An Exact and Sustainable Analysis of Non-Preemptive Scheduling

Mitra Nasri Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—This paper provides an exact and sustainable schedulability test for a set of non-preemptive jobs scheduled with a fixed-job-priority (FJP) policy upon a uniprocessor. Both classic work-conserving and recent non-work-conserving schedulers are supported. Jobs may exhibit both release jitter and execution time variation. Both best- and worst-case response time bounds are derived. No prior response-time analysis (RTA) for this general setting is both exact and sustainable, nor does any prior RTA support non-work-conserving schedulers. The proposed analysis works by building a *schedule graph* that precisely abstracts all possible execution scenarios. Key to deferring the state-space explosion problem is a novel path-merging technique that collapses similar scenarios without giving up analysis precision. In an empirical evaluation with randomly generated workloads based on an automotive benchmark, the method is shown to scale to 30+ periodic tasks with thousands of jobs (per hyperperiod).**

I. Introduction

The key challenge in the analysis of non-preemptive uniprocessor scheduling is that it exhibits anomalies due to execution time variations and release jitter. That is, even though a periodic real-time workload may meet all deadlines when all jobs exhibit maximal execution times and arrive as late as possible (which intuitively should constitute the "worst case"), under non-preemptive scheduling, the workload may still suffer deadline misses if some jobs execute for a shorter time or arrive earlier, which intuitively should be an "easier" scenario, but often is not.

Such variations, however, are inevitable in real systems due to practical issues such as interrupt-handling delays, timer inaccuracies, data dependencies, multiple program paths, or system configuration changes. Further, in addition to these system factors, existing timing analysis frameworks exhibit imprecision in estimating both the *best-* and *worst-case execution times* (BCETs and WCETs, respectively) of programs, and particularly so if the underlying processor uses caches or out-of-order pipelines. And even in non-processor contexts such as the analysis of messages transmitted over a CAN bus, payload sizes and the times at which messages become available for transmission may vary unpredictably at runtime.

As a result, to be useful in practice, schedulability analysis must be designed to cope with uncertain resource needs and timing behavior. That is, a practical schedulability test must be *sustainable* [1], which means that if it deems a workload schedulable, then all deadlines must be met even if jobs underrun their WCETs or exhibit less jitter than anticipated.

Unfortunately, guaranteeing that analysis results for non-preemptive policies are sustainable is quite difficult due to the inherent unsustainability of the scheduler itself. In particular, it rules out any approaches based on simulating the worst case, simply because the true worst case is unknown in general.

The traditional approach for sustainable schedulability analysis is hence to pessimistically over-approximate the true worst case with an artificially designed scenario that may not actually be possible at runtime. For example, in the classic *response-time analysis* (RTA) for *non-preemptive fixed-priority* (NP-FP) scheduling [2, 3], a task's response time is bounded under the assumption that a job in the worst case incurs *both* maximal blocking due to lower-priority jobs *and* maximal interference due to higher-priority jobs. And while this is *exact* for *sporadic* tasks, for periodic workloads—the focus of this paper—such an approach is often too pessimistic since typically no individual job suffers both maximal blocking and maximal interference.

As a more recently explored alternative, approaches based on model checking (e.g., [4, 5]) or the exhaustive exploration of all system states [6, 7, 8] can guarantee exact and sustainable analysis results in principle. However, in addition to the fact that these proposals have so far been aimed at a very different scheduling problem, namely preemptive global multiprocessor scheduling, they suffer also from massive state-space explosion issues that render these techniques impractical even for modestly sized workloads (as discussed in more detail in Sec. V).

Common to both classic RTA and the more recent techniques based on formal methods is that neither approach supports *non-work-conserving* schedulers, which in the past few years have been shown to offer promising schedulability advantages [9, 10]. In particular, *Precautious Rate-Monotonic* (P-RM) [9] and *Critical-Window EDF* (CW-EDF) [10] scheduling, two recent proposals built on the idea of separating an *idle-time insertion policy* (IIP) from the usual job-ordering policy, offer empirically excellent performance while remaining conceptually simple. Unfortunately, the only available schedulability tests for these non-work-conserving algorithms are limited to specific, narrow special cases such as *harmonic tasks* with no release jitter and workloads that do not exhibit varying execution times. The lack of a general and sustainable schedulability analysis for non-work-conserving schedulers is a major gap in understanding, both in practical terms and from a foundational point of view.

To summarize, the current state of the art in the analysis of non-preemptive periodic real-time workloads on uniprocessors leaves considerable room for improvement.

This paper. To address these shortcomings, we provide the first *sustainable and exact* schedulability analysis for *both* work-conserving and non-work-conserving fixed-job-priority (FJP) scheduling algorithms—including NP-FP, non-preemptive EDF (NP-EDF), P-RM, and a variant of CW-EDF—while allowing for both execution time variation and release jitter.

Our schedulability analysis (Sec. III) is based on building a graph (Sec. III-B), called the *schedule graph*, that *precisely* abstracts all possible execution orders of a set of jobs (Sec. III-C), and from which *tight* response-time bounds can be easily inferred (Sec. III-D). The key technique in the construction of

the schedule graph is a novel *merge phase* that collapses similar scenarios without giving up analysis precision. This pruning of the graph allows our solution to scale to real-world-sized workloads, as we show in Sec. IV with an empirical evaluation considering workloads consisting of more than 30 periodic tasks with thousands of jobs (per hyperperiod).

II. System Model and Definitions

For ease of exposition, we first introduce our analysis for finite job sets (i.e., without a task concept), and discuss how to apply the technique to periodic tasks later in Sec. III-E.

A. Job and System Model

We consider the problem of scheduling a finite set of non-preemptive jobs $\mathcal{J}$ on a uniprocessor. Each job J_i has an earliest release time r_i^{min}, latest release time r_i^{max}, absolute deadline d_i, BCET C_i^{min}, WCET C_i^{max}, and priority p_i. A numerically smaller value of p_i implies higher priority. All job parameters are integer multiples of the system clock.

At runtime, each job is *released* at an *a priori* unknown time r_i, where $r_i \in [r_i^{min}, r_i^{max}]$, and requires $C_i \in [C_i^{min}, C_i^{max}]$ units of processor service. *Release jitter* (i.e., bounded release-time uncertainty) is caused by implementation factors such as interrupt latency or timer inaccuracy, and also if releases are triggered by external events that may be delayed (e.g., network packets). *Execution time variation* arises due to processor caches, input dependencies, program-state dependencies, program path diversity, etc. Note that release jitter does not affect the absolute deadline of a job, i.e., d_i is relative to the *expected* release time r_i^{min}. We do not consider job-discarding policies: released jobs remain pending until completed.

We say a job J_i is *possibly released* at time t if $t \geq r_i^{min}$, and *certainly released* if $t \geq r_i^{max}$. If a job J_i with execution cost C_i starts execution at time $t_x \geq r_i$, then it occupies the processor during the interval $[t_x, t_x + C_i)$ and we say that J_i *finishes by* time $t_x + C_i$, i.e., the next job can commence execution *at* time t if the previous job finishes *by* time t.

We assume that any ties in priority are broken by task and/or job ID. For ease of notation, we assume that the "$<$" order reflects this tie-breaking rule, i.e., tie-breaking is implicit.

We use $\{.\}$ to denote a set of items in which the order of elements is irrelevant and $\langle . \rangle$ to denote an enumerated set (or sequence) of items. In the latter case, we assume that items are indexed in the order of their appearance in the sequence. Finally, we let $\mathcal{P}(\mathcal{J})$ denote the power set of $\mathcal{J}$, and use $\mathbb{N}$ (including zero and ∞) to model discrete time.

B. Execution Scenarios, Schedulers, and Idle-Time Insertion

A set of jobs $\mathcal{J}$ is *schedulable* under a given scheduling policy if no execution scenario of $\mathcal{J}$ results in a deadline miss, where an execution scenario is defined as follows.

Definition 1. An *execution scenario* $\gamma = (C, R)$ for a set of jobs $\mathcal{J}$ 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, for each job J_i, $C_i \in [C_i^{min}, C_i^{max}]$ and $r_i \in [r_i^{min}, r_i^{max}]$.

Algorithm 1: IIP-Aware FJP Scheduler

Input : t: the current time, L: the IIP, $\mathcal{J}^S$: completed jobs

```
1 J_i ← the highest-priority pending job;
2 if t ≤ L(J_i, t, J^S) then
3 |  Schedule J_i;
4 else
5 |  Idle the processor until the next job is released;
6 end
```

In this paper, we focus exclusively on *deterministic* scheduling algorithms, i.e., scheduling algorithms that always produce the same schedule for a given execution scenario. The schedulability analysis presented in the paper can be applied both to classic work-conserving and a special class of non-work-conserving FJP scheduling algorithms that augment the job priority ordering with an *idle-time insertion policy* (IIP) [10].

An IIP is invoked whenever the system may commence the execution of a job, as shown in line 2 of Algorithm 1. It determines whether the highest-priority job should be scheduled (as is always the case in a work-conserving scheduler), or whether alternatively the processor should be idled in a non-work-conserving fashion. The basic IIP idea is that strategically inserted idle time can greatly increase non-preemptive schedulability by avoiding pathological blocking scenarios [10].

We generalize this notion to the case of *job-set-dependent-IIPs* (JIIPs). A JIIP determines the *latest permissible start time* $L(J_i, t, \mathcal{J}^S)$ of the highest-priority pending job J_i at time t, given that the set of jobs denoted by $\mathcal{J}^S$ have already finished (see line 2 in Algorithm 1). In other words, if $t \leq L(J_i, t, \mathcal{J}^S)$, then job J_i will be scheduled, otherwise the processor will be idled until another job is released (line 5 in Algorithm 1).

We require JIIPs to be *stable* in their decisions, i.e., a JIIP is not allowed to "flip-flop" on its decision whether to block J_i. More precisely, if at some point in time t the JIIP blocks a job J_i from being scheduled (for a given $\mathcal{J}^S$), then it will also block J_i at any later point in time (for the same $\mathcal{J}^S$): if $t > L(J_i, t, \mathcal{J}^S)$, then also $t' > L(J_i, t', \mathcal{J}^S)$ for any $t' > t$. That is, once a job J_i is blocked by the JIIP, the set of scheduled jobs $\mathcal{J}^S$ must change before J_i may be considered again.

Our JIIP definition covers many forms of previously studied IIPs. For example, under P-RM [9], a low-priority job may be scheduled only if it cannot cause the next maximum-priority job to miss a deadline, which can be formalized as follows. Let $\hat{p}$ denote the maximum priority level. At a given time t, let J_x ($x \neq i$) be the job (if any) that satisfies the condition

$$r_x^{max} = \min\left\{ r_z^{max} \;\middle|\; J_z \in \mathcal{J} \setminus \mathcal{J}^S \wedge p_z = \hat{p} \wedge t < r_z^{max} \right\}.$$

Based on the intuition that J_x could miss its deadline if J_i starts execution after time $d_x - C_x^{max} - C_i^{max}$, we define

$$L(J_i, t, \mathcal{J}^S) = \begin{cases} d_x - C_x^{max} - C_i^{max} & \text{if } J_x \text{ exists,} \\ \infty & \text{otherwise.} \end{cases} \quad (1)$$

If there is no future higher-priority job that must be "protected," or if J_i has maximum priority itself ($p_i = \hat{p}$), then

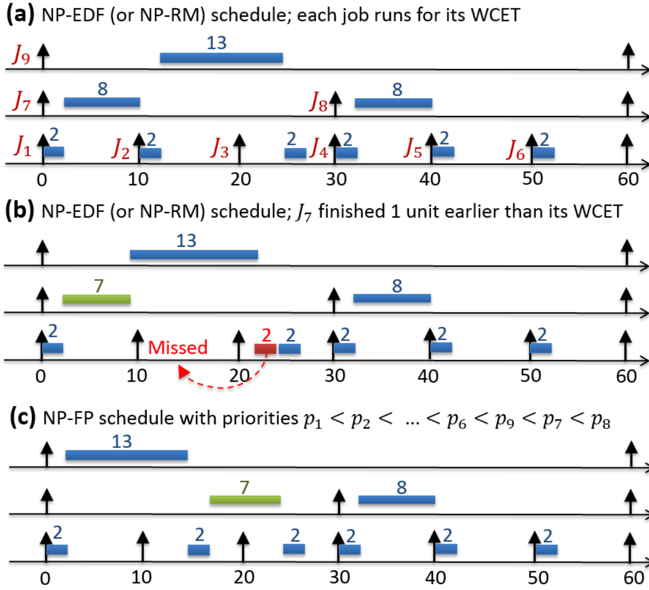

Fig. 1. An example for 3 different scheduling scenarios of a job set $J = \{J_1, J_2, \ldots, J_9\}$ with no release jitter. The execution time range for J_1 to J_6 is $[1, 2]$, for J_7 and J_8 it is $[7, 8]$, and for J_9 it is $[3, 13]$. The relative deadline of J_1 to J_6 is 10, for J_7 and J_8 it is 30, and for J_9 it is 60.

$L(J_i, t, \mathcal{J}^S) = \infty$, which means that the JIIP never blocks J_i. The CW-EDF JIIP [10] is conceptually similar, but notationally more involved, in part since CW-EDF assumes tasks without release jitter; these details can be found in the appendix.

The proposed schedulability analysis supports the broad class of work-conserving and non-work-conserving non-preemptive scheduling algorithms that satisfy the following three properties: **(i)** the non-preemptive scheduling policy must be based on fixed job priorities (FJP), i.e., a job commences execution only if it has the highest priority among all pending jobs; **(ii)** it must use a stable JIIP, or no IIP at all; and **(iii)** it must be an *eager* scheduling algorithm, i.e., it must not leave the processor idle as long as the highest-priority pending job in the system is allowed to be scheduled by the JIIP. Hereafter, we use the terms IIP and JIIP interchangeably and assume the presence of an IIP. Work-conserving algorithms that do not require an IIP can be modeled with the trivial JIIP $\forall J_i, t, \mathcal{J}^S : L(J_i, t, \mathcal{J}^S) \triangleq \infty$.

III. Exact Schedulability Analysis

The core of our schedulability analysis is a search in a graph that abstracts the schedules of all possible execution scenarios. We first illustrate the main idea of the approach with an example, then precisely define the algorithm, establish its correctness, show how it yields exact response-time bounds, and finally discuss how it can be applied to periodic tasks.

A. Motivation and Basic Idea

Consider a job set subject to execution time variation as shown in Fig. 1.[1] For simplicity, there is no release jitter in this example. In the execution scenario in which all execution costs are maximal, i.e., $\forall i, C_i = C_i^{max}$, all deadlines are met in the schedule produced by NP-EDF as illustrated in Fig. 1-(a).

[1]This job set is based on an example in [10].

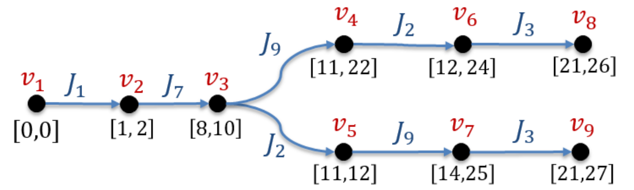

Fig. 2. A schedule graph for the job set in Fig. 1 scheduled by NP-EDF.

However, if $C_7 = 7 < C_7^{max} = 8$, a deadline is missed as shown in Fig. 1-(b), which illustrates the problem of scheduling anomalies under non-preemptive polices. The job set is hence not schedulable under NP-EDF as there exists a scenario in which not all deadlines are met.

In this example, the problem can be avoided by using NP-FP and the priority assignment $p_1 < p_2 < \ldots < p_6 < p_9 < p_7 < p_8$, as shown in Fig. 1-(c). In fact, with this priority assignment, no deadline is missed in *any* execution scenario.

In general, it is not easy to distinguish between the cases illustrated in Figs. 1-(b) and 1-(c), i.e., to decide whether there exists an execution scenario in which a deadline is missed (for a given set of jobs and a given priority order). The central contribution of this paper is a method that solves this problem with practical time and memory requirements.

Our schedulability analysis is based on searching all possible job orderings (or sequences) produced by a given scheduling algorithm $\mathcal{A}$ for all possible execution scenarios. For example, if NP-EDF is used to schedule the job set in Fig. 1, although there are many possible execution scenarios, there are only exactly two possible job sequences that can arise: $\langle J_1, J_7, J_2, J_9, J_3, J_4, J_8, J_5, J_6 \rangle$ (Fig. 1-(a)) and $\langle J_1, J_7, J_9, J_2, J_3, J_4, J_8, J_5, J_6 \rangle$ (Fig. 1-(b)). Once all such sequences are known, the schedulability analysis problem reduces to simply calculating the *earliest-* and *latest-possible finish time* (EFT and LFT) of each job in each possible sequence. The main challenge is hence the need for an exact yet efficient way for enumerating all possible job sequences.

For example, consider the job set in Fig. 1 under NP-EDF scheduling. In particular, consider the prefix sequence $\langle J_1, J_7 \rangle$: due to the execution time uncertainty of J_1 and J_7, the EFT and LFT of J_7 are 8 and 10, respectively. Since a high-priority job is released within this interval (J_2 is released at time 10), two cases may arise depending on the total sum of execution costs $C_1 + C_7$: if $C_1 + C_7 = 10$, then J_2 is scheduled next (Fig. 1-(a)), but if $C_1 + C_7 \leq 9$, then J_9 is scheduled next (and J_2 misses its deadline, Fig. 1-(b)).

This information can be expressed with a directed acyclic graph (DAG) as shown in Fig. 2, where each edge is labeled with the job that is scheduled next, and each vertex is labeled with an interval of time spanning the EFT and LFT of the last-scheduled job. Equivalently, the label of a vertex v_i can be seen as the EFT and LFT of the sequence of jobs that are edge labels on the path from v_1 to v_i. Importantly, this graph has a branch (i.e., a path originating at the root v_1) for every possible job sequence, thereby abstracting all possible execution scenarios. For instance, in Fig. 2, the EFT of v_4 is

reached when $C_1 = 1$, $C_7 = 7$, and $C_9 = 3$, while the EFT of v_5 is reached when $C_1 + C_7 = 10$ and $C_2 = 1$.

More formally, a *schedule graph* is a DAG $G = (V, E)$, where the label of each edge $e_k \in E$ is a job $J_x \in \mathcal{J}$ and the label of each vertex $v_i \in V$ is an interval, denoted by $[e_i, l_i]$, that represents the EFT and LFT of the sequence of jobs (i.e., edge labels) in any path that connects the root vertex v_1 to v_i. That is, with the exception of v_1, for any v_i and any time $t \in [e_i, l_i]$, we require that there exists an execution scenario such that the last job in the sequence of jobs on the corresponding path from v_1 to v_i finishes by time t.

We next explain how to build a schedule graph G for a given set of jobs $\mathcal{J}$ such that G abstracts all schedules that can arise due to any possible execution scenario of $\mathcal{J}$.

B. Generating a Schedule Graph

At a high level, the proposed algorithm consists of two alternating phases, the *expansion phase* and the *merging phase*. During the expansion phase, (one of) the shortest path(s) in the graph from the root to a leaf is grown by considering all jobs that can appear next in the sequence represented by the path. For each such job, a new vertex is created, added to the graph, and connected via an edge from the last-expanded vertex.

After a path has been expanded, the merging phase tries to compact the graph. To this end, the ending vertices of paths that have the same set of labels (i.e., jobs) and intersecting end-vertex labels (i.e., finish-time intervals) are merged. This step is essential to avoid redundant work, i.e., to recognize two or more similar states early on before they are all expanded.

The algorithm terminates when there are no more incomplete paths to expand, or if a dead-end path is discovered (i.e., an incomplete path that cannot be further expanded).

Algorithm 2 shows the proposed iterative *schedule graph generation algorithm* (SGA) in full detail. The graph is initialized in line 1 with a root vertex v_1 labeled with the interval $[0, 0]$. The repeating expansion phase corresponds to lines 2-10; lines 12–14 constitutes the merging phase. The expansion and merging phases repeat until every path in the graph contains $|\mathcal{J}|$ distinct jobs, or until a path fails to be expanded (lines 4–6). In the remainder of this section, we discuss the expansion and merge phases phases in detail.

1) Expansion phase: Fig. 3 shows an illustration of the expansion phase. In general, given a path P from v_1 to a leaf v_i (line 3), let $\mathcal{J}^P$ and $|\mathcal{J}^P|$ denote the job sequence (i.e., the sequence of labels of traversed edges) and the length of P, respectively, and let e_i and l_i denote the EFT and LFT of v_i, respectively (i.e., v_i is labeled with the interval $[e_i, l_i]$).

In order to expand P, Algorithm 2 must consider all jobs that can be scheduled after e_i in *some* execution scenario. Conversely, the goal is to filter jobs that will certainly *not* be scheduled next in *any* execution scenario. To this end, we introduce three notions of "eligibility" to be scheduled next.

Definition 2. A possibly released job J_j is *priority-eligible* at time t iff

$$\nexists J_x \in \mathcal{J} \setminus \mathcal{J}^P \text{ s.th. } r_x^{max} \leq t \wedge p_x < p_j. \tag{2}$$

Algorithm 2: Schedule Graph Construction (SGA)

Input : Job set $\mathcal{J}$, scheduling algorithm $\mathcal{A}$
Output : Schedule graph $G = (V, E)$

1 Initialize G by adding a root vertex v_1 with interval $[0, 0]$;
2 **while** ($\exists$ *path* P *from* v_1 *to a leaf s.th.* $|P| < |\mathcal{J}|$) **do**
3 $\quad P \leftarrow$ the shortest path from v_1 that ends in a leaf;
4 $\quad$ **if** *(there is no eligible job (Definition 5)* **then**
5 $\quad\quad$ **return** unschedulable;
6 $\quad$ **end**
7 $\quad$ **for each** *eligible successor job* J_j *(Definition 5)* **do**
8 $\quad\quad$ Add a new vertex v_k to V with label $[e_k, l_k]$ based on Equations (3) and (5);
9 $\quad\quad$ Add an edge from v_i to v_k with label J_j;
10 $\quad\quad$ Let path $P' = P + \langle v_k \rangle$;
11 $\quad\quad$ **while** ($\exists$ *path* Q *that matches* P' *(Definition 6)* **do**
12 $\quad\quad\quad$ Update $[e_k, l_k] \leftarrow [e_q, l_q] \cup [e_k, l_k]$;
13 $\quad\quad\quad$ Redirect all incoming edges of v_q to v_k;
14 $\quad\quad\quad$ Remove v_q from V;
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 **end**

In other words, J_j is priority-eligible iff $r_j^{min} \leq t$ and there is no certainly released, still incomplete higher-priority job.

Example. Consider Fig. 3, where J_5 is the last-scheduled job on path P to v_i. J_5 can finish by any time from e_i until l_i. In this example, J_7, J_6, J_3, and J_2 are priority-eligible because for each job there exists an execution scenario in which it becomes the highest-priority pending job at some time during $[e_i, l_i]$. If J_5 finishes by time e_i and J_6 is released prior to e_i, then J_6 becomes the highest-priority job at e_i. However, if J_5 finishes by time e_i and J_6 is released at time r_6^{max}, then J_7 has the highest priority at time e_i. Similarly, if J_5 finishes by time r_3^{min} (respectively, time r_2^{min}), then J_3 (respectively, J_2) can become the highest-priority pending job. Crucially, J_4 can never succeed J_5 in any execution scenario as it is always released after the higher-priority job J_3; J_4 is thus not priority-eligible and can be safely ignored when expanding P.

Next, we observe that it is similarly safe to ignore jobs that are certainly blocked by the IIP.

Definition 3. A job J_j is *IIP-eligible* at time t iff the IIP permits it to be scheduled, i.e., iff $t \leq L(J_j, t, \mathcal{J}^P)$.

Finally, we must take into account a third necessary condition: to be scheduled next, a job J_j must either **(i)** be already released by the time the predecessor finishes (i.e., at the latest by time l_i, which implies $r_j^{min} \leq l_i$), or **(ii)** there must exist an execution scenario such that no other job is scheduled in the time between the completion of the predecessor and the release of J_j. That is, if $r_j^{min} > l_i$, then for J_j to be scheduled next, the system must be certainly idle in the time interval $[l_i, r_j^{min})$. We refer to this criterium as "potentially-next eligibility."

Definition 4. A job J_j is *potentially next* at time t iff either: **(i)** J_j can be released before l_i, i.e., $r_j^{min} \leq l_i$, or

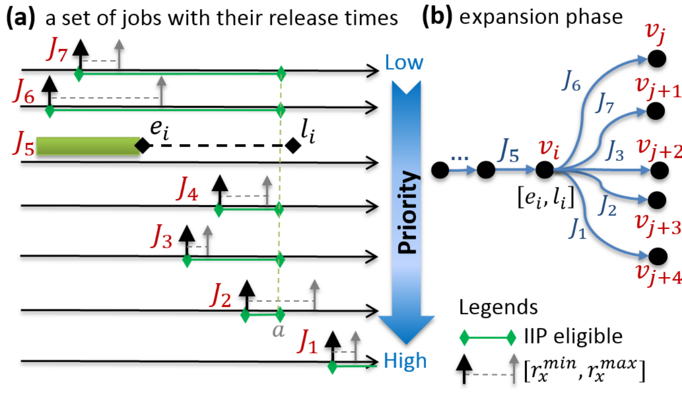

Fig. 3. An example showing how a given path P ending in v_i can be expanded. Note that jobs on lower axes have higher priority. The label of v_i, the interval $[e_i, l_i]$, is marked in the figure by a dashed line between two diamond-shaped points. For the sake of illustration, assume that $a = l_i - 1$.

(ii) no other job is certainly scheduled before J_j, i.e., there does not exist a job $J_x \in \mathcal{J} \setminus \mathcal{J}^P$ ($i \neq x$, $r_x^{max} < t$) that is both priority-eligible and IIP-eligible at time $\max\{r_x^{max}, l_i\}$.

Example. Returning to the example in Fig. 3, jobs J_2–J_7 are trivially potentially-next jobs as they satisfy clause (i) of Definition 4. Next, consider job J_1, which does not satisfy clause (i). However, since none of the jobs J_2, J_3, J_4, J_6, and J_7 are IIP-eligible at the time l_i, job J_1 satisfies the second clause, and hence, is a potentially next job for v_i.

Based on Definitions 2–4, we can precisely characterize the set of potential successor jobs in path P.

Definition 5. A job $J_j \in \mathcal{J} \setminus \mathcal{J}^P$ is an *eligible successor* for path P ending in vertex v_i iff J_j is IIP-eligible, priority-eligible, and potentially next at time $t_E = \max\{e_i, r_j^{min}\}$.

In Fig. 3-(a), J_7, J_6, J_3, J_2, and J_1 are eligible successors of J_5. Job J_4 is not eligible because it is not priority-eligible at time $\max\{e_i, r_4^{min}\} = r_4^{min}$ due to the presence of J_3.

We later show in Lemma 3 that Definition 5 is a necessary and sufficient condition for a job to be scheduled next in some possible execution scenario. Next, we derive the earliest- and latest-possible finish times of a successor job.

2) EFT and LFT of a new vertex: Consider an eligible-successor job J_j that is added at the end of path P after vertex v_i as the label of an edge to a new vertex v_k (lines 7–10 in Algorithm 2). The EFT of J_j (or, equivalently, of v_k) is

$$e_k = t_E + C_j^{min}, \tag{3}$$

where $t_E = \max\{e_i, r_j^{min}\}$ is the earliest start time of J_j.

The LFT is determined by three factors that bound the latest time at which J_j must start execution to be next in line. First, under eager policies, J_j must start execution by time

$$t_L = \max\{l_i, r_j^{max}\},$$

because by that time it will certainly be released ($t_L \geq r_j^{max}$) and the previous job has certainly finished ($t_L \geq l_i$).

Second, since we consider FJP policies, J_j must start execution before a higher-priority job is certainly released, since the release of a higher-priority job implies that J_j cannot commence execution. Formally, given the set of incomplete higher-priority jobs $\mathcal{J}^H = \{J_x \mid J_x \in \mathcal{J} \setminus \mathcal{J}^P i \wedge p_x < p_i\}$, J_j's latest start time is upper-bounded by

$$t_H = \begin{cases} \infty, & \text{if } \mathcal{J}^H = \emptyset \\ \min\{r_x^{max} \mid J_x \in \mathcal{J}^H\} - 1 & \text{otherwise.} \end{cases} \tag{4}$$

Third, the latest time after time t_E at which the IIP permits J_j to start, which is given by $t_I = L(J_j, t_E, \mathcal{J}^P)$, is also an upper bound on the latest start time if J_j is scheduled next.

Based on these considerations, the LFT J_j (or v_k) is

$$l_k = \min\{t_L,\ t_H,\ t_I\} + C_j^{max}. \tag{5}$$

We next discuss the merge phase, which seeks to collapse redundant branches of the schedule graph.

3) Merge phase: To slow the growth of the graph, we merge "matching" paths that represent the same jobs (lines 11–15).

Definition 6. A path Q from v_1 to v_q is *matching* a given path P from v_1 to v_p iff $\mathcal{J}^P = \mathcal{J}^Q\ \wedge\ [e_q, l_q] \cap [e_p, l_p] \neq \emptyset$, where $\mathcal{J}^P$ and $\mathcal{J}^Q$ are the sets of jobs of paths P and Q (i.e., the labels of the traversed edges), respectively.

By *merging* a path Q with a path P we mean replacing the last vertex of Q with the last vertex of P. Assume that v_q and v_p are the last vertices of Q and P, respectively. First, we update the interval $[e_p, l_p]$ (i.e., the label of v_p) as follows:

$$[e_p, l_p] \leftarrow [e_p, l_p] \cup [e_q, l_q]. \tag{6}$$

Second, we modify the incoming edges to v_q to point them to v_p instead. At this point, v_q is no longer connected to or reachable from the rest of the graph and we simply remove v_q. As a result, after the merge, both paths P and Q end in v_p.

Example. In Fig. 2, the two paths $P = \langle v_1, v_2, v_3, v_4, v_6\rangle$ and $Q = \langle v_1, v_2, v_3, v_5, v_7\rangle$ are matching paths because they represent the same set of scheduled jobs (albeit in different orders) and because the labels of v_6 and v_7 intersect: $[12, 24] \cap [14, 25] \neq \emptyset$. The two paths will thus be merged by Algorithm 2, which leads to the final graph shown in Fig. 4.

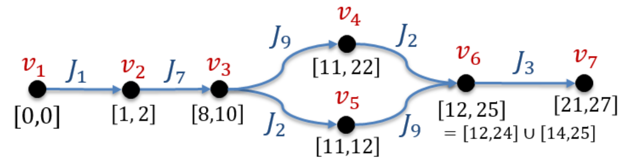

Fig. 4. The schedule graph produced by Algorithm 2 for the job set in Fig. 1 under NP-EDF scheduling. In v_6, the two paths representing the scenarios in Figs. 1-(a) and 1-(b) merge; the vertex label is hence a union of two intervals.

It is worth noting that, by design, Algorithm 2 does not merge vertices that have outgoing edges. This property is ensured because Algorithm 2 always expands (one of) the shortest path(s), similar to a breadth-first traversal. As a result, all paths grow in a balanced way so that, if a path Q matches the newly formed path P' in line 11 of Algorithm 2, then Q's last vertex has not yet been expanded. That is, at any point in time, any two paths from v_1 to any leafs differ in length by at most one.

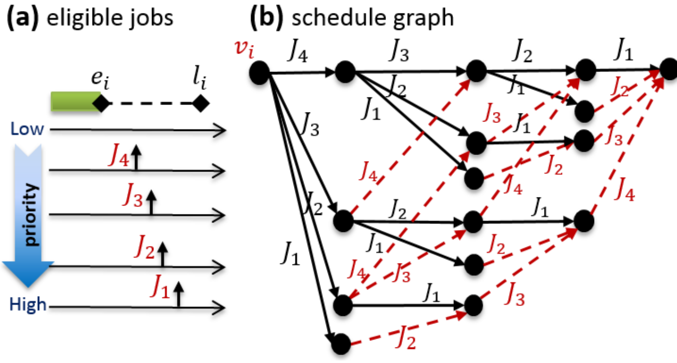

Fig. 5. The schedule graph generated by Algorithm 2 for a pathological case where jobs are released in reverse order of their priorities, the WCET of each job exceeds $l_i - e_i$, and the BCET of each job is zero, which maximizes the number of possible job sequences. For simplicity, there is no release jitter.

Fig. 5 shows a graph created by Algorithm 2 for a set of jobs that are released in the reverse order of their priority, which represents the worst-case scenario with regard to the number of leaf vertices that must be explored in the expansion phase.

However, as indicated with dashed edges, although there are many eligible successors initially, the width of the graph reduces quickly after several merge steps. While an analysis of the computational complexity of Algorithm 2 is beyond the scope of this paper (see also Sec. III-F), it is interesting that, given $n = 4$ eligible jobs in Fig. 5, the number of vertices reachable from v_i via a path of length x is given by the binomial coefficient $\binom{n}{x}$, i.e., there are $\binom{4}{1} = 4$ vertices reachable from v_i in 1 hop, $\binom{4}{2} = 6$ vertices reachable in 2 hops, $\binom{4}{3} = 4$ vertices reachable in 3 hops, and only $\binom{4}{4} = 1$ vertex reachable in 4 hops. In contrast, a naïve brute-force expansion without a merge phase would generate a tree with $n!$ vertices.

C. Proof of Correctness

In the following we establish that the graph constructed by Algorithm 2 reflects all job sequences that can arise from any possible execution scenario (Theorem 1). The proof has two main steps: we first argue that the EFT and LFT obtained from Equations (3) and (5) are tight (Lemmas 1 and 2), and then show that Definition 5 is a necessary and sufficient condition for a job to be scheduled after a given path (or job sequence) in at least one execution scenario (Lemma 3).

Lemma 1. *The EFT and LFT of a newly created vertex v_p cannot be smaller than* (3) *or larger than* (5)*, respectively.*

Proof. By induction. The base case is for v_1, in which the EFT and LFT are both trivially 0. In the induction step, assume that each label on every vertex from v_1 to v_i is an exact bound on the EFT and LFT of any path P from v_1 to v_i. We show that for a new vertex v_p that is added after v_i and connected by an edge labeled with a job J_j, (3) and (5) provide tight bounds on the EFT and LFT of all paths from v_1 to v_p.

EFT: The earliest start time of J_j, i.e., t_E, cannot be smaller that e_i since, by the induction hypothesis, e_i is the earliest time at which all prior jobs in path P can be finished. A lower bound on t_E is given by r_j^{min}, because J_j cannot execute before it is released. If J_j starts its execution at t_E, it cannot finish before (3) since its minimum execution time is C_j^{min}.

LFT: Next, we show that $l_p - C_j^{max}$ cannot exceed t_L, t_I, or t_H. First, consider t_H and suppose $t_H \neq \infty$ (otherwise the claim is trivial). Since a higher-priority job is certainly released at the latest at time $t_H + 1$, J_j is no longer the highest-priority pending job after t_H, and hence cannot commence execution under a FJP scheduler after time t_H. Hence, J_j will be a direct successor of path P only if its execution starts no later than time t_H. For the same reason, J_j must start its execution no later than time t_I, because if it does not commence execution before or at time t_I, then a stable IIP will continuously block J_j until another job has been scheduled.

If $l_i < r_j^{max} \leq \min\{t_I, t_H\}$, and if J_j is the job that is scheduled next, then J_j is the highest-priority job at time r_j^{max} that is allowed to be scheduled by the IIP. Since the scheduling algorithm under consideration is an *eager* scheduler, J_j is scheduled at r_j^{max}. Otherwise, if $r_j^{max} \leq l_i \leq \min\{t_I, t_H\}$, then J_j will be the highest-priority job at l_i and must be scheduled no later than l_i since all prior jobs in the path have finished by time l_i. Finally, since J_j can exhibit its maximum execution time C_j^{max}, the latest finish time of J_j is $l_p = C_j^{max} + \min\{t_I, t_H, \max\{l_i, r_j^{max}\}\}$. □

Lemma 2. *For every vertex $v_p \in V$ and any time $t \in [e_p, l_p]$, there exists an execution scenario and a corresponding path $P = \langle v_1, \ldots, v_p \rangle$ such that the last job in the job sequence represented by path P finishes by time t.*

Proof. Vertex labels are created in line 8 and modified in line 12 of Algorithm 2, where the union of two *intersecting* intervals replaces the previous label. Since such merges trivially maintain the claimed property, it is sufficient to show that each interval label created from (3) and (5) satisfies the claim.

The claim can be established inductively, where the base case (v_1) is again trivial. For the induction step, let v_i be the vertex via which v_p is initially connected when v_p's vertex label is first created. Let J_j be the label of the edge from v_i to v_p, and suppose the claim holds for v_i, i.e., for any time $t' \in [e_i, l_i]$, there exists an execution scenario such that the last job from v_1 to v_i finishes exactly by t'.

From (3), we have $e_p = t_E + C_j^{min}$, and from (5), we have $l_p \leq t_L + C_j^{max}$, which translates into four (not necessarily disjoint) cases: **(i)** $e_i \leq r_j^{min} \wedge l_i \leq r_j^{max}$, **(ii)** $r_j^{min} \leq e_i \wedge r_j^{max} \leq l_i$, **(ii)** $r_j^{min} \leq e_i \leq l_i \leq r_j^{max}$, and **(iv)** $e_i \leq r_j^{min} \leq r_j^{max} \leq l_i$. For each case, we identify an execution scenario such that J_j finishes by any chosen time $t \in [e_p, l_p]$.

In case (i), $e_p = r_j^{min} + C_j^{min}$ and $l_p \leq r_j^{max} + C_j^{max}$. Let J_j arrive at time $r_j = \max\{t - C_j^{max},\ r_j^{min}\}$, which is possible since $t \leq r_j^{max} + C_j^{max}$, let J_j execute for $C_j = \min\{C_j^{max},\ t - r_j^{min}\}$ time units, which is possible since $t \geq r_j^{min} + C_j^{min}$, and choose an execution scenario such that the predecessor of J_j (if any) finishes by time $t' = e_i \leq r_j^{min}$, which exists by the induction hypothesis. Then J_j is scheduled immediately at time r_j and finishes by time t.

In case (ii), $e_p = e_i + C_j^{min}$ and $l_p \leq l_i + C_j^{max}$. Let J_j arrive at time $r_j = \min\{e_i,\ r_j^{max}\}$, which is possible since

$r_j^{min} \leq e_i$, let J_j execute for $C_j = \min\{C_j^{max},\ t - e_i\}$ time units, which is possible since $t \geq e_i + C_j^{min}$, and choose an execution scenario such that the predecessor of J_j (if any) finishes by time $t' = \max\{e_i,\ t - C_j^{max}\}$, which exists by the induction hypothesis and since $t \leq l_i + C_j^{max}$. Then J_j is scheduled at time t' and finishes by time t.

In case (iii), $e_p = e_i + C_j^{min}$ and $l_p \leq r_j^{max} + C_j^{max}$. Let J_j arrive at time $r_j = \max\{e_i,\ t - C_j^{max}\}$, which is possible since $r_j^{min} \leq e_i$ and $t \leq r_j^{max} + C_j^{max}$, let J_j execute for $C_j = \min\{C_j^{max},\ t - e_i\}$ time units, which is possible since $t \geq e_i + C_j^{min}$, and choose an execution scenario such that the predecessor of J_j (if any) finishes by time $t' = \min\{l_i,\ r_j\}$, which exists by the induction hypothesis and since $r_j \geq e_i$. Then J_j is scheduled at time r_j and finishes by time t.

Finally, in case (iv), $e_p = r_j^{min} + C_j^{min}$ and $l_p \leq l_i + C_j^{max}$. Let J_j arrive at time $r_j = \min\{r_j^{max},\ \max\{r_j^{min},\ t - C_j^{max}\}\}$, let J_j execute for $C_j = \min\{C_j^{max},\ t - r_j\}$ time units, which is possible since $t \geq r_j^{min} + C_j^{min}$, and choose an execution scenario such that the predecessor of J_j (if any) finishes by time $t' = \max\{t - C_j^{max},\ r_j\}$, which exists by the induction hypothesis and since $e_i \leq r_j \leq l_i$ and $t \leq e_l + C_j^{max}$. Then J_j is scheduled at time t' and finishes by time t. □

Note that Lemma 2 does *not* claim that *every* path to a given vertex represents a set of execution scenarios that covers the full range of the vertex label. This is because the merge phase may widen vertex labels. For example, in Fig. 4, the label of v_6 is $[12, 25]$, but in any execution scenario that ends in J_2 being scheduled last (i.e., corresponding to the path $\langle v_1, \ldots, v_4, v_6 \rangle$), the last job in the sequence finishes by time 24 at the latest (as is also apparent in Fig. 2). However, the precise maximum finish time of J_2 can still be inferred from the graph in Fig. 4 by evaluating Equation (5) in the context of v_4 and the path prefix $\langle v_1, \ldots, v_4 \rangle$. Next, we consider the set of eligible jobs.

Lemma 3. *Job J_j is scheduled in some execution scenario as the next job in the job sequence represented by a path to a vertex v_i iff it is an eligible successor (Definition 5).*

Proof. Let P denote a path from v_1 to v_i.

If: Consider the following execution scenario: each job $J_x \in \mathcal{J} \setminus \mathcal{J}^P$ other than J_j is released at r_x^{max} and J_j is released at r_j^{min}. Since J_j is an eligible-successor, it is priority-eligible, and hence must be the highest-priority pending job among all other jobs that are certainly released before t_E. It is further IIP-eligible, and hence must be allowed to be scheduled by the IIP at time t_E. If $t_E \leq l_i$, then J_j can be scheduled at time t_E in an execution scenario in which the last job of path P finishes by time t_E. The existence of such an execution scenario follows from Lemma 2.

Otherwise, if $t_E > l_i$, then the following execution scenario allows J_j to be scheduled after P: the last job of path P finishes its execution at time l_i, J_j is released at time r_j^{min}, no other job is scheduled between the last job of P and J_j (i.e., the processor idles from l_i until , r_j^{min}), and hence J_j is scheduled at time r_j^{min}. The fact that the processor remains idle between the last job of P and J_j follows from clause (ii) of Definition 4 (which applies if $t_E > l_i$): J_j is an eligible successor of P only if any other job J_x that is certainly released before time $t_E = r_j^{min}$ (i.e., $r_x^{max} < t_E$) is either IIP-ineligible or priority-ineligible at time $\max\{r_x^{max}, l_i\}$ (where $\max\{r_x^{max}, l_i\} < t_E = r_j^{min}$ since $l_i < t_E$ and $r_x^{max} < t_E$). Furthermore, any such job J_x remains IIP-ineligible or priority-ineligible (or both) throughout the interval $[\max\{r_x^{max}, l_i\}, r_j^{min}]$ since a job is priority-ineligible while the processor is idle only if there exists a pending higher-priority, IIP-ineligible job, and since the IIP is stable (i.e., any job J_x that is IIP-ineligible at time $\max\{r_x^{max}, l_i\}$ remains indefinitely IIP-ineligible while the processor idles). Thus, J_j can be scheduled at r_j^{min} while no other job is scheduled between the last job of P and J_j.

Only if: We show that if J_j is not eligible by Definition 5, then there is no execution scenario in which J_j is scheduled after P. If J_j is not IIP-eligible, then the IIP will indefinitely block J_j from being scheduled until another job is scheduled since the IIP is stable. If J_j is not priority-eligible at time t_E, it cannot be scheduled after P since there will certainly exist a higher-priority pending job at time t_E, which prevents J_j from being scheduled under a FJP scheduler. Finally, if J_j is not next-eligible, then there exists at least one job J_x that is certainly released before r_j^{min} and that is IIP- and priority-eligible at time $\max\{r_x^{max}, l_i\} < r_j^{min}$. Under an eager FJP scheduler, this job will precede J_j regardless of its priority. □

Based on Lemmas 1–3, we conclude that Algorithm 2 constructs a precise abstraction: the final graph reflects all possible (Lemmas 1 and 3) and no impossible scenarios (Lemmas 2 and 3), which we summarize as Theorem 1.

Theorem 1. *If Algorithm 2 terminates successfully, then there exists an execution scenario such that a job $J_j \in \mathcal{J}$ completes at some time t (under the given scheduler) iff there exists a path $P = \langle v_1, \ldots, v_i, v_p \rangle$ in the schedule graph such that J_j is the label of the edge from v_i to v_p and $t \in [e', l']$, where e' and l' are given by Equations* (3) *and* (5)*, respectively.*

D. Obtaining Exact Worst- and Best-Case Response Times

The *response time* of a job J_j that completes at time t is $t - r_j^{min}$. Based on Theorem 1, it is easy to obtain J_j's worst- and best-case response times (WCRT and BCRT, respectively): simply check all edges that have J_j as a label. More precisely, let A_j be the set of vertices that have an *outgoing* edge with label J_j. The exact BCRT and WCRT of J_j are given by:

$$BCRT_j = \min\{e' \mid v_i \in A_j\} - r_j^{min} \tag{7}$$

$$WCRT_j = \max\{l' \mid v_i \in A_j\} - r_j^{min} \tag{8}$$

where e' and l' are obtained from (3) and (5) based on any path from v_1 to v_i. $BCRT_j$ and $WCRT_j$ can be incrementally computed as a part of Algorithm 2's expansion phase.

In a hard real-time context, a job set is schedulable under a given scheduling policy if $WCRT_j \leq d_j - r_j^{min}$ for every $J_j \in \mathcal{J}$. However, our approach is oblivious to the underlying notion of temporal correctness; for instance, Equation (8) also yields an exact *tardiness bound*: $\max\{0, WCRT_j - d_j + r_j^{min}\}$.

For example, in Fig. 4, for J_2, we have $A_2 = \{v_3, v_4\}$, and hence $WCRT_2 = \max\{12, 24\} - 10 = 14$. Our analysis thus shows that, since $WCRT_2 = 14 > d_2 - r_2^{min} = 10$, J_2 is not hard real-time schedulable under NP-EDF. However, a maximum tardiness of 4 can be guaranteed.

E. Applicability to Periodic Tasks

While we have focused so far on finite job sets for the sake of clarity, our work is motivated by, and intended for, the analysis of periodic real-time workloads. We briefly explain how our analysis can be applied in this context.

A periodic task generates an infinite sequence of jobs. Thus, in order to apply our job-set-based analysis to a set of periodic tasks, we need to first generate a finite set of jobs that represents an interval of time in which the release pattern of all tasks repeats. We call this representative timeframe the *observation interval* (OI). If the set of jobs in the OI can be shown to be schedulable, then the infinite sequence of jobs is schedulable as well. The choice of OI depends on the type of workload.

First, consider constrained-deadline tasks. For periodic tasks without release offset, the OI is the hyperperiod of the tasks, denoted by H. While in the worst case the number of jobs in H is exponential in the number of tasks, typical task sets found in industry exhibit usually only a few hundred to a few thousand jobs in a hyperperiod (e.g., see [11, 12] for examples).

For periodic tasks with release offsets, the OI is $2H$ if the release offset of each task is an integer multiple of its period and smaller than H. Otherwise, for work-conserving schedulers, the OI is $2H + O^{max}$ [13], where O^{max} is the maximum offset. For non-work-conserving algorithms, however, the problem of choosing a safe OI given arbitrary offsets is largely open.

For the case of periodic tasks with arbitrary deadlines, Goossens et al. [13] recently derived an upper bound on the length of the OI that holds for any deterministic scheduling algorithm. It includes both work-conserving and non-work-conserving algorithms. However, the provided bound grows rapidly with the number of tasks and is likely impractical.

F. Time Complexity

Finally, we briefly remark on the computational complexity of our approach. First, in the context of periodic tasks (i.e., if the input is n tasks), it clearly depends on the number of jobs in a hyperperiod, and hence is inherently exponential in n in the worst case. (However, again, in practice one is unlikely to find workloads in which all periods are relatively prime.)

In the context of finite job sets (i.e., if $n = |\mathcal{J}|$), the runtime complexity of Algorithm 2 is more interesting, and largely depends on how effective the merge phase is (assuming that the IIP function L has polynomial runtime complexity). Intuitively, if all paths with the same set of jobs merge, the maximum number of concurrently explored paths can be described by a binomial coefficient that models the number of distinct combinations of unique items (jobs) in the paths, as illustrated in Fig. 5. However, showing that, or precisely under which conditions, all such paths merge is nontrivial in the presence of IIPs. Due to space constraints, we omit a detailed proof and instead focus on the performance of the analysis in practice.

IV. Empirical Evaluation

We conducted experiments to answer two main questions: **(i)** does our exact test offer significant schedulability improvements? And **(ii)** is the runtime of our analysis practical?

We applied Algorithm 2 to four scheduling policies: work-conserving NP-FP and NP-EDF scheduling, and the two non-work-conserving policies P-RM [9] and CW-EDF$^+$, where the latter is a slightly tweaked version of CW-EDF [10] that adds support for release jitter, as discussed in the appendix.

As a baseline, we use the classic RTA of Davis et al. [2] for NP-FP (denoted by *NP-FP classic test*). Regarding Jeffay's classic test for NP-EDF [14], since it evaluates *all points in time* across a large test interval, it was not sufficiently scalable for our experiments, which use nanosecond resolution.

In our experimental setup, we closely followed the description of an automotive benchmark application [12], where each task is a sequence of functions, called *runnables*, which are activated in series. All runnables in a task have the same period, which is chosen from $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ (milliseconds). Kramer et al. [12] provide a realistic (non-uniform) distribution of runnables across these periods and statistics on BCETs and WCETs of runnables with a given period.

To randomly generate a task set with a given utilization U, we first randomly generated as many runnables as needed to reach the target utilization, following the distribution of periods, BCETs, and WCETs reported by Kramer et al. [12], and then packed runnables of the same period into tasks. Since a necessary schedulability condition for non-preemptive tasks is that $\forall i, 2 \leq i \leq n : C_i^{max} \leq 2(T_1 - C_1^{max})$, where T_1 is the shortest period and n is the number of tasks, we first chose a *packing threshold* a_i uniformly at random from $(0, 2(T_1 - C_1^{max})]$ and then aggregated as many runnables into a task until the threshold a_i was reached. If more runnables with the same period remained at this point, we created another task, chose a new threshold a_i, and repeated the process until all runnables were assigned to tasks. Tasks were assumed to have implicit deadlines. We considered three scenarios for release jitter of the tasks: **(i)** *no jitter*, **(ii)** *small jitter*, i.e., each task has a jitter value that is randomly chosen from $[0, 100]$ microseconds, and **(iii)** *large jitter*, i.e., the release jitter of each task is drawn uniformly at random from $[0, 0.2 \times T_i]$. Cases (ii) and (iii) roughly represent jitter magnitudes that can expected due to interrupt handling and network congestion delays, respectively.

In the experiments, a task set was claimed unschedulable as soon as either an execution scenario with a deadline miss is found or a timeout of four hours was reached. Experiments were conducted on a host equipped with an Intel Xeon E7-8857 v2 processor clocked at 3 GHz and 1.5 TiB RAM; the analysis was implemented as a single-threaded C++ program[2]. Fig. 6 reports the observed schedulability ratio, the average number of states (vertices) of the graph, and the average and individual runtime of the analysis. In total, we generated 9,000 task sets for this experiment (1,000 each for $U \in \{0.1, \ldots, 0.9\}$).

[2]Available at: https://people.mpi-sws.org/~bbb/papers/details/rtss17

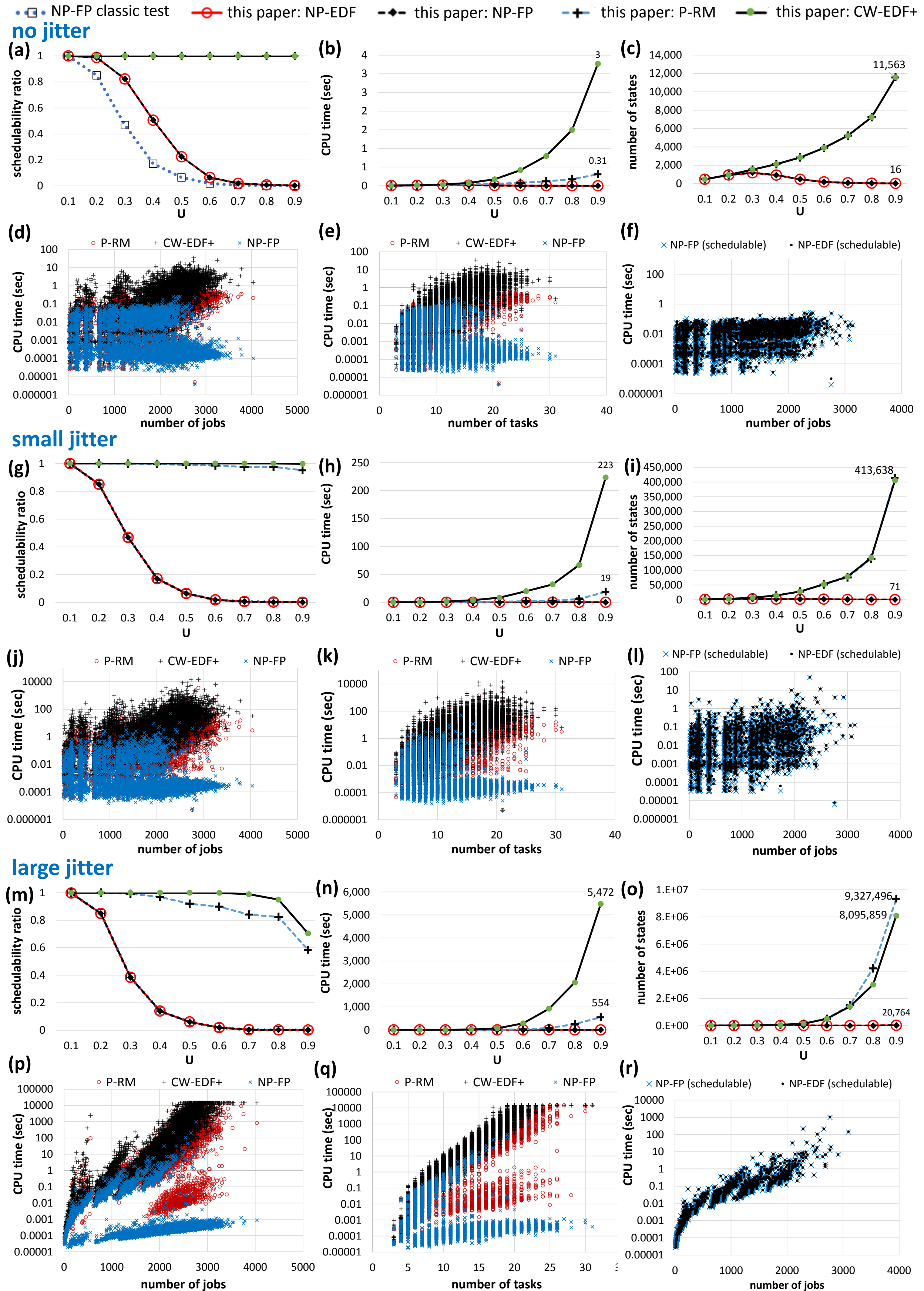

Fig. 6. Experimental results for task sets with zero, small, and large jitter. **(a, g, m)** Schedulability ratio. **(b, h, n)** Average analysis runtime. **(c, i, o)** Average number of states in the graph. **(d, j, p)** Analysis runtime vs. the number of jobs in a hyperperiod. **(e, k, q)** Analysis runtime vs. the number of tasks. **(f, l, r)** Analysis runtime of the NP-FP and NP-EDF analyses for schedulable task sets.

Figs. 6-(a), (g), and (m) show the schedulability ratio as a function of total utilization U when tasks have zero, small, and large jitter, respectively. First, note that the difference between our new exact and the prior sufficient RTA for NP-FP exceeds 20% in the range from 0.3 to 0.5 when there is no release jitter, which shows that our analysis is able to reclaim pessimism in the state-of-the-art analysis for periodic tasks.

Another key observation is that CW-EDF$^+$and P-RM are much more efficient even when task sets are subject to release jitter and execution time variation (prior studies on CW-EDF and P-RM have not considered either type of uncertainty). This result shows that non-work-conserving algorithms are significantly more sustainable than work-conserving algorithms since they are less susceptible to anomalies, handle blocking times more carefully, and can hence correctly schedule many more workloads. Prior to this work, no response-time bounds were knownxzx z for periodic tasks under P-RM or CW-EDF.

As is apparent from Figs. 6-(b), (h), and (n), the runtime of the analyses increases with the increase in jitter. When jobs are subject to large jitter, more eligible jobs appear in the graph expansion phase and a (much) larger number of possible interleavings must be considered, and hence there will be many more vertices (as it can be seen in Figs. 6-(c), (i), and (o)).

Figs. 6-(d), (j), (p) and (e), (k), (q), depict the exact runtime of the analyses for each task set versus the number of jobs in a hyperperiod and the number of tasks in the task set, respectively. As the number of jobs increases, the maximum and average runtimes of the analyses increase. In the case of workloads with large jitter and many tasks, the CW-EDF$^+$analysis sometimes did not complete within the four-hour timeout; task sets for which this happened were reported as unschedulable. That is why there is a drop in the schedulability of CW-EDF$^+$in Fig. 6-(m). In particular, about 22% of the task sets generated with $U = 0.9$ exceeded the timeout. The timeout is also apparent in the scatter plots shown in Figs. 6-(p) and (q), which exhibit a sharp boundary at 14,400 seconds (i.e., four hours).

Figs. 6-(f), (l), and (r) show the runtimes of the NP-FP and NP-EDF analyses for only schedulable task sets (which require the exploration of the entire graph, i.e., no early exit). As can be seen, our analysis for these algorithms finishes considerably faster than for CW-EDF$^+$due to the additional cost of calculating the JIIP function during the expansion phase, and since CW-EDF$^+$achieves better schedulability for higher-utilization workloads (which necessitates the exploration of many more states). Overall, the observed runtimes for systems with small or zero jitter range from less than a second to a few minutes. In the case of workloads with large jitter, the analysis can take multiple hours for complex workloads, which however can still be acceptable for an offline, design-time analysis. Regarding memory use, the analysis required only a few dozen to a few hundreds megabytes of RAM for most task sets; the peak memory use was about 26.9 GiB (for CW-EDF$^+$).

To evaluate the effect of the path-merging phase, we selected a representative task set with $U = 0.3$ and a moderate number of jobs in the hyperperiod. We performed the analysis for NP-FP with and without path merging on increasingly larger

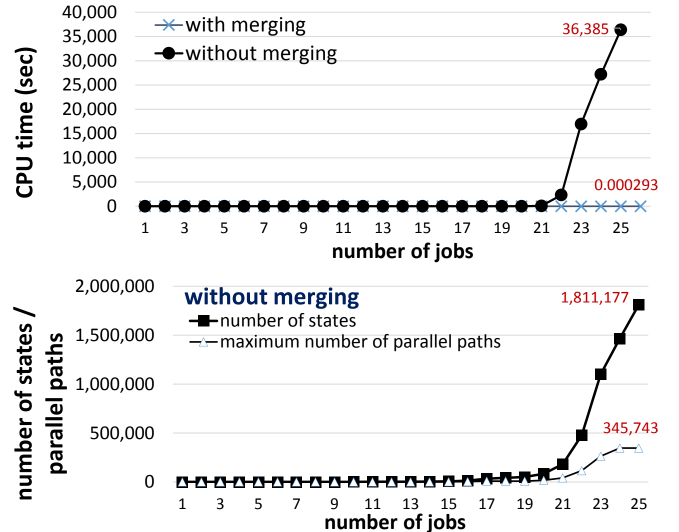

Fig. 7. Impact of the path-merging phase. **(a)** Runtime of the analyses. For 26 or more jobs, the naïve analysis without the path-merging phase exceeded the 12-hour timeout. With the path-merging phase enabled, the algorithm required less than 20 milliseconds to analyze all 631 jobs in the hyperperiod. **(b)** Number of states and parallel paths without the path-merging technique.

subsets of the job set, using a per-analysis timeout of twelve hours. Fig. 7-(a) shows the runtime of both setups as a function of the number of jobs being processed; Fig. 7-(b) shows the number of states and the number of parallel paths in the graph for the setup without path merging. While *with* merging all 631 jobs in the hyperperiod could be analyzed in less than 20 milliseconds, the analysis *without* merging exceeded the timeout of twelve hours after analyzing the first 26 jobs in the hyperperiod. Clearly, the proposed path-merging technique plays an essential role in deferring the state-space explosion.

Finally, we conducted another experiment to better understand the effect of the number of jobs on the performance of Algorithm 2. In this experiment, each task set consisted of eight tasks and periods were selected randomly from $[1, 1000]$ milliseconds following a log-uniform distribution, which results in non-harmonic task sets with large hyperperiods. Each generated task set was sorted in order of increasing periods and a randomly selected execution time C_1^{max} was assigned to the task with the shortest period T_1. We then selected $C_i^{max} \in [0.001, 2(T_1 - C_1^{max})]$ uniformly at random as the WCET for each other task. Deadlines were equal to the period and C_i^{min} was chosen uniformly at random from $[0.5C_i^{max}, C_i^{max}]$. We considered the scenario with small release jitter (as explained earlier) and varied the maximum number of jobs per hyperperiod from 10,000 to 100,000.

Fig. 8 shows the schedulability ratio and average and individual runtimes of the four analyses. As can be seen, even when there are up to 100,000 jobs per hyperperiod, the runtimes do not exceed the four-hour timeout, and in fact most analyses complete within a few minutes. This shows that the runtime of Algorithm 2 is much more dependent on the (relative) magnitude of jitter, rather than the absolute number of jobs.

Interestingly, P-RM exhibits much reduced schedulability in Fig. 8-(a). The reason for this drop in performance is that

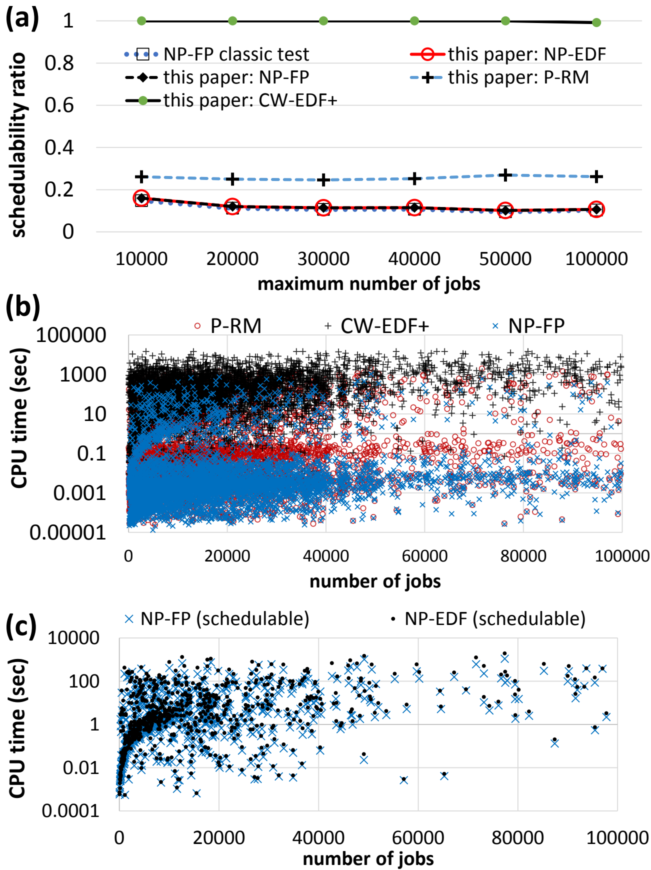

Fig. 8. Experimental results for non-harmic periodic task sets with small jitter and long hyperperiods. **(a)** Schedulability ratio, **(b)** runtime of the proposed analyses for NP-FP, P-RM, and CW-EDF$^+$versus the number of jobs in a hyperperiod, and **(c)** runtime of the proposed analysis for NP-FP and NP-EDF for only schedulable task sets. Note the logarithmic scale in insets (b) and (c).

the P-RM IIP is designed to protect *the* task with the shortest period, whereas the generated workloads sometimes contained *multiple* tasks with the same short period T_1, which the P-RM IIP then failed to consider; an opportunity for future work.

Overall, we conclude that the proposed analysis is practical for realistic workload sizes, that for workloads without jitter the proposed exact analysis is significantly more accurate than prior analyses for sporadic tasks, and that non-work-conserving schedulers are substantially more robust to scheduling anomalies, and thus can deliver much higher schedulability, than classic work-conserving policies.

V. Related Work

Non-preemptive scheduling with release and due dates has been considered by many authors (see [15] for an overview). However, these works typically focus on finding an *offline* schedule that optimizes some objective function (e.g., maximum tardiness) rather than verifying the schedulability of the job set under an existing *online* policy.

Prior exact schedulability tests for non-preemptive policies have been introduced by Jeffay et al. [14] (for NP-EDF) and by Tindell et al. [3] and Davis et al. [2] (for NP-FP). Although these tests are sustainable w.r.t. execution time variation, they are exact only for sporadic tasks, where only the minimum inter-arrival time of jobs is known. As shown previously [10] and in Fig. 6-(a), these tests are pessimistic for periodic tasks. Poon and Mok [16] investigated conditions for non-preemptive sustainability w.r.t. varying periods and execution times.

Sun et al. [17] proposed three sufficient schedulability tests for NP-FP scheduling and tasks with preemptive and non-preemptive execution segments. Their test considers both periodic and non-periodic tasks with given offsets (but not with release jitter). These tests carefully characterize the blocking caused by any lower-priority tasks that can start its execution before the job under analysis. However, since they do not precisely account for the earliest and latest finish times of lower-priority jobs, the test cannot reliably filter all impossible cases and hence is not exact and potentially pessimistic.

Stigge and Yi [18] proposed a sufficient schedulability test for preemptive and non-preemptive *digraph* tasks. A digraph task has a set of modes (with different periods or execution times), between which it transitions at runtime in a non-deterministic way. The schedulability analysis proposed by Stigge and Yi [18] is based on searching all possible scenarios while gradually pruning the number of test cases that will certainly not lead to a deadline miss. Using this method, experiments were conducted using task sets with up to 20 tasks. Our approach differs from theirs [18] in three main aspects: first, we consider a different workload model that incorporates release jitter (which drastically increases the number of possible interleavings); second, our approach is not based on post-hoc pruning, but rather on the early merging of matching paths; third, we provide an exact analysis; and fourth, our approach works even for non-work-conserving policies.

An earlier exact schedulability test was introduced by Baker and Cirinei [7] for preemptive sporadic tasks scheduled by global EDF. The test uses a finite state machine that models all possible combinations of arrival times and execution sequences. As reported by the authors, the method can handle only tasks with periods chosen from $\{3, 4, 5\}$ due to an early state-space explosion. Bonifaci and Marchetti-Spaccamela [8] and Burmyakov et al. [6] later improved this technique, however, without substantially altering its practical scalability limitations.

Guan et al. [4] used a model-checking approach based on timed automata to analyze sporadic tasks under global FP scheduling; their method was shown to scale acceptably only as long as tasks have periods in the range from 8 to 20. Similarly, Sun and Lipari [5] proposed an exact schedulability analysis for a set of *preemptive* sporadic tasks under global FP scheduling on a multiprocessor using hybrid automata. To improve scalability, they provide a set of sound pruning rules. According to the reported evaluation [5], the analysis can handle up to 7 tasks and 4 processors before timing out. Although these works are similar to our proposal in that they seek to explore the space of all possible schedules, they leverage general-purpose formal methods that are known to scale poorly, whereas we have developed a much narrower, problem-specific solution that scales much better. Furthermore, we have focused on uniprocessor scheduling; an extension to multiprocessors remains an interesting avenue for future work.

To the best of our knowledge, this work provides the first *exact* schedulability analysis of sets of non-preemptive jobs (or periodic tasks) that both **(i)** is sustainable with regard to release jitter and execution time variation and **(ii)** scales well to relatively large job sets, e.g., in our experiments, we were able to analyze workloads with up to 35 tasks or up to 100,000 jobs in their hyperperiod. Moreover, this work provides the first general—and even exact—schedulability analysis for non-work-conserving FJP policies based on idle-time insertion.

VI. Summary and Conclusion

We have introduced an exact and sustainable schedulability analysis for non-preemptive jobs under work-conserving and non-work-conserving fixed-job-priority scheduling, taking into account both release jitter and execution times variation.

Our analysis is based on a graph that reflects all possible execution scenarios and resulting job sequences. To cope with the large state space, the analysis greedily prunes the graph by merging similar paths. Importantly, the merge operation retains sufficient information to obtain exact BCRTs and WCRTs. Experiments with randomly generated workloads designed to resemble an automotive benchmark show that task sets with up to 35 tasks can be analyzed with our method with runtimes ranging from less than a second to a few hours, depending on the choice scheduling policy and the amount of jitter.

We plan to extend this method to non-preemptive global scheduling on identical multiprocessors and to take into account explicit precedence constraints. Moreover, as the graph reflects all possible job sequences, it may be possible to use this information to more accurately characterize the cache and microarchitectural processor state before a job commences execution, which could enable more accurate timing analysis.

References

[1] S. Baruah and A. Burns, “Sustainable scheduling analysis,” in *RTSS*, 2006, pp. 159–168.

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

[3] K. W. Tindell, A. Burns, and A. J. Wellings, “An extendible approach for analyzing fixed priority hard real-time tasks,” *Real-Time Syst.*, vol. 6, no. 2, pp. 133–151, 1994.

[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, pp. 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 Syst.*, vol. 52, no. 3, pp. 323–355, 2016.

[6] A. Burmyakov, E. Bini, and E. Tovar, “An exact schedulability test for global FP using state space pruning,” in *RTNS*, 2015.

[7] T. P. Baker and M. Cirinei, “Brute-Force Determination of Multiprocessor Schedulability for Sets of Sporadic Hard-Deadline Tasks,” in *OPODIS*, 2007, pp. 62–75.

[8] V. Bonifaci and A. Marchetti-Spaccamela, “Feasibility analysis of sporadic real-time multiprocessor task systems,” in *ESA*. Springer, 2010, pp. 230–241.

[9] M. Nasri and M. Kargahi, “Precautious-RM: a predictable non-preemptive scheduling algorithm for harmonic tasks,” *Real-Time Syst.*, vol. 50, no. 4, pp. 548–584, 2014.

[10] M. Nasri and G. Fohler, “Non-work-conserving non-preemptive scheduling: motivations, challenges, and potential solutions,” in *ECRTS*, 2016, pp. 165–175.

[11] S. Anssi, S. Kuntz, S. Gérard, and F. Terrier, “On the gap between schedulability tests and an automotive task model,” *Journal of Systems Architecture*, vol. 59, no. 6, pp. 341–350, 2013.

[12] S. Kramer, D. Ziegenbein, and A. Hamann, “Real world automotive benchmark for free,” in *WATERS*, 2015.

[13] J. Goossens, E. Grolleau, and L. Cucu-Grosjean, “Periodicity of real-time schedules for dependent periodic tasks on identical multiprocessor platforms,” *Real-Time Syst.*, vol. 52, no. 6, pp. 808–832, 2016.

[14] K. Jeffay, D. F. Stanat, and C. U. Martel, “On non-preemptive scheduling of period and sporadic tasks,” in *RTSS*, 1991.

[15] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 4th ed. Springer-Verlag New York, 2016.

[16] W. C. Poon and A. K. Mok, “Necessary and Sufficient Conditions for Non-preemptive Robustness,” in *RTCSA*, 2010, pp. 349–354.

[17] J. Sun, M. K. Gardner, and J. W. S. Liu, “Bounding completion times of jobs with arbitrary release times, variable execution times and resource sharing,” *IEEE Transactions on Software Engineering*, vol. 23, no. 10, pp. 603–615, 1997.

[18] M. Stigge and W. Yi, “Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities,” in *Real-Time Syst.*, vol. 51, no. 6, 2015, pp. 639–674.

Appendix: JIIP for CW-EDF$^+$

Critical-window EDF (CW-EDF) [10] is a non-work-conserving scheduling algorithm that is designed for periodic task sets without release jitter. We apply a few changes to accommodate release jitter and to make the resulting IIP compatible with our notion of a stable JIIP. To avoid confusion, we refer to this tweaked variant of CW-EDF as CW-EDF$^+$.

As explained in Sec. III-E, we assume that $\mathcal{J}$ is partitioned into disjoint subsets, called tasks. We let τ_i denote the task to which job J_i belongs.

For a given job J_i, the CW-EDF$^+$ JIIP is defined as follows. First, define a set of *influencing jobs*, denoted by $\mathcal{I}$, which may be affected by J_i. These jobs are selected as follows: for each task τ_x (except τ_i), the influencing job J_x is the first not-scheduled job of τ_x in $\mathcal{J} \setminus \mathcal{J}^S$ (if any) that satisfies

$$J_x \in \mathcal{J} \setminus \mathcal{J}^S \wedge J_x \neq J_i \wedge r_x^{min} = \min\{r_z^{min} | \tau_z = \tau_x\}.$$

Note that $r_x^{min} < t$ is possible. Task without remaining pending jobs are simply omitted from consideration.

Next, the jobs in $\mathcal{I}$ are sorted by deadline in ascending order. Let I_i be the job index of the ith job in $\mathcal{I}$. The latest start time for each influencing job is then calculated as:

$$\beta_{I_i} = \begin{cases} d_{I_i} - C_{I_i}^{max} & I_i = |\mathcal{I}|, \\ \min\{\beta_{I_{i-1}}, d_{I_i}\} - C_{I_i}^{max} & \text{otherwise.} \end{cases} \quad (9)$$

If there is no influencing job, then β_{I_1} is simply set to ∞. Finally, the function L is defined as $L(J_i, t, \mathcal{J}^S) \triangleq \beta_{I_1} - C_i^{max}$. Note that for a given job J_i, as long as $\mathcal{J}^S$ remains unchanged, the set of influencing jobs remains constant regardless of the value of t. The CW-EDF$^+$ JIIP is hence stable.