fails the sufficient schedulability test. At line 1 of Algorithm 4, $\mathcal{J}^S$ is initialized with the *direct successors* of v_p, i.e., successors that the original SAG analysis would naturally add immediately after state v_p according to Definition 2.

Dealing with interfering jobs. Through lines 3-5, the interfering job set $\mathcal{J}^I$ is formed. If $\mathcal{J}^I$ is empty (line 6), i.e., there are no interfering jobs for $\mathcal{J}^S$, the algorithm checks whether $\mathcal{J}^S$ passes the sufficient schedulability test (line 7). If so, the POR is safe and $\mathcal{J}^S$ is returned as the safe reduction set (line 8). Otherwise, the empty set is returned (line 10) as the POR of $\mathcal{J}^S$ is unsafe and therefore rejected. If there are interfering jobs (line 11), a $J_x \in \mathcal{J}^I$ is chosen according to input criterion X (line 12) and added to $\mathcal{J}^S$ (line 13).

Criteria to expand the candidate reduction set. The order in which interfering jobs are integrated into $\mathcal{J}^S$ may affect the composition and success of the candidate reduction set at the end of the algorithm. We use a greedy criterion (denoted by X) to move one job $J_x \in \mathcal{J}^I$ from the interfering set $\mathcal{J}^I$ to $\mathcal{J}^S$. In the experimental section of this paper, we adopted a criterion X that always selects the highest-priority job in $\mathcal{J}^I$. After adding J_x to $\mathcal{J}^S$ (line 13) the while-loop repeats, either until the POR is accepted (line 8) or rejected (line 10).

Lemma 16. *A POR of a safe reduction set $\mathcal{J}^M$ as returned by line 8 of Algorithm 4 maintains exact schedulability.*

Proof. According to Corollary 1 and 2, $\overline{EFT}(\mathcal{J}^M, v_p)$ and $\overline{LFT}(\mathcal{J}^M, v_p)$ computed with Algorithms 2 and 3, are exact bounds on the finish time of every job in a safe reduction set $\mathcal{J}^M$. Hence, the reduction of $\mathcal{J}^M$ to a single scheduling decision will not affect the analysis of the execution of jobs not contained in $\mathcal{J}^M$. Furthermore, the reduction will not affect the schedulability analysis of $\mathcal{J}^M$ itself because no $J_i \in \mathcal{J}^M$

Algorithm 4: Safe reduction set construction

```
Input  : System state $v_p$, criterion $X$
Output: The safe reduction set $\mathcal{J}^M(v_p)$
1  $\mathcal{J}^S \leftarrow$ direct successors of $v_p$ (Definition 2);
2  while true do
3      $\mathcal{J}^I \leftarrow \emptyset$;
4      while $\exists J_j \in \mathcal{J} \setminus (\mathcal{J}^P \cup \mathcal{J}^S)$ s.t. $J_j$ is an interfering job
         for $\mathcal{J}^S$ do
5          $\mathcal{J}^I \leftarrow \mathcal{J}^I \cup \{J_j\}$;
6      if $\mathcal{J}^I = \emptyset$ then
7          if $\mathcal{J}^S$ is schedulable according to (5) then
8              return $\mathcal{J}^S$;
9          else
10             return $\emptyset$;
11     else
12         $J_x \leftarrow$ a job in $\mathcal{J}^I$ according to criterion $X$;
13         $\mathcal{J}^S \leftarrow \mathcal{J}^S \cup \{J_x\}$;
```

can miss its deadline (Lemma 14). Thus, if the POR of $\mathcal{J}^M$ is safe, the schedulability analysis remains exact. □

Lemma 17. *A POR of a safe reduction set $\mathcal{J}^M$ as returned by line 8 of Algorithm 4 maintains safe response-time bounds.*

Proof. As the POR of $\mathcal{J}^M$ will not affect the analysis of the execution of jobs not contained in $\mathcal{J}^M$ (Lemma 16), the response-time bounds of these jobs will remain exact as in the original analysis by Nasri et al. [7]. For the jobs contained in $\mathcal{J}^M$, the EFT and LFT are lower and upper bounds respectively (Lemma 15 and 13 respectively). Therefore, the response-time bounds of jobs in $\mathcal{J}^M$ are also safe. □

V. Empirical evaluation

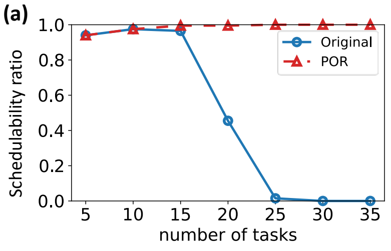
We conducted empirical experiments to answer the following questions: (i) does POR provide a speedup and state-space reduction over the original SAG implementation [7] (referred to as 'original')? and (ii) how is the worst-case response time affected by the partial-order reduction? We considered the problem of analyzing the schedulability of a set of non-preemptive periodic task sets with implicit deadlines on a uniprocessor platform scheduled by a JLFP scheduling policy. We derived the set of jobs of all tasks in one hyperperiod and used it as an input for SAG. We performed our POR-based analysis according to Algorithm 1 and compared it to the *original* SAG analysis of Nasri et al. [7] using their implementation available at [15]. Both analyses were implemented as a single-threaded C++ program.

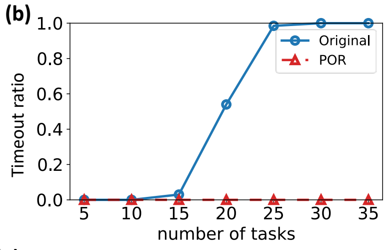
Metrics. Our performance metrics are as follows. **Schedulability ratio** is the ratio of schedulable task sets to all generated task sets (per number of tasks). A task set is deemed unschedulable when either an execution scenario containing a deadline miss is encountered, or a timeout of four hours is reached. **State-reduction ratio** is defined as $(1 - N^P/N^O) \cdot 100\%$, where N^P and N^O are the number of states explored by the POR and the *original* analyses, respectively, for an input job set. The closer this ratio is to 100%, the more states have been *removed* from the state-space by POR in comparison to the original analysis. **Speedup** is the CPU time required by the original analysis to analyze a job set $\mathcal{J}$ divided by the CPU time required by POR to analyze $\mathcal{J}$. **Normalized WCRT** is the WCRT of each task reported by the POR-based analysis divided by the WCRT of the same task reported by the original analysis.

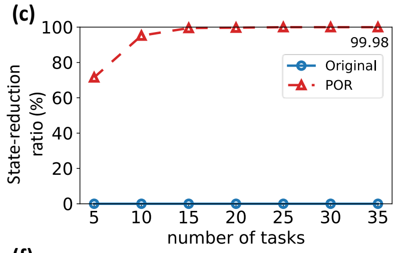
A. Case study

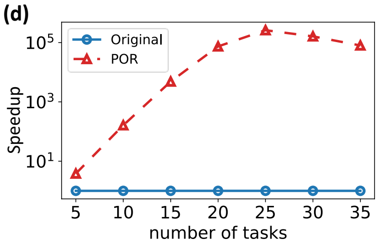
We performed a case study using the APP4MC specification of the automotive use case provided at the WATERS 2017 industrial challenge [16]. We considered the task set assigned to Core 2 (of [16]), which consists of 710 periodic runnables grouped into 7 periodic tasks. We performed two experiments: **(i)** each task is assumed to be non-preemptive, and **(ii)** each runnable is assumed to be non-preemptive but preemptions can happen between the execution of runnables of the same or different tasks. As there was no predefined release jitter in the use case specification, we assumed two cases: no jitter, and jitter equal to the largest WCET among runnables. We evaluated the performance using the priorities specified for each period/task in the model specification [16]. Using the provided clock speed of 200 MHz resulted in inherently unschedulable task sets because the total utilization is larger than 1. Hence, we assumed the first clock speeds where the task sets were schedulable, namely 400 MHz and 1.2 GHz for experiments (i) and (ii), respectively. The experiments were performed on a cluster with AMD EPYC 7H12 processors clocked at 2.6GHz and 256GB RAM.

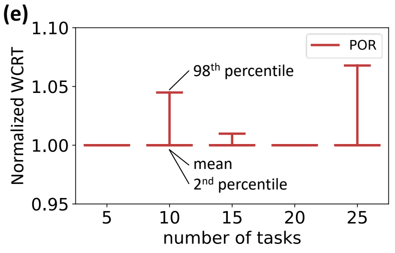
Fig. 4(h) shows a summary of the results. In all but the simplest case, POR provided a speedup over the original analysis, either because it finished faster, or the original analysis never finished due to running out of memory (as is the case for the last row of the table). POR is only outperformed by the original analysis when the system has very few tasks that do not have release jitter (first row of the table) because then there are very few branches in the SAG that can be reduced by POR. In that case, the overhead of performing POR out-weights the small reduction in state-space. When there are many tasks with release jitter (the last row of the table), the original analysis runs out of memory, whereas POR is able to finish in a matter of seconds (i.e., 14 seconds). This is because there are many more jobs interfering with each other when there is release jitter, which gives POR more opportunities to create reduction sets. It is confirmed by the fact that, when there is release jitter, an edge in the graph accounts for 107 jobs on average, whereas there are only about 2.5 jobs per edge when there is no release jitter. Since there are more jobs per edge, POR produces a significantly smaller state space (351 states with jitter vs. 15,261 states without jitter). In conclusion, this case study shows that POR enables the SAG analysis to scale to large industrial use cases made of hundreds of runnables and with more than 37.000 jobs.

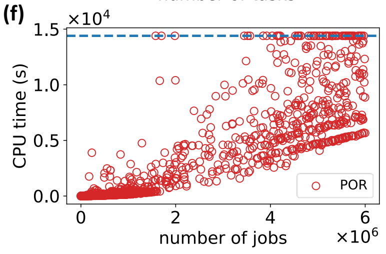
(a)


(b)


(c)


(d)


(e)


(f)


(g)

	Min	2%	Median	Mean	98%	Max
State-reduction ratio	0.329060	0.736457	0.999690	0.985257	0.999991	0.999999
Speedup	0.8311283	3.224060	$2.777467*10^4$	$1.123618*10^5$	$9.289019*10^5$	$6.553969*10^6$
Normalized WCRT	1.000000	1.000000	1.000000	1.001235	1.018518	1.160054

(h)

Preemption	Clock speed	#tasks/ runnables	#jobs per hyperperiod	Jitter	CPU time (s)		#states		Average #jobs per edge	Speedup	State-reduction ratio (%)
					Original	POR	Original	POR			
Task-level	1.2 GHz ($U \approx 0.18$)	7	786	No	0.00538	0.00574	809	766	1.027	**0.937**	**5.315**
				Yes	0.00951	0.00441	1248	586	1.344	**2.157**	**53.045**
Runnable-level	400 MHz ($U \approx 0.53$)	710	37,459	No	782.044	20.968	1,076,504	15,261	2.455	**37.298**	**98.582**
				Yes	Out of memory	13.996	N/A	351	107.026	**N/A**	**N/A**

Fig. 4: Experimental results for performance on synthetic task sets. (a) Schedulability ratio. (b) Timeout ratio. (c) State-reduction ratio. (d) Speedup. (e) Distribution of normalized WCRT. (f) Scalability in terms of CPU time. (g) Information about the distribution of the state-reduction ratio, speedup, and normalized WCRT diagrams (for all generated task sets). (h) Case study results.

B. Empirical results for synthetic task sets

To evaluate the scalability of POR, we generate random synthetic task sets with n periodic tasks and a total utilization of U, we follow the method of Emberson et al. [17]; we generate periods following a log-uniform distribution in the range $[10, 100]$ms with a granularity of 5ms. We generate n task-utilization values with a total sum of U with Rand-FixedSum [18]. Using periods and task-utilization values, we obtain the WCET of each task. The BCET is set to 0 for all tasks (i.e., we maximize execution-time variation). We assumed a release jitter of 100µs for each job of a task in the hyperperiod. We assume rate-monotonic priorities. In the following experiments, we set $U = 0.3$ and varied the number of tasks per task set to evaluate the impact of the number of tasks (scalability). Choosing $U = 0.3$ ensures generating mostly schedulable task sets which tend to increase the runtime of POR and SAG analyses. Finally, we generated 200 random task sets for each parameter value. The experiments were performed on a cluster with Intel Xeon E5-2690 v3 processors clocked at 2.6GHz and 64GB RAM.

For the performance evaluation, task sets with more than 50,000 jobs per hyperperiod were discarded to limit the runtimes, as otherwise we would not be able to compare our solution against the original SAG that was susceptible to state-space explosions. For the scalability evaluation (Fig. 4(f)), task sets were limited to 6,000,000 jobs per hyperperiod.

Performance results. Fig. 4(a) shows a large difference between the schedulability ratio of the analyses due to the large number of timeouts after 4 hours in the original analysis, as can be seen in Fig. 4(b). In these experiments, the POR-based analysis was able to explore the entire state-space within 5.2 seconds on average. This shows that POR allows task sets with more uncertainties and more tasks to be analyzed before the timeout is reached.

Fig. 4(c) shows that the state-reduction ratio for POR increases with an increase in the number of tasks showing that it becomes more successful on larger task sets because then POR has more opportunities to form bigger reduction sets. The average state-reduction ratio is 98.53%, demonstrating that POR is able to remove the vast majority of states from the state-space compared to the original analysis.

Fig. 4(d) shows that POR provides a significant speedup over the original analysis for any number of tasks per task set, with an average speedup of 1.12×10^5. Fig. 4(e) shows a box plot of the normalized WCRT for task sets for which neither analyses timed out. The whiskers represent the 2nd and 98th percentiles. Observe that the normalized WCRT is very close to 1, with an average of 1.001. This shows that the WCRT reported by our POR-based analysis is only a slight over-approximation of the true WCRT.

Scalability results. Fig. 4(f) shows that the runtime of our analysis increases with the number of jobs per hyperperiod. Although there are some timeouts from $1.5 \cdot 10^6$ jobs per hyperperiod onward, the vast majority of task sets are analyzed within four hours. This experiment shows that POR provides a significant increase in the scalability of SAG-based analyses even to task sets with $6 \cdot 10^6$ jobs per hyperperiod.

VI. Conclusion

In this work, we applied partial-order reduction (POR) to one of the most recent reachability-based response-time analyses called schedule-abstraction graph (SAG). As a first step, we used POR for the analysis of independent tasks scheduled non-preemptively on a single-core platform. POR limits the combinatorial exploration all scheduling decisions when the order in which they are taken does not affect the schedulability of the system. This allows us to improve the scalability of the SAG analysis without jeopardizing its soundness and with barely any added pessimism on the WCRT estimates.

Our empirical evaluation shows that our POR was able to reduce the runtime by *five orders of magnitude* and the number of explored states by 98% compared to the original SAG analysis. Our solution could analyse a large automotive use case made of hundreds of runnables and tens of thousands of jobs in a matter of seconds when the original analysis failed to finish due to running out of memory. Lastly, tests on synthetic task sets showed that our new solution scales to $6 \cdot 10^6$ jobs per hyperperiod, whereas the original analysis already fails to finish at $5 \cdot 10^4$ jobs per hyperperiod. This shows that POR allows us to analyze significantly more complex task sets (with more tasks and more jobs) than the original analysis and has the potential to be used in industrial design-space exploration tools.

The foundation laid out in this paper opens the path to extending the POR technique to all the scheduling strategies, workload models, and execution platforms supported by the schedule-abstraction graph framework, including limited-preemptive parallel tasks, tasks accessing shared resources, multiprocessor platforms, and global scheduling policies.

Acknowledgment

This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative and the TRANSACT project (101007260 ECSLxxx).

References

[1] P. Ekberg, “Rate-Monotonic Schedulability of Implicit-Deadline Tasks is NP-hard Beyond Liu and Layland’s Bound,” in *RTSS*, 2020, pp. 308–318.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, “Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling,” *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[3] R. Davis, A. Burns, R. Bril, and J. Lukkien, “Controller Area Network (CAN) schedulability analysis : refuted, revisited and revised,” *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[4] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu, “Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking,” in *SEUS*, 2007, p. 263–272.

[5] Y. Sun and G. Lipari, “A Pre-Order Relation for Exact Schedulability Test of Sporadic Tasks on Multiprocessor Global Fixed-Priority Scheduling,” *Real-Time Systems*, vol. 52, no. 3, p. 323–355, 2016.

[6] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg, “An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks,” in *DATE*, 2019, pp. 1228–1233.

[7] M. Nasri and B. B. Brandenburg, “An Exact and Sustainable Analysis of Non-preemptive Scheduling,” in *RTSS*, 2017, pp. 12–23.

[8] M. Nasri, G. Nelissen, and B. B. Brandenburg, “A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling,” in *ECRTS*, 2018, pp. 9:1–9:23.

[9] M. Nasri, G. Nelissen, and B. B. Brandenburg, “Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling,” in *ECRTS*, 2019, pp. 21:1–21:23.

[10] S. Nogd, G. Nelissen, M. Nasri, and B. B. Brandenburg, “Response-Time Analysis for Non-Preemptive Global Scheduling with FIFO Spin Locks,” in *RTSS*, 2020, pp. 115–127.

[11] S. Baruah, A. Mok, and L. Rosier, “Preemptively scheduling hard-real-time sporadic tasks on one processor,” in *RTSS*, 1990, pp. 182–190.

[12] K. Jeffay, D. Stanat, and C. Martel, “On non-preemptive scheduling of period and sporadic tasks,” in *RTSS*, 1991, pp. 129–139.

[13] S. Srinivasan, G. Nelissen, and R. J. Bril, “Work-in-Progress: Analysis of TSN Time-aware Shapers using Schedule Abstraction Graphs,” in *RTSS*, 2021, accepted for publication.

[14] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, “An Empirical Survey-based Study into Industry Practice in Real-time Systems,” in *RTSS*, 2020, pp. 3–11.

[15] “NP Schedulability Test,” https://github.com/gnelissen/np-schedulability-analysis, 2021.

[16] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, “WATERS Industrial Challenge 2017,” in *WATERS*, 2017.

[17] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the synthesis of multiprocessor tasksets,” in *WATERS*, 2010, pp. 6–11.

[18] R. Stafford. (2006) Random vectors with fixed sum. [Online]. Available: https://nl.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum