Reachability-Based Response-Time Analysis of Preemptive Tasks Under Global Scheduling

Pourya Gohari
Eindhoven University of Technology (TU/e), The Netherlands

Jeroen Voeten
Eindhoven University of Technology (TU/e), The Netherlands

Mitra Nasri
Eindhoven University of Technology (TU/e), The Netherlands

Abstract

Global scheduling reduces the average response times as it can use the available computing cores more efficiently for scheduling ready tasks. However, this flexibility poses challenges in accurately quantifying interference scenarios, often resulting in either conservative response-time analyses or scalability issues. In this paper, we present a new response-time analysis for preemptive periodic tasks (or job sets) subject to release jitter under global *job-level fixed-priority* (JLFP) scheduling. Our analysis relies on the notion of schedule-abstraction graph (SAG), a reachability-based response-time analysis known for its potential accuracy and efficiency. Up to this point, SAG was limited to non-preemptive tasks due to the complexity of handling preemption when the number of preemptions and the moments they occur are not known beforehand. In this paper, we introduce the concept of time partitions and demonstrate how it facilitates the extension of SAG for preemptive tasks. Moreover, our paper provides the first response-time analysis for the global EDF(k) policy – a JLFP scheduling policy introduced in 2003 to address the Dhall's effect.

Our experiments show that our analysis is significantly more accurate compared to the state-of-the-art analyses. For example, we identify *12 times more schedulable task sets* than existing tests for the global EDF policy (e.g., for systems with 6 to 16 tasks, 70% utilization, and 4 cores) with an average runtime of 30 minutes. We show that EDF(k) outperforms global RM and EDF by scheduling on average 24.9% more task sets (e.g., for systems with 2 to 10 cores and 70% utilization). Moreover, for the first time, we show that global JLFP scheduling policies (particularly, global EDF(k)) are able to schedule task sets that are not schedulable using well-known partitioning heuristics.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Scheduling

Keywords and phrases Response-time analysis, global scheduling, preemptive, job-level fixed-priority scheduling policy, multicore, schedule-abstraction graph

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.3

Supplementary Material *Software (Source Code)*: https://github.com/SAG-org

Funding This work used the Dutch national e-infrastructure with the support of the SURF Cooperative using grant no. EINF-8595 and the EU ECSEL Joint Undertaking under grant agreement no. 101007260 (project TRANSACT).

1 Introduction

Response-time analysis is crucial for the design and certification of real-time systems, providing insights into task scheduling and execution. With 80% of industrial real-time systems using multicore platforms [1,2], there is a rising need for accurate response-time analyses for global scheduling policies (e.g., global FP, EDF). Supported by prominent operating systems (Linux, RTEMS, VxWorks, ...), global scheduling better utilizes the computing cores, reduces average response times, and improves load balancing.

© Pourya Gohari, Jeroen Voeten, and Mitra Nasri;
licensed under Creative Commons License CC-BY 4.0

However, due to the difficulty of quantifying task interference under global preemptive scheduling, the existing response-time analyses are either very pessimistic (provide loose response-time bounds) [11,26,39] or do not scale well (due to exhausting usage of CPU or memory resources) [7,14,15,25]. This results in missed opportunities to improve system utilization, and may considerably increase the cost and duration of the system design and development phases by discarding task sets (or configurations) that meet the timing requirements. These costs can be mitigated through the use of more accurate analyses that have reasonable runtime (e.g., take a couple of hours but not a day or week to complete).

The above roadblock has also significantly obstructed the development of response-time analyses for other global *job-level fixed-priority* (JLFP) scheduling policies such as EDF(k) and EDF-US[$m/(2m-1)$]. These were introduced decades ago [24,37] to address the Dhall's effect and are known to have a better utilization bound than global EDF. Currently, there exists no response-time analysis for these policies, preventing a comprehensive understanding of their capabilities. This question remains true even for global EDF and FP given the recent discovery of Biondi et al. [13] that shows most of the current schedulability tests for these policies cannot identify any task set that is schedulable under global scheduling but not under partitioned scheduling. Our work fills all these gaps by introducing a highly accurate response-time analysis for JLFP policies.

Related work. Sufficient schedulability tests for global scheduling have been around since two decades ago [3,24]. Most of these analyses are based on the concept of busy-window and the quantification of carry-in workload [5,6,8,10,26,29,41,43,45,46]. However, as we will show in Sec. 5, these analyses tend to be pessimistic, particularly for larger systems (with more tasks or cores). The literature also includes exact schedulability tests using time automata [25], linear hybrid time automata [40,42], simulation-based tests [18], reachability analysis using timed-labeled transition systems [23], or reachability-based tests [7,14–16]. However, despite their steady progress, these analyses still struggle to scale beyond small system sizes and task parameters (as we will show in Sec. 5).

The *schedule-abstraction graph* (SAG) technique is a recent reachability-based response-time analysis introduced by Nasri et al. [30–32,35]. It systematically explores the decision space of a job-level fixed-priority (JLFP) scheduling policy when scheduling a set of non-preemptive jobs (or periodic tasks) on cores of a multicore platform. This exploration involves constructing a directed acyclic graph (DAG) with vertices denoting reachable system states after job execution and edges representing scheduling decisions that evolve system states. By automatically and accurately identifying interference scenarios, SAG provides very tight bounds on the response-time of each job (task) for problems such as global non-preemptive scheduling of sequential tasks [31], parallel DAG tasks [32], tasks that use shared resources protected by FIFO and priority-based spin locks [34], scheduling of moldable gang tasks [33], multi-rate task graphs [21], fault-tolerant tasks [22], and an analysis of FIFO and time-aware shapers in time-sensitive networking (TSN) [38]. For instance, SAG identifies *4.3 times more schedulable* tasks than other busy-window-based analyses [36] for systems with 16 cores and 10 parallel DAG tasks. In comparison to exact schedulability tests such as [44] in UPPAAL, SAG has been more than *three-orders of magnitude* faster (e.g., for systems with 4 cores and 15 tasks, which were the largest that [44] could analyze in a 4-hour time budget) [32].

However, until now, all extensions of SAG have been confined to non-preemptive jobs (or periodic tasks) due to the intricacy of (i) handling preemptions when higher-priority jobs have uncertain release times, (ii) abstracting schedules when the number of preemptions and the moments they happen are not known a priori, and (iii) merging or pruning states that include an inconsistent number of completed jobs.

This paper. We introduce a novel response-time analysis for global preemptive job-level fixed-priority scheduling policies by extending SAG to support preemption, addressing the challenges outlined above. Our key contributions are:

- constructively extending the schedule-abstraction technique to analyze preemptive periodic tasks (or arbitrary job sets) that are subject to release jitter (which, to our knowledge, has not been addressed before in the literature);
- providing the first response-time analysis for EDF(k) policy [24] showing that it indeed outperforms global RM and EDF.

Our empirical evaluations show that our analysis identifies considerably more globally schedulable task sets than the state-of-the-art sufficient tests [11,24,26,41,43] for both G-EDF and G-RM scheduling policies and scales much better than the state of the art exact schedulability tests such as [9,15,17,18,23,25]. Our analysis outperforms the best sufficient test for G-EDF by identifying 12 times more schedulable task sets, with an average runtime of 40 minutes (e.g., for systems with 6 to 16 tasks, 70% utilization, and 4 cores).

Furthermore, we show that our analysis for G-EDF(k), G-EDF, and G-RM is able to find globally schedulable task sets that are not schedulable by the well-known partitioning heuristics. For example, for a system with 4 cores, 6 tasks, and 90% utilization, our analysis identifies 37 task sets (out of 200) that are only schedulable by global EDF(k) but not any partitioning strategy.

2 Models and assumptions

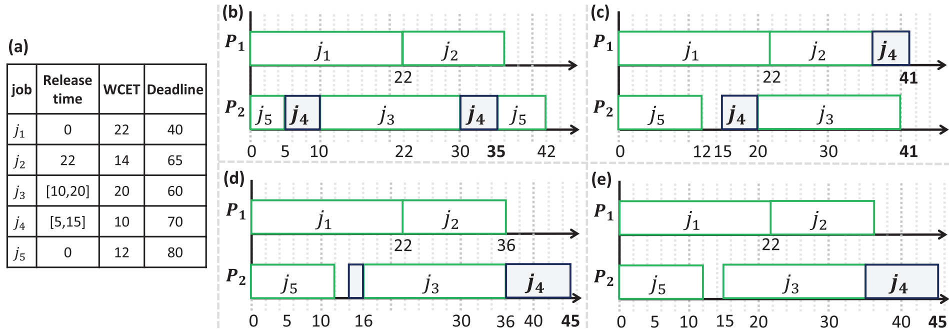
We consider the problem of obtaining response time bounds for a finite set of independent preemptive jobs $\mathcal{J}$ with arbitrary release times on a homogeneous multicore platform with m cores. Jobs in the job set can have an arbitrary arrival model. For example, they can represent jobs of periodic tasks within one hyperperiod (see Sec. 2.1 for a detailed discussion).

A job $j \in \mathcal{J}$ is characterized by its earliest-release time $r^{min}(j)$ (a.k.a. *arrival time* in Audsely's terminology [4]), latest-release time $r^{max}(j)$, absolute deadline $d(j)$, best-case execution time (BCET) $C^{min}(j)$[1], worst-case execution time (WCET) $C^{max}(j)$, and priority $P(j)$ which is assigned by the underlying scheduling policy. Also, without loss of generality, we assume that the job timing parameters are integer multiples of the system clock.

We consider a global work-conserving job-level fixed-priority (JLFP) scheduling policy, which includes policies such as the earliest-deadline first (EDF) and fixed-priority (FP). For example, under EDF policy, the priority of a job $P(j)$ is equal to its absolute deadline. We consider that a lower numerical value for $P(j)$ indicates a higher priority and that priority ties are broken in an arbitrary yet consistent manner. We assume the "$<$" operator implicitly uses this tie-breaking rule.

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. Moreover, we use $\min_\infty\{X\}$ and $\max_0\{X\}$ with a set of positive integers X to indicate that if $X \neq \emptyset$ then $\max_0\{X\} = \max\{X\}$, otherwise, $\max_0\{X\} = 0$ and similarly, if $X \neq \emptyset$ then $\min_\infty\{X\} = \min\{X\}$, otherwise, $\min_\infty\{X\} = \infty$.

[1] As shown by Ha and Liu [27], preemptive scheduling is sustainable w.r.t. shortening the execution time. Therefore, if a job executes with an execution time shorter than its WCET, it cannot create a scenario resulting in the worst-case response time. In our analysis, BCET is used to calculate the best-case response time for a job. If deriving the best-case response time is not in the interest of the user, then the BCET can be assumed to be equal to the WCET.



job	Release time	WCET	Deadline
j_1	0	22	40
j_2	22	14	65
j_3	[10,20]	20	60
j_4	[5,15]	10	70
j_5	0	12	80

Figure 1 The impact of release jitter on the response time, **(a)** the job set to be scheduled by global EDF on two cores, **(b)** a schedule assuming all jobs are released as early as possible, **(c)** a schedule assuming all jobs are released as late as possible, **(d)** a schedule assuming j_3 is released at time 16 and j_4 is released as late as possible, **(e)** a scenario formed by a combination of the earliest and latest release time of jobs that can derive the worst-case response-time for job j_4.

2.1 From periodic tasks to a job set

As mentioned earlier, the input of our analysis is a set of jobs, which can be formed from various arrival models. Recent surveys [1,2] show that periodic activation is the most prevalent arrival model in the industry and is present in more than 80% of industrial systems. Here, we elaborate on the process of constructing a job set from a set of synchronous periodic tasks.

The arrival pattern of synchronous periodic tasks repeats after a hyperperiod, the least-common multiple of the periods (denoted by H). For a safe analysis, the job set must include all task instances (jobs) released by each periodic task within the hyperperiod.

Let $T(\tau)$ represent the period, $C^{min}(\tau)$ and $C^{max}(\tau)$ denote the BCET and WCET, $D(\tau)$ stands for the deadline (assuming $D(\tau) \leq T(\tau)$), $P(\tau)$ denote the priority, and $\sigma(\tau)$ denote the release jitter of a task τ. For a given task τ, we generate $n(\tau) = H/T(\tau)$ jobs within the interval $[0, H)$. Let j be the kth job ($1 \leq k \leq n(\tau)$) of task τ. Then $r^{min}(j) = (k-1) \cdot T(\tau)$ $r^{max}(j) = (k-1) \cdot T(\tau) + \sigma(\tau)$, $d(j) = (k-1) \cdot T(\tau) + D(\tau)$, $C^{min}(j) = C^{min}(\tau)$, and $C^{max}(j) = C^{max}(\tau)$. If the scheduling policy is fixed-priority, then job $P(j) = P(\tau)$. If the policy is EDF, then $P(j) = d(j)$. It is worth noting that according to Audsley's definitions, the job's absolute deadline is calculated based on its arrival time, i.e., $r^{min}(j) + D(\tau)$. Therefore, a job that is released late due to release jitter does not get a longer deadline.

Assuming constrained-deadline tasks, if the job set is schedulable, there's no carry-out workload to the next hyperperiod. Our SAG-based analysis automatically explores schedules generated from any permissible release time of the jobs in a hyperperiod, therefore, it does not need to analyze more than one hyperperiod to conclude the schedulability of the job set.

3 Motivation and key ideas

Due to the release jitter of the jobs, the exact times at which a higher-priority job preempts a lower-priority job is not known a priori. Moreover, as shown by Gohari et al. [23], simply considering the minimum or maximum release times for all jobs in a job set does not provide a safe bound on the worst-case response-time of the jobs.

We elaborate on the findings of [23] to explain the key idea behind our response-time analysis. In the example shown in Fig. 1, jobs j_3 and j_4 have release jitter. Figs. 1b and c show the corresponding global EDF schedule of the job set when all jobs are released as early and as late as possible, respectively. Neither of these scenarios results in the worst-case

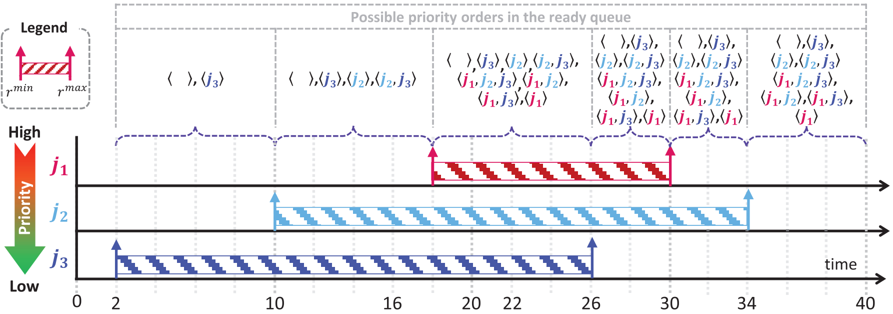


Figure 2 Possible states of the ready queue in the presence of release jitter.

response time (WCRT) of j_4 as such case happens when j_3 is released somewhere in the middle of its release-time interval, e.g., at time 16 (see Fig. 1d). This observation gives rise to a question: *How can WCRT be quantified without considering all possible job release times?*

Our key observation. There is no need to consider all release instants of each job but only those that can potentially change the priority ordering of jobs in the ready queue. This is elaborated in Fig. 2, where we have shown the release intervals of three jobs in the bottom-half and the possible states of the ready queue in the top-half. For example, during the interval $[2, 10)$, the ready queue may have two states: is empty (if j_3 releases after time 10) or is $\langle j_3 \rangle$ (otherwise). Similarly, during the interval $[10, 18)$, the ready queue can be $\langle\rangle$, $\langle j_2 \rangle$, $\langle j_3 \rangle$, and $\langle j_2, j_3 \rangle$. As we see, such a set of possible ready queues may change only at the boundary of intervals separated by the r^{min} and r^{max} of the jobs, namely: $[2, 10)$, $[10, 18)$, $[18, 26)$, $[26, 30)$, $[30, 34)$, and $[34, \infty)$. Such a partitioning of the analysis window allows us to ignore individual release instants and only focus on the set of possible ready queues at the beginning of each interval (we call it a *time partition*). Applying this notion to the example in Fig. 1, we will have the following time partitions $[0, 5)$, $[5, 10)$, $[10, 15)$, $[15, 20)$, $[20, 22)$, and $[22, \infty)$. The set of potential ready queues in the partition $[15, 20)$ is $\langle j_4 \rangle$ and $\langle j_3, j_4 \rangle$. Among these, the latter ready queue can result in an execution scenario where j_3 interferes with j_4 for the longest time and results in the scenario shown in Fig. 1e.

▶ **Definition 1** (Time partition). *A time partition, denoted as $[a, b)$, is defined as an interval of time where $a \in \{0, r^{min}(j), r^{max}(j) | j \in \mathcal{J}\}$ and $b = \min_\infty\{\{r^{min}(j) | j \in \mathcal{J} \wedge a < r^{min}(j)\} \cup \{r^{max}(j) | j \in \mathcal{J} \wedge a < r^{max}(j)\}\}$.*

Later in Sec. 4, we expand upon this concept by introducing *priority-aware time partitions* that consider both the current system state and the priority of not-yet-dispatched jobs.

Scalability challenge. Even though the above time partitioning can considerably reduce the search space, is not yet enough to design an efficient analysis because despite the fact that the number of time partitions is linear to the number of jobs (i.e., it is $2 \cdot |\mathcal{J}|$), the number of possible ready queues in each partition can be exponentially large because it depends on the number of all possible subsets of a set with X elements, where X is a function of the number of overlapping release and execution intervals of jobs in the system.

Efficient exploration of schedules in a schedule-abstraction graph (SAG). SAG efficiently captures all schedules of a non-preemptive job set (with release jitter) by constructing a reachability graph representing possible dispatch orderings under a JLFP policy [32]. Each

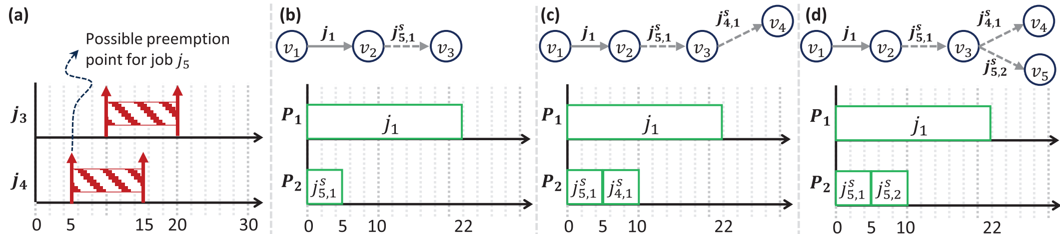


Figure 3 An example of conservative execution of the jobs. **(a)** The release interval for j_3 and j_4, **(b)** dispatching the first segment of job j_5 before the possible release time of higher priority jobs, **(c)** a scenario in which job j_5 is preempted by job j_4, **(d)** a scenario where job j_5 continue its execution without preemption at time 5.

system state maintains the availability of the cores in the form of m *uncertainty intervals*, each representing the earliest and latest time by which x cores become simultaneously free (where x is one, two, ..., or m).

SAG is built following a breadth-first strategy by iteratively expanding and then merging states. During expansion, it chooses a state with the least number of dispatched jobs and identifies all jobs that may have a chance to be a direct successor of the state, namely, to be the head of a ready queue that can be created from that state. To do so, for each not-yet-dispatched job j, it evaluates whether j has a chance to become the highest-priority job *before any other job*. This is done by assessing an eligibility condition, namely, the earliest time at which j can start its execution in state v should be smaller than the time at which a JLFP policy will no longer allow j to be dispatched on a core before another job.

When a non-preemptive job is dispatched, it won't reappear in the ready queue of subsequent states, while a preemptive job can return to the ready queue when it gets preempted and therefore potentially interfere with not-yet-dispatched jobs in future states.

Challenges in extending SAG to preemptive job sets. Supporting preemptive execution in SAG faces a fundamental challenge: determining the remaining execution time of a preempted job without knowing the exact release time of a potentially preempting higher-priority job. Efficiently accounting for remaining execution times is crucial for accurately quantifying potential future interference generated by preempted jobs when they resume their execution in future states. It directly influences analysis accuracy.

The second challenge arises in the merging phase. In the original SAG, states with identical sets of dispatched jobs could be merged. However, under preemptive scheduling, states will also carry preempted jobs (with different remaining execution times in different states). This challenges the design of safe merging rules that maintain soundness without compromising analysis accuracy. Without state merging (or other forms of state-space reduction), our solution faces scalability issues similar to other exact analyses in the literature.

How to accurately keep track of the remaining execution times? Scalability of our solution stems from the observation we made about the impact of time partitions on the potential preemption scenarios in Fig. 2. To accurately capture scenarios where a job could be preempted by a higher-priority job, we pause the exploration during the expansion phase of a state at the boundaries of priority-aware time partitions as they may lead to preemption by a higher-priority job. Such scenarios will be explored in separate (future) states.

To illustrate this concept, let's revisit the example in Fig. 1a. Jobs j_1 and j_5 are released at time zero. Given that j_1 has the highest priority and is released at time zero, it is the first to be dispatched by the JLFP policy. Next, the scheduler will dispatch j_5 as the second

core is available at time zero. However, job j_5 may later be preempted by two potential higher-priority jobs, j_3 and j_4 (shown in Fig. 3a). To accurately track the remaining execution time of j_5 under these potential preemptions, we virtually break job j_5 into two segments: one that executes from time 0 to 5 without preemption (denoted by $j^s_{5,1}$), and one that represents the remaining execution time of j_5 after time 5 (when job j_4 might be released). In our new SAG, the expansion phase will only look up to time 5 for job j_5 and leave the remaining scenarios to the subsequent states as shown in Fig. 3b. Consequently, there are two possible scenarios in state v_3: (i) job j_5 continues, which in that case, it will be segmented again because of reaching the boundary of the next time partition (i.e., $[5, 10)$) at time 10 as shown in Fig. 3d, or (ii) job j_4 starts its execution and executes its first segment (before it could potentially be preempted by j_3) as shown in Fig. 3c. Our SAG automatically obtains the earliest and latest start time of j_4 in state v_3, taking into account scenarios where j_4 might be released later than its earliest release time and before time 10.

4 Response-time analysis

Our response-time analysis builds a schedule-abstraction graph (SAG) for preemptive job sets, as motivated in Sec. 3. SAG systematically explores the space of JLFP schedules for the given job set, considering uncertainties in the release times of the jobs. These schedules are represented in a graph $G = (V, E)$, where V denotes the set of reachable system states and E represents the set of scheduling decisions (when a JLFP policy dispatches a job on a core or resumes its execution). Following convention [32], we assume that the system starts with an initial state v_1, where all cores are idle and no job has started executing yet.

4.1 System state representation

Definition of a system state plays a key role in the accuracy and scalability of a reachability analysis like SAG. To accurately monitor the remaining execution times of preempted jobs, our state definition incorporates crucial elements beyond [32]. We not only store the core availability intervals, but also a set of completed jobs, a set of preempted jobs, the earliest and the latest finish time of the latest segment of preempted jobs, and the remaining execution times of preempted jobs:

- **Core availability intervals.** Following [32], we utilize the symmetry properties of identical cores to represent their availability through m *uncertainty intervals* denoting the times at which one, two, three, ..., and m cores simultaneously become available. For instance, an interval $A_x(v) = [A^{min}_x(v), A^{max}_x(v)]$ $(1 \leq x \leq m)$ indicates the earliest $(A^{min}_x(v))$ and latest $(A^{max}_x(v))$ times at which x cores are simultaneously available.
- **Set of completed jobs.** We store the set of jobs completed before reaching state v in $\mathcal{J}^{Co}(v)$. At any state v, $\mathcal{J} \setminus \mathcal{J}^{Co}(v)$ indicates jobs that have not yet been dispatched, or have been preempted, and therefore, may appear in a ready queue in that state.
- **Set of preempted jobs.** We keep track of jobs that have started but not yet completed (preempted) in the set $\mathcal{J}^{Pr}(v)$. In any state v, $\mathcal{J}^{Pr}(v) \subseteq \mathcal{J} \setminus \mathcal{J}^{Co}(v)$.
- **Finish times of preempted jobs.** As discussed in Sec. 3, we assume that a potential preemption may split a job into virtual segments (one for before a preemption and one for after). Namely, for each preempted job $j \in \mathcal{J}^{Pr}(v)$, we record the possible finish time of the latest executed segment of the job using $FT(j, v) = [FT^{min}(j, v), FT^{max}(j, v)]$ under the assumption that the job will be preempted, where $FT^{min}(j, v)$ and $FT^{max}(j, v)$ are the earliest and latest finish times of that segment

- **Remaining execution times of preempted jobs.** We record the remaining execution time of each preempted job $j \in \mathcal{J}^{Pr}(v)$ as $RM(j,v) = [RM^{min}(j,v), RM^{max}(j,v)]$, where $RM^{min}(j,v)$ and $RM^{max}(j,v)$ indicate the remaining execution time of job j in state v in the best and worst case, respectively.

4.2 Edges of the graph

In our preemption-aware SAG, state transitions are depicted in the graph through directed edges labeled with either a job or a job segment. As it will be discussed in Sec. 4.4, when a job j crosses the boundary of a time partition in which a higher-priority job may release (Definition 2), we divide it into two segments: one segment before the partition boundary, and another segment after the partition boundary. The second segment represents the remaining execution time of the job after a potential preemption at the partition boundary. An edge $e \in E$ from state v to v' with label j or j^s indicates that executing the whole job j or a segment j^s of job j in state v evolves it to a new state v'.

4.3 Schedule-abstraction graph generation

We build our SAG graph in an iterative manner following a breadth-first strategy as shown in Algorithm 1. Each iteration of the algorithm (lines 3-19) has two phases: the *expansion phase* (lines 5-19) and the *merge phase* (lines 16-19). In the **expansion phase**, a leaf state v with the fewest completed jobs is selected for expansion (line 4). For every such job j that could potentially be scheduled directly after state v (line 6), a new state v' is added to the graph (line 7) and is connected to v via a directed edge labeled with j (line 13) or a segment of job j^s in case it could be preempted (line 15). In the **merge phase**, we check if the newly created state v' can be merged with previously generated states, thereby reducing the overall graph states when their future can be explored together. Sec. 4.5 provides details of the merge phase. Moreover, Algorithm 1 keeps track of the best- and worst-case response time of each completed job in arrays BR and WR, respectively (lines 9 and 11).

4.4 Expansion phase

Next, we explain how to identify an *eligible job/segment*[2] – a job or segment with the potential to be scheduled by the JLFP scheduler in state v, and therefore, be a direct successor of state v. First, we determine when a job will be ready (Sec. 4.4.1). Then we explain how to check if it can be the highest-priority job in any of the possible ready queues of state v without constructing those ready queues (Sec. 4.4.2 and 4.4.3). If the job is eligible to be a direct successor of state v (Sec. 4.4.4), we determine its earliest and latest finish times (Sec. 4.4.5). Before proceeding to the expansion phase, we establish several key definitions to enhance clarity and facilitate understanding in this section. We denote the set of not-yet-scheduled jobs with a higher priority than a job j in a state v by

$$\mathcal{J}^{hp}(j,v) \triangleq \{j' \mid j' \in \mathcal{J} \setminus (\mathcal{J}^{Co}(v) \cup \mathcal{J}^{Pr}(v)) \land P(j') < P(j)\}. \tag{1}$$

▶ **Definition 2** (Priority-aware time partition). *A priority-aware time partition, denoted as $[a,b)$ for a given job j and state v, is an interval of time where $a \in \{0, r^{min}(j'), r^{max}(j') | j' \in \mathcal{J}^{hp}(j,v)\}$ and $b = \min_\infty\{\{r^{min}(j') | j' \in \mathcal{J}^{hp}(j,v) \land a < r^{min}(j')\} \cup \{r^{max}(j') | j' \in \mathcal{J}^{hp}(j,v) \land a < r^{max}(j')\}\}$.*

---

[2] In the remainder of the paper, for convenience, we use the notation "job/segment" to refer to a job j when $j \notin \mathcal{J}^{Pr}$, or to a segment of it (j^s) if it has been already preempted.

Algorithm 1 Construction of the SAG for job set $\mathcal{J}$.

Input: Job set $\mathcal{J}$
Output: Bounds on the BCRT and WCRT of every job in $\mathcal{J}$

```
1  ∀j ∈ J, BR(j) ← ∞, WR(j) ← 0;
2  Initialize G by adding v1 = ({[0,0],...,[0,0]}, {}, {}, {});
3  while ∃ a leaf state such that J^Co ≠ J do
4      v ← select the leaf state with minimum |J^Co|;
5      foreach job j ∈ J \ J^Co do
6          if j can be dispatched after v according to Eq. (11) then
7              Build v' using Algorithm 2;
8              if RM^min(j, v') = 0 then // job is possibly completed
9                  BR(j) ← min{EFT(j, v) − r^min(j), BR(j)};
10             if RM^max(j, v') = 0 then // job is certainly completed
11                 WR(j) ← max{LFT(j, v) − r^min(j), WR(j)} ;
12             if j ∉ J^Pr(v) and j ∉ J^Pr(v') then
13                 Connect v to v' with an edge labeled j;
14             else
15                 Connect v to v' with an edge labeled j^s;
16             while ∃ a leaf state v'' such that Rule 1 is satisfied for v' and v'' do
17                 Merge v' and v'' in v_m using Eqs. (18)-(20);
18                 Redirect all incoming edges of v' and v'' to v_m;
19                 Remove v' and v'' from G;
20 return BR(j), WR(j) for all j ∈ J;
```

4.4.1 Ready interval

A job j is ready only if it has been released, and if it is previously preempted, then its preceding segment has been completed. Since jobs may have release jitter, the exact time at which they become ready is unknown. Therefore, in a system state v, we determine a lower bound on the time when job j is possibly ready, denoted by $R^{min}(j, v)$, and an upper bound on the time when job j is certainly ready, denoted by $R^{max}(j, v)$. These can be obtained using r^{mim} and r^{max} if the job has not been dispatched/preempted before or using the finish time of the last segment of the job in the current state v ($FT^{min}(j, v)$ and $FT^{max}(j, v)$ which we will define later in Sec. 4.4.5 and Algorithm 2) if it has a dispatched/preempted segment:

$$R^{min}(j, v) \triangleq \begin{cases} r^{min} & \text{if } j \notin \mathcal{J}^{Pr}(v) \\ FT^{min}(j, v) & \text{otherwise,} \end{cases} \tag{2}$$

$$R^{max}(j, v) \triangleq \begin{cases} r^{max} & \text{if } j \notin \mathcal{J}^{Pr}(v) \\ FT^{max}(j, v) & \text{otherwise} \end{cases} \tag{3}$$

Following the above discussions, it is trivial to prove that in state v, job j (or the next segment of it) will not be ready prior to $R^{min}(j, v)$ and will certainly be ready at $R^{max}(j, v)$.

4.4.2 Earliest start time

By definition, the scheduling policy considers dispatching a job j in state v *only if* (i) it is ready and (ii) a core is available. The earliest time at which job j can be ready is at $R^{min}(j, v)$, and the earliest time at which one core becomes available is at $A_1^{min}(v)$ (from our state definition). Therefore, the earliest time when both of the above conditions hold is

$$EST(j, v) = \max\{R^{min}(j, v), A_1^{min}(v)\}. \tag{4}$$

▶ **Lemma 3.** *A JLFP policy will not consider dispatching a job j (or a segment of it) in state v before time $EST(j, v)$.*

Proof. The proof trivially follows the above discussion. ◀

4.4.3 Latest start time

As mentioned in Sec. 3, we want to impose exploration at the boundary of a priority-aware timing partition of job j in state v, where it may be preempted in that partition. We implement this idea by defining a concept called *partial* latest start time (P-LST), denoted by $LST(j, v)$, for each job j (or its remaining segment) in a state v.

▶ **Definition 4** (Partial latest start time – P-LST). *Time t represents the P-LST of a job/segment j in state v if and only if it indicates the latest time at which j can start execution under a work-conserving JLFP scheduling policy in state v without passing the boundary of a priority-aware time partition of job j in state v.*

The first upper bound on the $LST(j, v)$ is obtained from the work-conserving property of the scheduler, namely, such scheduler dispatches 'a' job/segment as soon as a job/segment is ready and a core is available. The latest time at which both of these conditions hold is the maximum between $A_1^{max}(v)$ (the latest time at which at least one core becomes available) and $R^{max}(j', v)$ (for all incomplete jobs j' in state v). We denote this time by $t_{wc}(v)$:

$$t_{wc}(v) \triangleq \max\left\{A_1^{max}(v), \min_{\infty}\{R^{max}(j', v) \mid j' \in \mathcal{J} \setminus \mathcal{J}^{Co}(v)\}\right\}. \tag{5}$$

▶ **Lemma 5.** *A work-conserving scheduler will not dispatch a job $j \in \mathcal{J} \setminus \mathcal{J}^{Co}(v)$ directly after state v at any time later than $t_{wc}(v)$.*

Proof. Assume that $t_{wc}(v) \neq \infty$ because otherwise, the claim is trivial. At time $t_{wc}(v)$, a core is certainly available (since $t_{wc}(v) \geq A_1^{max}(v)$) and a not-yet-completed job $j' \in \mathcal{J} \setminus \mathcal{J}^{Co}(v)$ becomes certainly ready, namely, $(t_{wc}(v) \geq min_{\infty}\{R^{max}(j', v) \mid j' \in \mathcal{J} \setminus \mathcal{J}^{Co}(v)\})$. Therefore, a work-conserving scheduler will dispatch job j' at $t_{wc}(v)$. Thus, to be a direct successor of v, job j must be scheduled no later than $t_{wc}(v)$. ◀

We derive the second upper bound on the $LST(j, v)$ from the JLFP property of the scheduling policy, namely, a JLFP policy always dispatches the highest-priority ready job/segment. Therefore, the earliest time at which it is certain that job j can no longer become the highest-priority ready job in state v, provides an upper bound on $LST(j, v)$. This notion is called $t_{high}(j, v)$ and was originally introduced by Nasri et al. [31,32].

$$t_{high}(j, v) \triangleq \min_{\infty}\left\{R^{max}(j', v) \mid j' \in \mathcal{J} \setminus \mathcal{J}^{Co}(v) \ \wedge\ P(j') < P(j)\right\}. \tag{6}$$

▶ **Lemma 6.** *Job/segment j will not be the highest-priority job in state v at any time later than $t_{high}(j, v) - 1$.*

Proof. Suppose that $t_{high}(j, v) \neq \infty$; otherwise j is not constrained by any higher-priority job and the claim trivially holds. At the time $t_{high}(j, v)$, a job/segment j' with a higher priority than j is certainly ready (released and its previous segment is certainly finished). Therefore, job j cannot be a direct successor of state v from time $t_{high}(j, v)$ onward as it will no longer be the highest-priority job in the ready queue. ◀

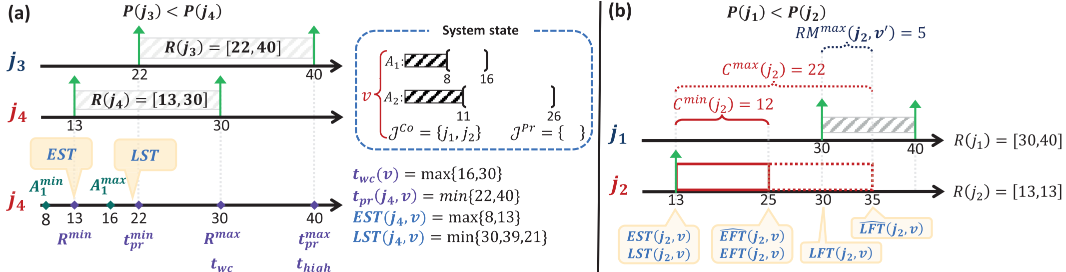


Figure 4 **(a)** an example of computing the earliest and latest start time for a job, **(b)** an example of computing partial earliest and latest finish times for a job.

The final upper bound on $LST(j, v)$ comes from the earliest priority-aware time-partition of j in state v. From Definition 2, we know that such time-partition starts with the earliest or latest release time of a not-yet-scheduled higher-priority job than j and finishes with the earliest or latest release time of another higher-priority job. Using $\mathcal{J}^{hp}(v, j)$, we obtain $t_{pr}(j, v)$ which is the starting point of such time partition:

$$t_{pr}(j, v) \triangleq \min\left\{t_{pr}^{min}(j, v), t_{pr}^{max}(j, v)\right\}, \tag{7}$$

$$t_{pr}^{min}(j, v) \triangleq \min_{\infty}\left\{r^{min}(j') \mid j' \in \mathcal{J}^{hp}(j, v) \wedge EST(j, v) < r^{min}(j')\right\}, \tag{8}$$

$$t_{pr}^{max}(j, v) \triangleq \min_{\infty}\left\{r^{max}(j') \mid j' \in \mathcal{J}^{hp}(j, v) \wedge EST(j, v) < r^{max}(j')\right\}. \tag{9}$$

▶ **Lemma 7.** *The start time of job j passes the boundary of a priority-aware time partition (in which a higher-priority job than j can be released) at any time later than $t_{pr}(j, v) - 1$.*

Proof. Suppose that $t_{pr}(j, v) \neq \infty$; otherwise $\mathcal{J}^{hp}(v, j) = \emptyset$, and thus, the claim trivially holds. Following Definition 2, a priority-aware time partition of j in state v contains higher-priority jobs that may release and preempt j, starts at the earliest (r^{min}) or latest release time (r^{max}) of not-yet-scheduled higher-priority jobs than j, namely, $\mathcal{J}^{hp}(v, j)$. Thus, the beginning of the next time partition after $EST(j, v)$ is the earliest time that a job in $\mathcal{J}^{hp}(v, j)$ can possibly or certainly be released, i.e., it is $\min\{t_{pr}^{min}(j, v), t_{pr}^{max}(j, v)\}$ (Eq. (7)). ◀

Following Definition 4, we derive the partial latest start time $LST(j, v)$ using three upper bounds: $LST(j, v) \leq t_{wc}(v)$ (Lemma 5), $LST(j, v) \leq t_{high}(j, v) - 1$ (Lemma 6), and $LST(j, v) \leq t_{pr}(j, v) - 1$ (Lemma 7). Hence, the partial latest start time for job j is given by

$$LST(j, v) = \min\left\{t_{wc}(v), t_{high}(j, v) - 1, t_{pr}(j, v) - 1\right\}. \tag{10}$$

▶ **Lemma 8.** *Under a JLFP policy, job j will not start executing as a successor of state v after $LST(j, v)$ without starting its execution in the priority-aware time partition of j in v.*

Proof. The proof trivially follows Definition 4, Lemma 5, Lemma 6, and Lemma 7. ◀

Fig. 4a shows a state v in SAG that is being expanded by job j_4. Here, $EST(j_4, v)$ is 13 (the maximum between $A_1^{min} = 8$ and $r^{min}(j_4) = 13$). $t_{wc}(v)$ is 30 as it is the earliest time at which a core is certainly available ($A_1^{max} = 16$) and a job is certainly released ($r^{max}(j_4) = 30$). $t_{high}(j_4, v)$ is 40 as it is the earliest time at which a higher-priority job than job j_4 is certainly released ($r^{max}(j_3) = 40$). Finally, $t_{pr}(j_4, v)$ is 22 which is the time from which job j_3 may preempt j_4. Putting all together, the $LST(j_4, v)$ is 21.

4.4.4 Eligibility condition

▶ **Lemma 9.** *Job/segment j can be dispatched by the JLFP scheduler in state v only if*

$$EST(j,v) \leq LST(j,v). \tag{11}$$

Proof. Proof by contradiction. Let's assume that $EST(j,v) > LST(j,v)$ and the scheduler dispatches job j in state v (before any other job). From Lemma 3, we know that $EST(j,v)$ is the earliest time that job j can start its execution in state v. Now, we analyze the three cases that $EST(j,v)$ can be larger than $LST(j,v)$:

Case 1 ($EST(j,v) > t_{wc}(v)$): From Lemma 5 we know that at time $t_{wc}(v)$ a not-yet-completed job j' becomes certainly ready and a core becomes certainly available. As a result, job j' will be dispatched by a work-conserving scheduler instead of job j.

Case 2 ($EST(j,v) \geq t_{high}(j,v)$): From Lemma 6 we know that a higher-priority job is certainly ready at time $t_{high}(j,v)$. If $EST(j,v) \geq t_{high}(j,v)$, the JLFP scheduler will not consider dispatching j before that higher-priority job in state v.

Case 3 ($EST(j,v) \geq t_{pr}(j,v)$): From Eqs. (8) and (9) we know that $t_{pr}(j,v)$ is strictly larger than $EST(j,v)$, therefore, this case is not possible.

The above discussion concludes that $EST(j,v) > LST(j,v)$ leads to a contradiction, namely, job j cannot be dispatched in state v and thus cannot be a direct successor of it. ◀

4.4.5 Earliest and latest finish times

After checking the eligibility of a job/segment that requires obtaining its potential EST and P-LST, we derive the corresponding earliest and latest finish times of the job taking into account the boundaries of the time partitions that could potentially lead to a preemption (if $t_{pr}(j,v) < \infty$). To make this section easier to follow, we use a running example illustrated in Fig. 4b. We also assume the EST and P-LST of job j_2 are equal to 13, and it has not been split into segments before.

As mentioned in Sec. 3 (in Fig. 3), we split a job j in state v into segments at the boundary of a time partition that contains a higher-priority job, i.e., at time $t_{pr}(j,v)$, because there might be a scenario in which j is preempted in that time partition. To check whether such a scenario can happen, we first obtain a preemption-free version of the finish time of the job and then check if the job may finish later than $t_{pr}(j,v)$. We denote the preemption-free earliest and latest finish times by $\widehat{EFT(j,v)}$ and $\widehat{LFT(j,v)}$, respectively. We consider two cases to obtain each of them. If the job has not been divided into segments before, we use its BCET and WCET, otherwise, we use its remaining execution time (maintained in the state):

$$\widehat{EFT(j,v)} = \begin{cases} EST(j,v) + C^{min}(j) & j \notin \mathcal{J}^{Pr}(v), \\ EST(j,v) + RM^{min}(j,v) & \text{otherwise} \end{cases} \tag{12}$$

$$\widehat{LFT(j,v)} = \begin{cases} LST(j,v) + C^{max}(j) & j \notin \mathcal{J}^{Pr}(v), \\ LST(j,v) + RM^{max}(j,v) & \text{otherwise} \end{cases} \tag{13}$$

where $RM^{min}(j,v)$ and $RM^{max}(j,v)$ are the best- and worst-case remaining execution times of job j that are stored in state v (later, we will discuss them in Eqs. (16) and (17)).

In the example of Fig. 4b, the preemption-free earliest finish time of j_2 ($\widehat{EFT(j_2,v)}$) is 12 units of time after its earliest start time at time 25 and similarly, its preemption-free latest finish time ($\widehat{LFT(j_2,v)}$) will be 22 units of time after its latest start time at time 35.

Now, we use the preemption-free latest finish time of the job $\widehat{LFT(j,v)}$ to assess whether there is a chance that the job is preempted by a higher-priority job arriving at or after $t_{pr}(v,j)$. If $t_{pr}(j,v) < \widehat{LFT(j,v)}$, the higher-priority job might be released before job j finishes and therefore we impose exploration in $t_{pr}(j,v)$ (namely, we construct the state resulted from the dispatching of job j in such a way that the higher-priority job be an eligible job in that state, and therefore, its interference is accounted for the response time of j in the states that follow). Otherwise, either there is no higher-priority job ($t_{pr}(j,v) = \infty$) or it will be released after the latest finish time of job j. In such cases, the preemption-free earliest and latest finish times are the same as partial earliest and latest finish times, which denote the earliest and latest finish times of the job/segment without passing $t_{pr}(j,v)$.

In Fig. 4b, the earliest time that job j_2 crosses time partition $[30, 40)$ is at time $t_{pr}(j_2, v) = 30$. Since the preemption-free earliest finish time of job j_2 at time 25 is smaller than $t_{pr}(j_2, v)$, there are scenarios in which j_2 can finish without preemption, therefore, $EFT(j_2, v) = 25$. On the other hand, $\widehat{LFT(j_2,v)} = 35$ which is larger than $t_{pr}(j_2, v) = 30$. Here, we impose an exploration by considering that job j_2 will be divided into two segments, and the partial latest finish time for the first segment is 30. Later, we explain how to obtain the remaining execution time of j_2 (to be dispatched in the future, possibly after being preempted by j_1).

In summary, the partial earliest and latest finish time for a job j are derived when the preemption-free finish times are contrasted against $t_{pr}(j,v)$ as follows:

$$EFT(j,v) = \begin{cases} \widehat{EFT(j,v)} & \widehat{EFT(j,v)} \le t_{pr}(j,v), \\ t_{pr}(j,v) & \text{otherwise} \end{cases} \tag{14}$$

$$LFT(j,v) = \begin{cases} \widehat{LFT(j,v)} & \widehat{LFT(j,v)} \le t_{pr}(j,v), \\ t_{pr}(j,v) & \text{otherwise} \end{cases} \tag{15}$$

Next, we obtain the execution time of the *remaining* segment of job j (if it may not finish before $t_{pr}(j,v)$) for the new state v'. To do so, we calculate the difference between the preemption-free finish time $[\widehat{EFT(j,v)}, \widehat{LFT(j,v)}]$ and the partial finish time $[EFT(j,v), LFT(j,v)]$:

$$RM^{min}(j,v') = \widehat{EFT(j,v)} - EFT(j,v), \tag{16}$$

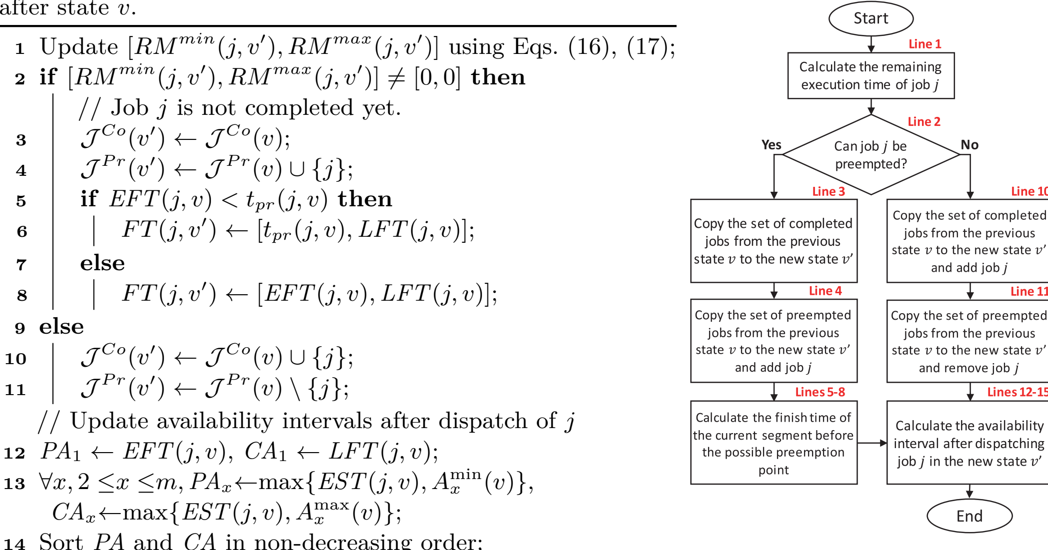
$$RM^{max}(j,v') = \widehat{LFT(j,v)} - LFT(j,v). \tag{17}$$

In Fig. 4b, $\widehat{EFT(j_2,v)} = 25$ which is smaller than $t_{pr}(j_2,v) = 30$. As a result, $RM^{min}(j_2,v') = 0$. Conversely, $\widehat{LFT(j_2,v)} = 35 > t_{pr}(j_2,v)$, therefore $RM^{max}(j_2,v') = 5$.

4.4.6 Creating a new state

When a job/segment $j \in \mathcal{J} \setminus \mathcal{J}^{Co}$ satisfies inequality (11), it can be dispatched as a direct successor of state v. For each such job/segment, a new node v' is created in SAG which represents the new state of the system after dispatching j as outlined in Algorithm 2 and depicted in the flowchart presented in Fig. 5. First, Line 1 obtains the remaining execution time for the dispatched job/segment using Eqs. (16) and (17). If any execution time remains for the dispatched job/segment, it indicates that this job/segment can cross the boundary of the time partition. Therefore, the job is added to the preempted job set and its finish time is updated in the new state (Lines 5-8). Otherwise, the job has finished and we add it to the completed jobs and remove it from the preempted job set in the new state (Lines 9-11). To update the finish time of the segment, we have to consider that the job can only be preempted

Algorithm 2 Create a new state v' by dispatching job j after state v.



1 Update $[RM^{min}(j,v'), RM^{max}(j,v')]$ using Eqs. (16), (17);
2 **if** $[RM^{min}(j,v'), RM^{max}(j,v')] \neq [0,0]$ **then**
  // Job j is not completed yet.
3 $\mathcal{J}^{Co}(v') \leftarrow \mathcal{J}^{Co}(v)$;
4 $\mathcal{J}^{Pr}(v') \leftarrow \mathcal{J}^{Pr}(v) \cup \{j\}$;
5 **if** $EFT(j,v) < t_{pr}(j,v)$ **then**
6   $FT(j,v') \leftarrow [t_{pr}(j,v), LFT(j,v)]$;
7 **else**
8   $FT(j,v') \leftarrow [EFT(j,v), LFT(j,v)]$;
9 **else**
10 $\mathcal{J}^{Co}(v') \leftarrow \mathcal{J}^{Co}(v) \cup \{j\}$;
11 $\mathcal{J}^{Pr}(v') \leftarrow \mathcal{J}^{Pr}(v) \setminus \{j\}$;
 // Update availability intervals after dispatch of j
12 $PA_1 \leftarrow EFT(j,v),\ CA_1 \leftarrow LFT(j,v)$;
13 $\forall x, 2 \le x \le m, PA_x \leftarrow \max\{EST(j,v), A_x^{\min}(v)\}$, $CA_x \leftarrow \max\{EST(j,v), A_x^{\max}(v)\}$;
14 Sort PA and CA in non-decreasing order;
15 $\forall x, 1 \le x \le m, A_x(v') \leftarrow [PA_x, CA_x]$;

Figure 5 The flowchart of Alg. 2.

if it reaches $t_{pr}(j,v)$. If so, we consider the earliest finish time for a case that it can certainly be preempted and finishes at $t_{pr}(j,v)$ (lines 5-6). Finally, we update the availability intervals for the new state in lines 12-15 following [32] (as we use the same technique to represent the availability of cores in each state).

▶ **Corollary 10.** *After dispatching a lower-priority job j right after state v that makes a new state v', there is certainly a core available at time $t_{pr}(j,v)$ in the new state v'.*

Proof. From Lemma 8, we know that job j certainly starts before $t_{pr}(j,v)$. Moreover, from Eqs. (15) and (14), we know that job j is enforced to finish its execution before $t_{pr}(j,v)$. Finally, since the availability intervals are updated based on the finish time of the dispatched job (line 12 in Algorithm 2), we conclude that a core is certainly available at or before time $t_{pr}(v,j)$ in the new state v'. ◀

4.5 State space reduction

To reduce the number of states in the graph, we introduce a merge rule to combine states whose future can be explored together.

▶ **Rule 1** (Merge rule). *Two states v and v' can be merged if $\mathcal{J}^{Co}(v) = \mathcal{J}^{Co}(v')$, and $\mathcal{J}^{Pr}(v) = \mathcal{J}^{Pr}(v')$, and $\forall j \in \mathcal{J}^{Pr}, FT(j,v) \cap FT(j,v') \neq \varnothing$, and $\forall x, 1 \le x \le m, A_x(v) \cap A_x(v') \neq \varnothing$.*

When two states v and v' are merged to v'', each availability interval $A(v'')$, is constructed such that it includes any core-availability scenario which was possible in state v or v'. Hence

$$A_x(v'') = \left[\min\{A_x^{\min}(v), A_x^{\min}(v')\},\ \max\{A_x^{\max}(v), A_x^{\max}(v')\}\right]. \tag{18}$$

Eq. (18) ensures that x cores become possibly available in the merged state v'' only when x cores become potentially available in either of the original states v or v', and x cores are certainly available in v'' only when x cores are certainly available in both v and v'.

Moreover, the finish time interval of each segment of each job j in v'' is obtained in such a way that it includes the finish time intervals of the jobs derived previously for all execution scenarios that lead to either v or v', and thus to the merged state v''. Namely, any potential finish time (or potential remaining execution time) of the job j in state v or state v' is a potential finish time (or potential remaining execution time) for the job in state v'':

$$FT(j,v'') = \left[\min\{FT^{\min}(j,v), FT^{\min}(j,v')\},\ \max\{FT^{\max}(j,v), FT^{\max}(j,v')\}\right], \quad (19)$$

$$RM(j,v'') = [\min\{RM^{\min}(j,v), RM^{\min}(j,v')\}, \max\{RM^{\max}(j,v), RM^{\max}(j,v')\}]. \quad (20)$$

▶ **Lemma 11.** *Merging two states v and v' according to Rule 1 and Eqs.* (18)*,* (19)*, and* (20) *does not remove any potential reachable state from the graph.*

Proof. **First**, Rule 1 ensures that the set of completed jobs ($\mathcal{J}^{Co}$) and preempted jobs ($\mathcal{J}^{Pr}$) that have been dispatched on the path to v and v' are identical. Therefore, the set of jobs and segments that still need to execute in the merged state v'' is the same as v and v'. **Second**, according to Eq. (18), for any time $t \in A_x(v)$ or $t' \in A_x(v')$ $(1 \leq x \leq m)$, it is possible to find a $t'' \in A_x(v'')$ at which x cores become simultaneously available because $A_x^{min}(v'')$ is the minimum among $A_x^{min}(v)$ and $A_x^{min}(v')$ and $A_x^{max}(v'')$ is the maximum among $A_x^{max}(v)$ and $A_x^{max}(v')$. **Third**, according to Eqs. (19) and (20), for any job $j \in \mathcal{J}^{Pr}(v)$ (which is the same as $\mathcal{J}^{Pr}(v')$ and $\mathcal{J}^{Pr}(v'')$ due to Rule 1), and for any time $t \in FT(j,v)$ (or $t' \in FT(j,v')$) that represent potential finish time of the job in states v (or v'), it is possible for job j to finish its execution at time t (or t') in state v'' (similar to the Second fact). Similarly, for any time $t \in RM(j,v)$ (or $t' \in RM(j,v')$) that represents the potential remaining execution time of the job in states v (or v'), it is possible for job j to have a potential remaining execution time equal to t (or t') in state v''. ◀

4.6 Correctness

To prove the soundness of our analysis, we first show that for each job j that can be preempted by a higher-priority job j' in any execution scenario that leads to the best- or worst-case response time of j, there exists a path in the graph generated by Algorithm 1 that accounts for the interference of j' on the execution of j (Lemma 12). Then we put all the pieces together in Theorem 13 and establish soundness.

▶ **Lemma 12.** *If a job j can be preempted by a higher-priority job j', there is a path $\langle v_1, \ldots, v, v', v''\rangle$ in the SAG such that j^s (a segment of job j) is the label of the edge connecting v to v' and j' is the label of the edge connecting v' to v'' that captures this execution scenario.*

Proof. Under JLFP scheduling, a higher-priority job j' can only preempt a lower-priority job j at time t if (i) j is executing at time t, (ii) j' is the highest-priority ready job at time t, and (iii) all the cores are busy at time t. Now, we demonstrate that our SAG comprehensively explores all scenarios wherein a higher-priority job j' releases during the execution of j. From Eqs. (14) and (15) we know that the exploration and execution of a lower-priority job always will be interrupted at time $t_{pr}(j,v)$ which is the earliest time that a higher-priority job j' releases during the execution of j (as shown by Corollary 10).

Next, we establish that j' is eligible to be dispatched in state v' (by showing that $EST(j',v') \leq t_{pr}(j,v) \leq LST(j',v')$) using the following facts.

Fact 1: From Corollary 10, we know a core is available at $t_{pr}(j,v)$. According to Eq. (7) and the fact that j' was the job with the smallest r^{min} or r^{max} in $\mathcal{J}^{hp}(j,v)$, it must be true that $r^{min}(j') \leq t_{pr}(j,v)$. Therefore, $EST(j',v') \leq t_{pr}(j,v)$ as the EST is the maximum between the time at which a core is available and $r^{min}(j')$.

Fact 2: Eq. (6) indicates that t_{high} is the earliest time at which a higher-priority job is released or a higher-priority segment finishes its execution. In the former case, it follows that $t_{pr}(j,v) \leq t_{pr}^{max}(j,v) \leq t_{high}(j,v)$. In the latter case, as indicated by Eqs. (14) and (15), a segment that has not completed ($\notin \mathcal{J}^{Co}(v)$) finishes execution when a higher-priority job is released, implying another higher-priority job is released at t_{high}. This leads to a contradiction because we assumed that the earliest time that a higher-priority job than j can release is at time $t_{pr}(j,v)$. Therefore, the first scenario always holds, concluding $t_{pr}(j,v) \leq t_{high}(j,v)$.

Fact 3: Since job j' has a higher priority than j, $t_{high}(j,v) \leq t_{high}(j',v')$ because the set of jobs that have higher priority than j' also have higher priority than j. Combining this with Fact 1, we conclude that $t_{pr}(j,v) \leq t_{high}(j,v) \leq t_{high}(j',v')$.

Fact 4: $t_{wc}(j')$ is at least as large as $t_{pr}(j,v)$ because when creating the new state v', we enforce a core to be certainly available at $t_{pr}(j,v)$ (Corollary 10). From Fact 1 we know that the upper bound on $EST(j',v')$ is $t_{pr}(j,v)$ and from Facts 2, 3, 4 we conclude that the lower bound on $LST(j',v')$ is $t_{pr}(j,v)$. Therefore, $EST(j',v') \leq t_{pr}(j,v) \leq LST(j',v')$. Namely, j' can be dispatched in v' and preempt j. ◀

▶ **Theorem 13.** *For any execution scenario such that job j finishes at time t, there is a path $\langle v_1,\ldots,v,v'\rangle$ in the SAG such that j (or j^s) is the label of the edge connecting v to v' while $RM^{min}(j,v')=0$, and $t \in [EFT(j,v), LFT(j,v)]$.*

Proof. Initially, assume that the path $\langle v_1,\ldots,v\rangle$ respects the claim for all jobs/segments dispatched before j in the execution scenario that led j to finish at time t. Moreover, assume that **(i)** the availability intervals safely model the actual availability times, **(ii)** for each $j' \in \mathcal{J}^{Pr}(v)$, the interval $[FT^{min}(j',v), FT^{max}(j',v)]$ safely lower- and upper-bounds (i.e., contains) the completion time of the latest segment of j'. **(iii)** for each $j' \in \mathcal{J}^{Pr}(v)$, the interval $[RM^{min}(j',v), RM^{max}(j',v)]$ safely lower- and upper-bounds (i.e., contains) the remaining execution time of job j'. **(iv)** $\mathcal{J}^{Co}(v)$ correctly includes a subset of the jobs that are completed before j is dispatched, **(v)** $\mathcal{J}^{Pr}(v)$ correctly includes a subset of the jobs that are preempted before j is dispatched.

Now, we prove that there exists a vertex v' that is directly connected to v with an edge labeled j (or j^s) that all five requirements (i)-(v) hold for state v, and that the interval $[EFT(j,v), LFT(j,v)]$ contains the completion time of j. Under the inductive assumption stated above, Lemma 3 and Lemma 8 prove that $EST(j,v)$ and $LST(j,v)$ are safe lower- and upper-bounds on the preemption-free start time of j, respectively. Moreover, as we assume a job/segment executes non-preemptively until $t_{pr}(j,v)$, Eq. (14), and Eq. (15) provide a safe lower and an upper bound on the completion time of job/segment j, respectively. Further, by the inductive assumption, the condition of Lemma 9 must hold for job/segment j, and Algorithm 1 expands the graph with a new node v'. Moreover, as shown by Lemma 8 in [32], the availability intervals of v'' correctly bound the simultaneous availability of x cores for all $1 \leq x \leq m$. The remaining execution times are safely derived from Eqs. (16), (17) taking into account possible preemption at the possible preemption point ($t_{pr}(j,v)$) where a higher-priority not-yet-released job j^h can preempt job/segment j. Moreover, according to Lemma 12 job j^h can possibly preempt job j and become a direct successor of v'. The set of completed jobs and the set of preempted jobs in state v' are updated in lines 3 -4, and lines 10-11 of Algorithm 2 according to the remaining execution time of job j. Therefore, requirements (iv) and (v) are trivially respected. Because a job with possible zero remaining execution time ($RM^{min}(j,v')=0$) can be completed at any time within

$[EFT(j, v), LFT(j, v)]$, thus $EFT(j, v)$ and $LFT(j, v)$ are safe lower- and upper-bounds on t (i.e., proving that $t \in [EFT(j, v), LFT(j, v)]$). Also, according to Lemma 11, potentially merging v' with another state in Algorithm 1 maintains the validity of claims (i) to (v) for the newly generated state even if it is merged with another state.

Finally, given that in the initial state v_1, no job has been dispatched yet and all cores are available (satisfying (i)) and that the set of completed jobs and preempted jobs are empty (satisfying (ii)-(v)), the inductive assumptions holds for v_1, and thus, by induction, are respected in all the states created by Algorithm 1. ◀

5 Empirical evaluation

We compare our response-time analysis against the state of the art in terms of schedulability and runtime, considering global rate monotonic (RM), EDF, and EDF(k) policies.

Baselines. Exact schedulability tests: (i) simulating the G-EDF schedule of periodic tasks (without jitter) using SimSo [17] (SimSo-EDF-Pr), (ii) UPPAAL-based test for periodic tasks scheduled by G-RM [25] (UPPAAL-RM-Pr), (iii) the exact test of Burmyakov et al. [15] for sporadic tasks under G-RM (Burmyakov-RM-Sp), (iv) the exact test of Baruah et al. [9] for sporadic tasks under partitioned EDF (P-EDF-Sp), where, tasks were sorted by their deadlines in a non-decreasing order and then partitioned using the best strategy among best-fit (BF), first-fit (FF), and worst-fit (WF) placement heuristics.

Sufficient schedulability tests: (i) Guan et al. [26] test for sporadic tasks under G-RM (Guan-RM-Sp), (ii) Sun et al. [43] test for sporadic tasks under G-RM (Sun-RM-Sp), (iii) Bertogna et al. [11] test for sporadic tasks under G-EDF (Bertogna-EDF-Sp), (iv) Sun and Lipari [41] test for sporadic tasks under G-EDF (Sun-EDF-Sp), (v) the utilization bound test of Goossens et al. [24] for periodic tasks under G-EDF(k) (Goossens-EDF(k)-Pr).

Remarks. The sporadic task model is not compatible with periodic tasks with release jitter (because jitter reduces the inter-arrival time of the tasks). Therefore, to have a meaningful comparison between these models, we adjusted the minimum inter-arrival time of sporadic tasks to be equal to the period minus jitter. Moreover, given that the schedulability tests of Gohari et al. [23] and Cucu-Grosjean et al. [18] (designed for periodic tasks without jitter) give the same output as SimSo-EDF-Pr, we omitted them from the diagrams. Moreover, we could not compare against Zhou et al. [46] due to not having access to the code of the analysis, the complexity of the algorithms, and some ambiguities. We couldn't be sure if our implementation was faithful to [46]. It is worth mentioning that Zhou's analysis provides a 5% improvement in the schedulability ratio compared to Sun-EDF-Sp across a range of experiments reported in [46].

Execution platform. All methods were executed as a single-threaded C++ program on a computing cluster powered by AMD Rome 7H12 processors clocked at 2.6GHz with 1TB of memory. The runtime of each analysis was reported by measuring its CPU time.

5.1 Comparison against exact tests

We conduct two experiments to assess the impact of release jitter and utilization on the efficiency and accuracy of the schedulability tests. Due to scalability constraints in exact tests and UPPAAL's limit of values under 32,767, our first set of experiments were restricted to $n = 10$ tasks and $m = 4$ cores (Sect. 5.2 provides experiments on broader ranges of

parameters). For each task set, periods were chosen from the set $\{x \times 10^3 \mid 1 \leq x \leq 32\}$ with the log-uniform distribution (following Emberson's method [20]). We then used *UUnifast-Discard* [12,19] to generate a series of random utilizations that collectively sum up to the target system utilization for each task set. Each experiment consists of generating 200 task sets per data point, with the total utilization U ranging from 50% to 90%. Also, each test is allocated a time budget of 10 hours and a memory budget of 256GB.

Exp1 (jitter = 0%). In this experiment, we considered periodic tasks without release jitter. Fig. 6a shows the impact of system utilization on the schedulability of each analysis. As expected, as utilization increases, schedulability decreases, yet, our analysis for G-EDF (SAG-EDF-Pr) yields identical results as the exact analysis (here, SimSo-EDF-Pr) while it outperforms all other sufficient tests for G-EDF. Namely, our analysis identified 63% of task sets as schedulable, whereas Bertogna's and Sun's tests for G-EDF could only detect 20.9% and 31.1% of schedulable task sets, respectively.

Moreover, overall for G-RM, our analysis (SAG-RM-Pr) identified 61.1% schedulable task sets whereas Guan's and Sun's tests for G-RM could only detect 37.1% of schedulable task sets. Also, the exact test UPPAAL-RM-Pr could only identify 40.3% of schedulable task sets (as it was inconclusive for many task sets due to timeouts or memory overflows shown in Fig. 6c). Furthermore, the Burmyakov-RM-Sp exact test failed to complete any task set within the allocated 10 hours time budget and was unable to detect any schedulable task set across all system utilization levels.

We additionally used our JLFP-compatible analysis to evaluate the schedulability of the global EDF(k) policy (SAG-EDF(k)-Pr). As illustrated in Fig. 6a, thanks to our analysis, it is now evident that global EDF(k) significantly outperforms other global policies such as G-RM and G-EDF. Moreover, our analysis identifies, 8 times more schedulable task sets than the existing test for EDF(k) Goossens-EDF(k)-Pr.

In this experiment, partitioned scheduling P-EDF-Sp identifies on average, 77% of task sets as schedulable. The gap between the best global policy (i.e., EDF(k)) and partitioned scheduling appears only when utilization is 80%. This gap is not large, i.e., in 90% utilization, partitioned scheduling can schedule 11.5% more task sets than EDF(k). We believe this difference is due to the lack of an optimal priority assignment policy for global scheduling.

Fig. 6b shows the average runtime of each test. As expected, the sufficient tests (Bertogna-EDF-Sp, Sun-EDF-Sp, Guan-RM-Sp, Goossens-EDF(k)-Pr, and Sun-RM-Sp) are notably faster than others. Our analysis average runtime for G-EDF and G-RM is 5.12 seconds and 2.90 seconds, respectively, which is comparable to the simulation-based test in SimSo, averaging 7.55 seconds to complete. Our analysis does not have any timeouts while UPPAAL-RM-Pr runs out of memory for many schedulable task sets (e.g., for the utilization of 50%, 60%, or 70%).

Exp2 (jitter = 2.5%×period). In this experiment, we consider that each task has a release jitter as large as 2.5% times its period. Given that the exact tests SimSo-EDF-Pr and UPPAAL-RM-Pr do not support release jitter, we exclude them from the experiments.

As illustrated in Fig. 6d, our analysis consistently outperforms all the other tests for both G-RM and G-EDF policies. It identifies 55.5% of task sets as schedulable with G-EDF, which is 2.9 times more than Bertogna's and 2 times more than Sun's tests for G-EDF (marking 19.2% and 27.1% of task sets as schedulable). Our analysis for G-RM identified 53.4% of task sets as schedulable which is 1.5 times more than Guan-RM-Sp and Sun-RM-Sp (marking 34.5% of task sets as schedulable). Moreover, as shown in Fig. 6f, the exact test Burmyakov-RM-Sp did not finish within 10 hours time budget for any of the task sets.

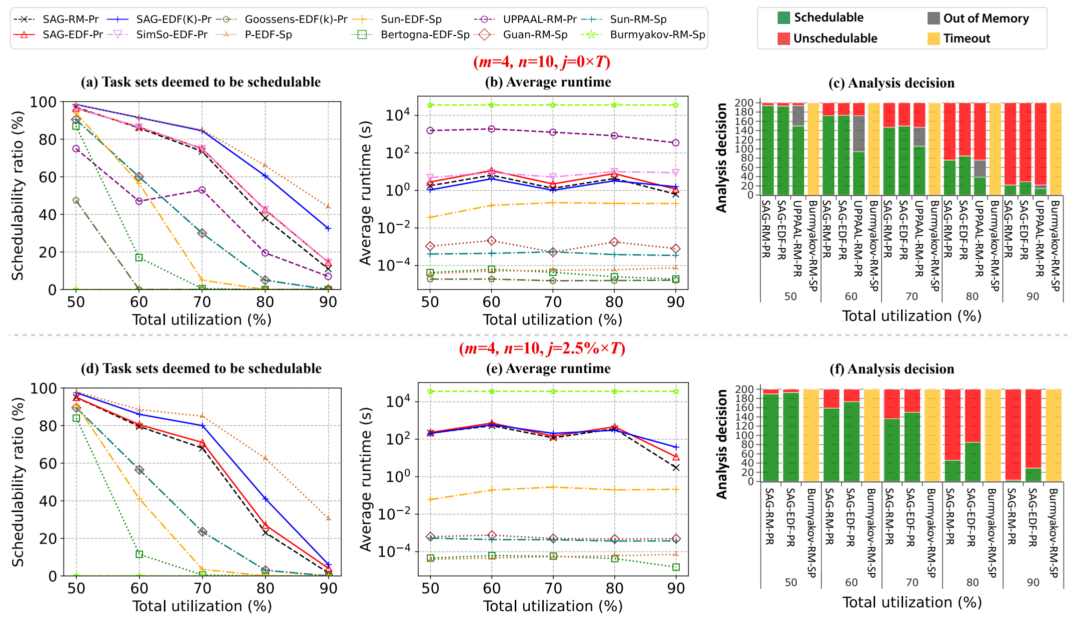


Figure 6 Experiments of Sec. 5.1: **(a-c)** Exp1 (no jitter), **(d-f)** Exp2 (with release jitter).

Fig. 6e shows the average runtime of each test. On average, our analysis for G-EDF and G-RM finished in 316.75 seconds and 245.74 seconds (about 5 minutes), respectively.

5.2 Comparison against sufficient tests

In this section, we excluded the exact tests to remove the limitation on task periods and the system size. We design three experiments to study the impact of system utilization (U), the number of cores (m), and the number of tasks (n) on the schedulability ratio and runtime of our analysis. In each experiment, we generated 200 periodic task sets for each data point.

Task set generation. We generated task sets as explained in Sec. 5.1 but we chose periods from $10,000$ to $100,000\mu s$ to show that our analysis's runtime is not affected by the time granularity. For each task, we assumed a release jitter of $200\mu s$ and an implicit deadline. We also discarded any task set with more than $100,000$ jobs per hyperperiod (note that industrial task sets usually have only a few thousand jobs [28]).

Exp3 (impact of utilization). We considered systems with $n = 6$ tasks and $m = 4$ cores and varied the utilization from 50% to 90%. As shown in Fig. 7a, our analysis for G-EDF identifies 1.8 times more schedulable task sets (overall 68%) than the best test for G-EDF (Sun-EDF-Sp, which overall, found only 37.4% schedulable task sets across all utilization values). Similarly, for G-RM, our analysis identifies 1.3 times more schedulable task sets (overall 68.4%) than the best test for G-RM (Guan-RM-Sp, Sun-RM-Sp, which overall, they found 52.3% schedulable task sets across all utilization values).

The average runtime of our analysis across all utilization values (Fig. 7b) for G-EDF, G-RM, and G-EDF(k) are 8.6, 7.2, and 3.3 seconds, respectively.

Global v.s. partitioned scheduling. Fig. 7a shows that overall P-EDF-Sp identifies 85.6% schedulable task sets across all utilizations while according to our analysis, G-EDF, G-RM, and G-EDF(k) identify 68%, 68.4%, and 88.9% schedulable task sets, respectively. Fig. 7j, k,

Figure 7 Experiments of Sec. 5.2: **(a–c)** Exp3, **(d–f)** Exp4, **(g–i)** Exp5, **(j–l)** dominance results for Exp3.

and l provide insight into the gap between global and partitioned scheduling. The top half of these figures illustrates the number of task sets that were deemed schedulable exclusively by global EDF, RM, or EDF(k), but not by any (or each) of the partitioning strategies. For example, at $U = 90\%$, G-RM and G-EDF(k) identify, respectively, 6 and 37 task sets (out of 200) that were not schedulable by any of the partitioning heuristics. The bottom half of Fig. 7j, k, and l shows task sets that were schedulable by a partitioning heuristic but not with G-EDF, G-RM, or G-EDF(k), respectively. We believe the performance of global scheduling can still improve with better (job-level) priority strategies.

Exp4 (impact of the number of tasks). In this experiment, we varied the number of tasks from 6 to 16 (for systems with 70% utilization and 4 cores). As shown in Fig. 7d, increasing the number of tasks generally significantly increases the pessimism of all state-of-the-art tests for global policies likely due to over-approximating the carry-in workload. For example, even the best test for G-EDF (Sun-EDF-Sp) cannot identify more than 7% schedulable task sets (across all data points in Fig. 7d), while our method identified 86.4% which is *12 times more schedulable task sets*. Thanks to the accurate tracking of interference scenarios, we can now confirm that increasing the number of tasks does not really worsen the performance of global policies (it even improves the schedulability of G-EDF from 82% to 90.5%). This is likely due to the fact that increasing the number of tasks, reduces per-task utilization which allows global policies to better balance the load among the cores. The average runtime of our analysis across all data points in this experiment, for G-EDF, G-RM, and G-EDF(k) is 2084.35, 1092.08, and 744.65 seconds, respectively, as depicted in Fig. 7e.

Exp5 (impact of the number of cores). We varied the number of cores (m) for systems with $U = 70\%$ and $n = 1.5 \times m$ tasks. Fig. 7g shows that the schedulability of global policies decreases when the number of cores increases, however, the existing sufficient tests suffer much more from the increase in the number of cores than our analysis. For instance, our analysis for G-EDF identifies *2.9 times more schedulable task sets* (overall 68.2%) than Sun-EDF-Sp (overall 22.9%). Similarly, our analysis for G-RM identifies *1.7 times more schedulable task sets* (overall 67.7%) than Guan-RM-Sp and Sun-RM-Sp (overall 39.7%). Despite the continued decrease in schedulability of G-EDF and G-RM when the number of cores increases, we noticed that G-EDF(k) (SAG-EDF(k)-Pr) and partitioned scheduling (P-EDF-Sp) maintain a high schedulability ratio. This is partly due to the inefficiency of EDF and RM priority assignment methods for global policies. Overall, on average SAG-EDF(k)-Pr and P-EDF-Sp detect 93.1% and 93.5% schedulable task sets, respectively. As shown in Fig. 7h, the average runtime of our analysis for G-EDF, G-RM, and G-EDF(k) is 6796.45, 2988.49, and 3095.24 seconds, respectively.

For a better understanding of the runtime of our analysis, we plotted the runtime of the analysis for each task set in Exp 3, Exp 4, and Exp 5 in Fig. 7c, f, and i, respectively. It can be seen, that the runtime increases as the number of jobs in a hyperperiod grows. Nevertheless, our analysis runtime typically ranges from a few seconds to a few minutes on average (e.g., for a system with 4 cores and 6 to 14 tasks). The only instances where the runtime exceeded an hour were when we scaled up the number of cores to 10 or the number of tasks to 16. Even then, overall, 95% of the generated task sets in all experiments completed analysis within 2 hours.

6 Conclusion and future work

In this paper, we introduced a novel response-time analysis for preemptive periodic tasks (or job sets) with release jitter under global JLFP scheduling. We extended the schedule-abstraction graph (SAG) concept, originally designed for non-preemptive tasks. We addressed the challenges of extending SAG to preemptive tasks by introducing concepts such as time partitions to accurately track interference from potential preemptions. Our analysis is considerably more accurate than the state of the art. For instance, we identify *12 times more schedulable task sets* than existing tests for G-EDF (and *2.7 times more schedulable task sets* for G-RM), e.g., for systems with 6 to 16 tasks, 70% utilization, and 4 cores. Similarly, we identify *2.9 times (1.6 times) more schedulable task sets* than existing tests for G-EDF (for

G-RM), for example, for systems with $m = 2$ to 10 cores, 70% utilization, and $1.5\times m$ tasks per core. We also provided the first response-time analysis for global EDF(k) – a global JLFP policy designed to address the Dhall's effect– showing that it can schedule 1.3 times more task sets (overall 93.1%) than G-EDF (overall 68.2%) for systems with $m = 2$ to 10 cores, 70% utilization, and $1.5\times m$ tasks per core.

With these results, we believe that our work makes a breakthrough in gaining insights into the true performance of global scheduling policies. In the future, we will extend our analysis to support precedence constraints and to use partial-order reduction to further reduce its runtime and memory consumption.

References

1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. An empirical survey-based study into industry practice in real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

2 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, 58(3):358–398, 2022.

3 Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 193–202, 2001.

4 Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal*, 8(5):284–292, 1993.

5 Theodore P Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Real-Time Systems Symposium (RTSS)*, pages 120–129, 2003.

6 Theodore P Baker. An analysis of EDF schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8):760–768, 2005.

7 Theodore P Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *International Conference On Principles Of Distributed Systems*, pages 62–75, 2007.

8 Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Real-Time Systems Symposium (RTSS)*, pages 119–128, 2007.

9 Sanjoy K Baruah, Aloysius K Mok, and Louis E Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

10 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 209–218, 2005.

11 Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2008.

12 Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-time systems*, 30(1-2):129–154, 2005.

13 Alessandro Biondi and Youcheng Sun. On the ineffectiveness of 1/m-based interference bounds in the analysis of global edf and fifo scheduling. *Real-Time Systems*, 54:515–536, 2018.

14 Vincenzo Bonifaci and Alberto Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. *Algorithmica*, 63:763–780, 2012.

15 Artem Burmyakov, Enrico Bini, and Chang-Gun Lee. Towards a tractable exact test for global multiprocessor fixed priority scheduling. *IEEE Transactions on Computers*, 71(11):2955–2967, 2022.

16 Artem Burmyakov, Enrico Bini, and Eduardo Tovar. An exact schedulability test for global FP using state space pruning. In *International Conference on Real Time and Networks Systems (RTNS)*, pages 225–234, 2015.

17 Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. SimSo: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 37–42, 2014.

18 Liliana Cucu-Grosjean and Joël Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of systems architecture*, 57(5):561–569, 2011.

19 Robert I Davis and Alan Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 398–409, 2009.

20 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pages 6–11, 2010.

21 Pourya Gohari, Mitra Nasri, and Jeroen Voeten. Data-age analysis for multi-rate task chains under timing uncertainty. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 24–35, 2022.

22 Pourya Gohari, Jeroen Voeten, and Mitra Nasri. Response-time analysis of fault-tolerant hard real-time systems under global scheduling. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 263–264, 2023.

23 Pourya Gohari, Jeroen Voeten, and Mitra Nasri. Work-in-progress: Tight response-time analysis for periodic preemptive tasks under global scheduling. In *Real-Time Systems Symposium (RTSS)*, pages 451–454, 2023.

24 Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25:187–205, 2003.

25 Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pages 263–272, 2007.

26 Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Real-Time Systems Symposium (RTSS)*, pages 387–397, 2009.

27 Rhan Ha and Jane WS Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *International conference on distributed computing systems*, pages 162–171, 1994.

28 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

29 Jinkyu Lee and Insik Shin. Limited carry-in technique for real-time multi-core scheduling. *Journal of Systems Architecture*, 59(7):372–375, 2013.

30 Mitra Nasri and Björn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling. In *Real-Time Systems Symposium (RTSS)*, pages 12–23, 2017.

31 Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. A response-time analysis for non-preemptive job sets under global scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 9–1, 2018.

32 Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 21–1, 2019.

33 Geoffrey Nelissen, Joan Marcè i Igual, and Mitra Nasri. Response-time analysis for non-preemptive periodic moldable gang tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2022.

34 Suhail Nogd, Geoffrey Nelissen, Mitra Nasri, and Björn B Brandenburg. Response-time analysis for non-preemptive global scheduling with FIFO spin locks. In *Real-Time Systems Symposium (RTSS)*, pages 115–127, 2020.

35 Sayra Ranjha, Pourya Gohari, Geoffrey Nelissen, and Mitra Nasri. Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks. *Real-Time Systems*, pages 1–55, 2023.

36 Maria A Serrano, Alessandra Melani, Sebastian Kehr, Marko Bertogna, and Eduardo Quiñones. An analysis of lazy and eager limited preemption approaches under dag-based global fixed priority scheduling. In *International Symposium on Real-Time Distributed Computing (ISORC)*, pages 193–202, 2017.

37 Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information processing letters*, 84(2):93–98, 2002.

38 Srinidhi Srinivasan, Geoffrey Nelissen, Reinder J. Bril, and Nirvana Meratnia. Analysis of tsn time-aware shapers using schedule abstraction graphs. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2024.

39 Youcheng Sun and Marco Di Natale. Assessing the pessimism of current multicore global fixed-priority schedulability analysis. In *ACM Symposium on Applied Computing*, pages 575–583, 2018.

40 Youcheng Sun and Giuseppe Lipari. A weak simulation relation for real-time schedulability analysis of global fixed priority scheduling using linear hybrid automata. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 35–44, 2014.

41 Youcheng Sun and Giuseppe Lipari. Response time analysis with limited carry-in for global earliest deadline first scheduling. In *IEEE Real-Time Systems Symposium*, pages 130–140, 2015.

42 Youcheng Sun and Giuseppe Lipari. A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor global fixed-priority scheduling. *Real-Time Systems*, 52:323–355, 2016.

43 Youcheng Sun, Giuseppe Lipari, Nan Guan, and Wang Yi. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–9, 2014.

44 Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233, 2019.

45 Quan Zhou, Guohui Li, and Jianjun Li. Improved carry-in workload estimation for global multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2527–2538, 2017.

46 Quan Zhou, Guohui Li, Chunyang Zhou, and Jianjun Li. Limited busy periods in response time analysis for tasks under global edf scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(2):232–245, 2020.