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 20+ 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 changing system modes. 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-RM* (P-RM) [9] and *Critical-Window EDF* (CW-EDF) [10], 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 leaves considerable room for improvement in the analysis of non-preemptive periodic real-time workloads on uniprocessors.

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 *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 20 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 assume that any ties in priorities will be broken in a first-in-first-out (FIFO) manner with respect to job release times. For ease of notation, we assume that the "$<$" operator reflects the 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 non-repeated 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 $r_i \in [r_i^{min}, r_i^{max}]$.

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

Algorithm 1: IIP-Aware FJP Scheduler

```
Input: $t$: the current time, $f$: the IIP, $\mathcal{J}^S$: completed jobs
1 $J_i \leftarrow$ the highest-priority pending job;
2 if $f(J_i, t, \mathcal{J}^S) = true$ then
3     Schedule $J_i$;
4 else
5     Idle the processor until a new job is released;
6 end
```

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 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 consists of two components. The first is the *JIIP predicate* $f(J_i, t, \mathcal{J}^S)$, which is used by the scheduler to decide whether the highest-priority pending job J_i is allowed to be scheduled at time t (see line 2 in Algorithm 1), given that the set of jobs denoted by $\mathcal{J}^S$ have already finished.

The second component is a function $g(J_i, t, \mathcal{J}^S)$ that yields the *latest permissible start time* of J_i after t, again given that the jobs in $\mathcal{J}^S$ have been scheduled before time t.

The latest permissible start time is simply the last time at which f will allow J_i to be scheduled (for a fixed $\mathcal{J}^S$), i.e., $t_I = g(J_i, t, \mathcal{J}^S)$ is the maximum t_I such that $f(J_i, t_I, \mathcal{J}^S) =$ *true* and $f(J_i, t_I{+}1, \mathcal{J}^S) =$ *false*. If J_i is already not permitted to run at time t, then $g(J_i, t, \mathcal{J}^S) < t$, and if f will never block J_i, then $g(J_i, t, \mathcal{J}^S) = \infty$.

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, and f and g to be *consistent*, i.e., g must truthfully describe f. As this is key to our analysis, we provide a formal definition.

Definition 2. A predicate $f : \mathcal{J} \times \mathbb{N} \times \mathcal{P}(\mathcal{J}) \mapsto \{\mathit{true}, \mathit{false}\}$ and a function $g : \mathcal{J} \times \mathbb{N} \times \mathcal{P}(\mathcal{J}) \mapsto \mathbb{N}$ form a JIIP iff f and g are *stable* and *consistent*, i.e., for any J_i, t, and $\mathcal{J}^S$:

(i) if $f(J_i, t, \mathcal{J}^S) =$ *true*, then J_i remains eligible until its latest permissible start time $t_I = g(J_i, t, \mathcal{J}^S)$; formally,

$$\forall t' \in [t, t_I] : f(J_i, t', \mathcal{J}^S) = \mathit{true} \wedge g(J_i, t', \mathcal{J}^S) = t_I; \quad (1)$$

(ii) if $f(J_i, t, \mathcal{J}^S) =$ *false*, then J_i remains ineligible until at a time t_R a higher-priority job is certainly released; formally

$$\forall t' \in [t, t_R] : f(J_i, t', \mathcal{J}^S) = \mathit{false}, \quad (2)$$

where t_R is given by

$$t_R = \min \left\{ r_x^{max} \;\middle|\; J_x \in \mathcal{J} \setminus \mathcal{J}^S \wedge p_x < p_i \wedge t \leq r_x^{max} \right\} \quad (3)$$

if such a job exists. If no such job exists, then simply $t_R = \infty$, which means that J_i remains blocked indefinitely, which can happen in an overloaded and thus unschedulable system.

Definition 2 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 be the job (if any) that satisfies the condition

$$r_x^{max} = \min \left\{ r_z^{max} \mid J_z \in (\mathcal{J} \setminus \mathcal{J}^S) \wedge p_z = \hat{p} \wedge t \leq r_z^{max} \right\}.$$

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

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

Recall that $g(J_i, t, \mathcal{J}^S) = \infty$ if f never blocks J_i, which under P-RM is the case if there is no future higher-priority job that must be "protected." The corresponding predicate f is then defined simply as $f(J_i, t, \mathcal{J}^S) = t \leq g(J_i, t, \mathcal{J}^S)$. That is, a lower-priority job J_i may start execution at time t only if there remains sufficient slack, i.e., if it cannot cause undue blocking to a later-arriving maximum-priority job. The JIIP of CW-EDF [10] is conceptually similar, but notationally more involved, in part since CW-EDF assumes tasks without release jitter; for brevity, we relegate these details to 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 JIIP as defined in Definition 2, 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 $f(J_i, t, \mathcal{J}^S) = \mathit{true}$ and $g(J_i, t, \mathcal{J}^S) = \infty$.

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 where all execution costs

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

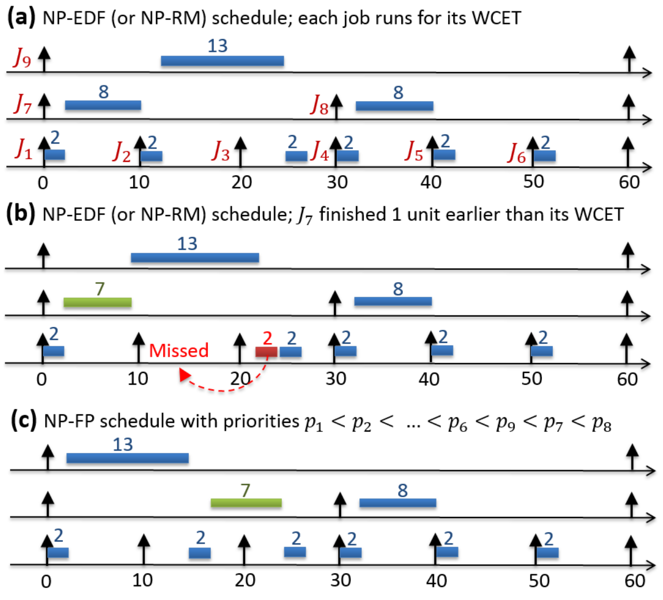
(a) NP-EDF (or NP-RM) schedule; each job runs for its WCET
J_9
13
J_7
8
J_8
8
J_1
J_2
J_3
J_4
J_5
J_6
2
0
10
20
30
40
50
60
(b) NP-EDF (or NP-RM) schedule; J_7 finished 1 unit earlier than its WCET
13
7
8
Missed
2
(c) NP-FP schedule with priorities $p_1 < p_2 < \ldots < p_6 < p_9 < p_7 < p_8$
13
7
8
2

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.

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).

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

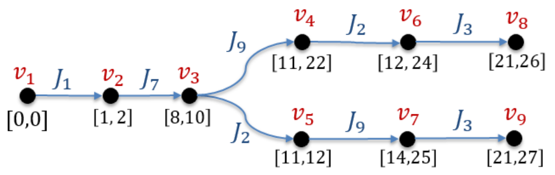

Fig. 2. A schedule graph for the job set in Fig. 1 scheduled by NP-EDF.

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 whether the total sum of execution costs $C_1 + C_7$ is at most 9: if $C_1 + C_7 > 9$, 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 happens when $C_1 = 1$, $C_7 = 7$, and $C_9 = 3$, while the EFT of v_5 happens 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 in $e_k \in E$ is a job $J_x \in \mathcal{J}$ and the label of each vertex $v_i \in V$ is an interval 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. 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

Algorithm 2 shows the proposed schedule graph generation algorithm (SGA), which constructs the graph iteratively. It starts with an *initialization phase* (line 1), in which a root vertex v_1 labeled with the interval $[0, 0]$ is created. This is followed by a repeating *expansion phase* (lines 2-10), in which (one of) the shortest path(s) is grown by considering all jobs that can appear next in the sequence represented by the path.

For each such job J_j, we create a destination vertex that is labeled with an EFT and an LFT based on the execution and release time of J_j (lines 7–10). If no such job exists, then the job set is unschedulable and the expansion aborts (lines 4–6).

Finally, after each iteration of the expansion phase, there is a *merging phase* (lines 12–14), where 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.

The expansion and merging phases repeat until every path in the graph contains $|\mathcal{J}|$ distinct jobs. In the remainder of this section, we discuss these two phases in detail.

1) Expansion phase: Fig. 3-(a) 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

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

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 3. A job J_j is *priority-eligible* at time t iff

$$p_j = \min \left\{ p_x \;\middle|\; J_x \in \mathcal{J} \setminus \mathcal{J}^P \;\wedge\; r_x^{max} \leq t \right\}. \quad (4)$$

In other words, J_j is priority-eligible iff it has the highest priority among those jobs that are certainly released by time t but not yet scheduled. For example, consider Fig. 3, where J_5 is the last-scheduled job in the path P that ends in v_i. J_5 can finish at any instant from e_i until l_i. In this example, J_7, J_6, J_3, and J_2 are priority-eligible jobs at some times during $[e_i, l_i]$, because there exist execution scenarios in which any of these jobs becomes the highest-priority pending job.

For instance, if J_5 finishes at 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_6 is released only at time r_6^{max} and J_5 finishes at time e_i, then J_7 has the highest priority at time e_i. Similarly, if J_5 finishes at time r_3^{min} (respectively, time r_2^{min}), then J_3 (respectively, J_2) may become the highest-priority pending job upon its release. 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 consider the IIP.

Definition 4. A job J_j is *IIP-eligible* at time t iff the IIP permits it to be scheduled, i.e., iff $f(J_j, t, \mathcal{J}^P) = true$.

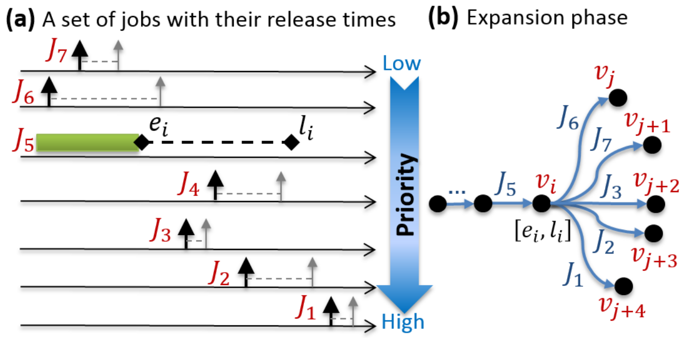

Fig. 3. An example showing how a given path P ending in v_i can be expanded. Note that jobs on lower axes have higher priority. Release jitter is indicated by dashed lines between two up-arrows. The label of v_i, the interval $[e_i, l_i]$, is marked in the figure by a dashed line between two diamond-shaped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

That is, when expanding a path, it is safe to ignore jobs that are certainly blocked by the IIP predicate f.

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 $r_j^{min} \le l_i$), 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 (i.e., if $r_j^{min} > l_i$, for J_j to be scheduled next, the system must be certainly idle from time $l_i + 1$ until time r_j^{min}). We refer to this criterium as "potentially-next eligibility."

Definition 5. A job J_j is *potentially next* at time t iff either:
(i) J_j can be released before l_i, i.e., $r_j^{min} \le l_i$, or
(ii) no other certainly released job can be scheduled before J_j, i.e., $\forall J_x \in \mathcal{J} \setminus \mathcal{J}^P$ s.th. $r_x^{max} \le t$, $\forall t'[l_i + 1, t]$: J_x must not be both priority-eligible and IIP-eligible at t'.

Returning to the example in Fig. 3, for the sake of argument, assume that all jobs are IIP-eligible during $[e_i, l_i]$, and that only J_1 is IIP-eligible during $(l_i, r_1^{min}]$. Jobs J_2–J_7 are trivially potentially-next jobs as they satisfy clause (i) of Definition 5. Next, consider job J_1, which does not satisfy clause (i). However, since the other pending jobs are not IIP-eligible during $(l_i, r_1^{min}]$, J_1 satisfies the second clause. Specifically, J_1 is scheduled directly after J_5 if J_5 finishes at time l_i, as then all other pending jobs are blocked by the IIP and J_1 is scheduled immediately when it is released at time r_1^{min}.

Based on Definitions 3–5, we can precisely characterize the set of potential successor jobs in path P.

Definition 6. 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 $t_S = \max\{e_i + 1, r_j^{min}\}$.

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 $\max\{e_i + 1, r_4^{min}\} = r_4^{min}$ due to the presence of J_3.

We later show in Lemma 3 that Definition 6 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

$$e_k = t_S + C_j^{min}, \tag{5}$$

where $t_S = \max\{e_i + 1, r_j^{min}\}$ 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, since we consider eager policies, J_j must start execution by time $t'_S = \max\{l_i + 1, r_j^{max}\}$ because it will certainly be released by that time, the processor will be available, and for J_j to be next in the sequence, no other job can be scheduled. That is, in any execution scenario in which J_j is scheduled next, it will certainly start by time t'_S.

Second, since we consider JFP policies, J_j must start execution before a higher-priority job (if any) is certainly released, since a higher-priority pending job implies that J_j cannot commence execution. Formally, if such a job exist, J_j's latest start time is bounded by $t_R = \min\{r_x^{max} | J_x \in \mathcal{J} \setminus \mathcal{J}^P \wedge p_x < p_j \wedge t_S \le r_x^{max}\} - 1$, similarly to Equation (3). Otherwise, if no such higher-priority job exists, we set $t_R = \infty$.

Third, it follows from Definition 2 that if the IIP blocks J_j from executing (after the earliest start time t_S), then a higher-priority job must finish before J_j can commence execution. Hence, the latest time after time t_S at which the IIP permits J_j to start, which is given by $t_I = g(J_j, t_S, \mathcal{J}^P)$, 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

$$l_k = \min\{t'_S,\ t_R,\ t_I\} + C_j^{max}. \tag{6}$$

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 7. A path Q from v_1 to v_q is *matching* a given path P from v_1 to v_p iff $\mathcal{J}^P = \mathcal{J}^Q \ \wedge\ [e_q, l_q] \cap [e_p, l_p] \ne \emptyset$, 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:

$$[e_p, l_p] \leftarrow [e_p, l_p] \cup [e_q, l_q]. \tag{7}$$

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.

For example, in Fig. 2, the two paths $P = \langle v_1, v_2, v_3, v_4, v_6\rangle$ and $Q = \langle v_1, v_2, v_3, v_4, v_5, v_7\rangle$ 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: $[12, 24] \cap [14, 25] \ne \emptyset$. The two paths will thus be merged by Algorithm 2, which leads to the final graph shown in Fig. 4.

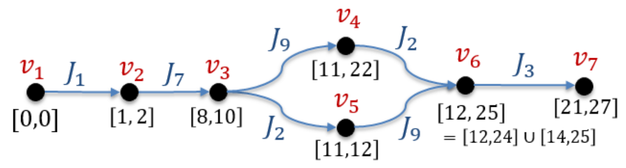

Fig. 4. The schedule graph produced by Algorithm 2 for the job set in Fig. 1 under NP-EDF scheduling. In v_6, the two paths representing the scenarios in Figs. 1-(a) and 1-(b) merge; the vertex label is hence a union of two intervals.

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 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 by red dashed edges, even though there are many eligible successors at the beginning, the size of the graph reduces considerably after several merge steps. While a detailed analysis of the computational complexity of Algorithm 2 is beyond the scope of this paper (see also Sec. III-E), it is interesting to note that, given $n = 4$ eligible jobs in Fig. 5, the number of distinct 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 (5) and (6) are tight (Lemmas 1 and 2), and then show that Definition 6 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* (5) *and larger than* (6)*, respectively.*

Proof. By induction. The base case is for v_1, in which 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, (5) and (6) provide tight bounds on EFT and LFT of all paths from v_1 to v_p.

EFT: The earliest start time of J_j, i.e., t_S, cannot be smaller that $e_i + 1$ since, by the induction hypothesis, e_i is the earliest

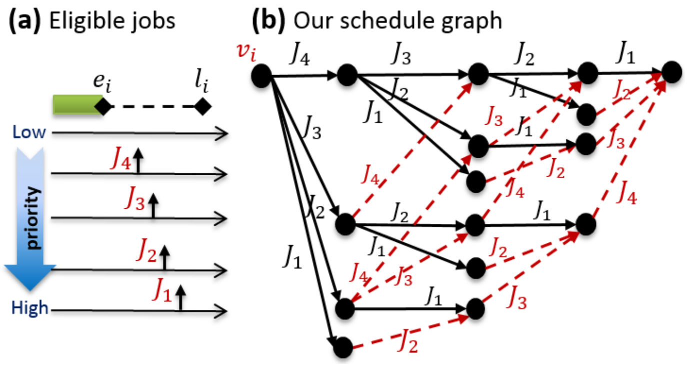

Fig. 5. The schedule graph generated by Algorithm 2 for a pathological case where jobs are released in reverse order of their priorities, the WCET of each job exceeds $l_i - e_i$, and the BCET of each job is zero, which maximizes the number of possible job sequences. For simplicity, there is no release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

time at which all prior jobs in path P can be finished. Moreover, it cannot be earlier than r_j^{min} since J_j arrive prior to being scheduled. If J_j starts its execution at t_S, it cannot finish before (5) since its minimum execution time is C_j^{min}.

LFT: Next, we show that $l_p - C_j^{max}$ cannot be larger than t_I, t_R, or t'_S. Since the scheduling algorithm is FJP, if J_j is not started by t_R, it will no longer be the highest-priority pending job at t_R. As a result, the FJP scheduling algorithm will not allow J_j to be scheduled directly after path P. Hence, J_j will be a direct successor of path P only if its execution starts by t_R. For the same reason, J_j must start its execution by t_I because according to clause (ii) in Definition 2, if $f(J_j, t_I + 1, \mathcal{J}^P) = false$ at time $t_I + 1$, the IIP predicate will remain *false* at least until the next release time of a higher-priority job (t_R).

If $l_i + 1 < r_j^{max} \leq \min\{t_I, t_R\}$, and if J_j is the job that is scheduled next, then J_j must be the highest-priority job at r_j^{max} that is allowed to be scheduled by the IIP. Since the scheduling algorithm under consideration is an *eager* scheduler, J_j must be scheduled at r_j^{max}. Otherwise, if $r_j^{max} \leq l_i + 1 \leq \min\{t_I, t_R\}$, then J_j will be the highest-priority job at $l_i + 1$ and must be scheduled no later than $l_i + 1$. Finally, since J_j can be executed for its longest execution time, the latest finish time of J_j is $l_p = C_j^{max} + \min\{t_I, t_R, \max\{l_i + 1, r_j^{max}\}\}$. □

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

Proof. Vertex labels are created in line 8 of Algorithm 2 and modified in line 12 of Algorithm 2, where in the latter case the union of two *intersecting* intervals replaces the previous label. Since merge operations trivially maintain the claimed property, and since line 8 is the only place in the algorithm where an interval label is created, it is sufficient to show that each interval created from (5) and (6) satisfies the claim.

This can be shown 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 job that labels the edge from v_i to v_p, and suppose the claim holds for v_i. From (5) we have

$e_p = t_S + C_j^{min}$ and from (6) we have $e_l \leq t'_S + C_j^{max}$, which translates into four cases: **(i)** $l_i < r_j^{min}$; **(ii)** $r_j^{max} \leq e_i$. **(iii)** $r_j^{min} \leq e_i \leq l_i < r_j^{max}$; and **(iv)** $e_i < r_j^{min} \leq r_j^{max} \leq l_i$.

In case (i), $e_p = r_j^{min} + C_j^{min}$ and $e_l \leq r_j^{max} + C_j^{max}$. As the previous job finishes prior to J_j's release, and since any $t \in [r_j^{min} + C_j^{min}, r_j^{max} + C_j^{max}]$ can be obtained by a combination of r_j and C_j, there trivially exists an execution scenario in which job J_j finishes at any time in $[e_p, l_p]$.

In the remainder, we exploit that, by the induction hypothesis, there exists an execution scenario and a corresponding path $\langle v_1, \ldots, v_i \rangle$ such that the last job J_k in the job sequence (which does not include J_j) finishes at any chosen time $t' \in [e_i, l_i]$.

In case (ii), $e_p = e_i + 1 + C_j^{min}$ and $e_l \leq l_i + 1 + C_j^{max}$, so simply let J_j arrive at time r_j^{max} and choose a suitable finish time t' for J_k and execution cost C_j so that $t' + 1 + C_j = t$.

In case (iii), $e_p = e_i + 1 + C_j^{min}$ and $e_l \leq r_j^{max} + C_j^{max}$. This case requires a combination of the previous two constructs. If $t \leq e_l + C_j^{max}$, then choose t' and C_j similarly to case (ii). Otherwise, if $t > e_l + C_j^{max}$, choose any combination of r_j and C_j that yields t as in case (i). Finally, in case (iv), $e_p = r_j^{min} + C_j^{min}$ and $e_l \leq l_i + 1 + C_j^{max}$. Here, one can simply choose t' and C_k such $t' + 1 + C_j = t$. Thus, there exists a path $P' = \langle v_1, \ldots, v_i, v_p \rangle$ and execution scenario such that the last-scheduled job J_j finishes at any time $t \in [e_p, l_p]$. □

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 (6) 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 job (Definition 6).*

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

If: Consider the following execution scenario: each job $J_x \notin \mathcal{J}^P$ 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_S. It is further IIP-eligible, and hence must be allowed to be scheduled by the IIP at time t_S. If $t_S \leq l_i$, then J_j can be scheduled at t_S in an execution scenario in which the last job of path P finishes at $t_S - 1$. The existence of such an execution scenario follows from Lemma 2. Otherwise, if $t_S > l_i$, then the following execution scenario allows J_j to be scheduled after P: the last job of path P finishes its execution at l_i and J_j is released at r_j^{min}. Note that, according to clause (ii) of Definition 5 (which applies if $t_S > l_i$), J_j is an eligible successor of P only if all other jobs are not IIP-eligible and priority-eligible in the interval $(l_i, r_j^{min}]$. 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 6, then there is no execution scenario in which J_j is scheduled after P. If J_j is not IIP-eligible, then according to Definition 2, $f(J_j, t_S, \mathcal{J}^P) = false$ will hold at least until t_R, which is the release time of a high-priority job (if no such job exists, then J_j is indefinitely blocked by the IIP). Since J_j will no longer be the highest-priority pending job at t_R, it cannot be directly scheduled after P. For the same reason, if J_j is not priority-eligible at t_S, it cannot be scheduled after P since there will be another pending job with a higher priority at t_S. Finally, if J_j is not next-eligible, then there exists at least one IIP- and priority-eligible higher-priority job that is released before r_j^{min}. Under an eager FJP scheduler, this job will precede J_j. □

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* (5) *and* (6)*, 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 infer the tight upper and lower response-time bounds: 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:

$$BCRT_j = \min\{e' \mid v_i \in A_j\} - r_j^{min} \quad (8)$$

$$WCRT_j = \max\{l' \mid v_i \in A_j\} - r_j^{min} \quad (9)$$

where e' and l' are obtained from (5) and (6) based on any path from v_1 to v_i. $BCRT_j$ and $WCRT_j$ can be incrementally computed as 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 (9) 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 and Time Complexity

While we have focused so far on finite job sets for the sake of clarity, our work is motivated by, and intended for, the

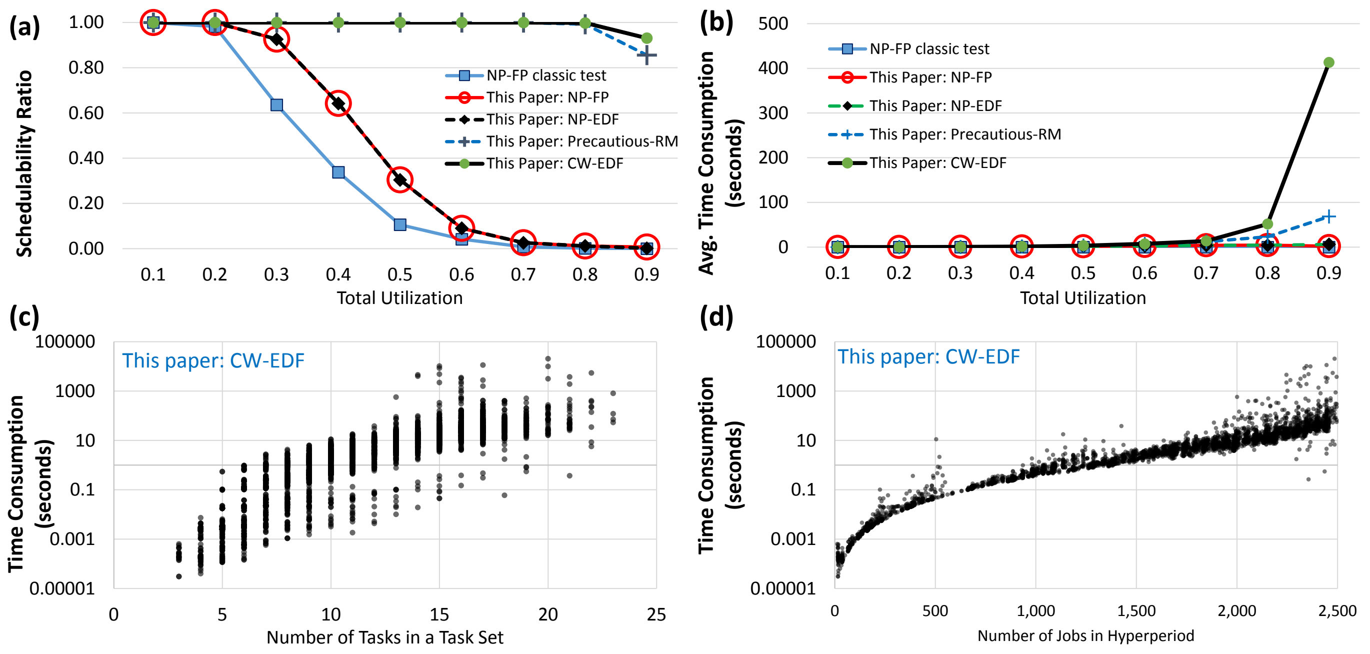

Fig. 6. Experimental results. **(a)** Schedulability under the four policies and the baseline analysis. **(b)** Average runtime of the analyses. **(c)** Runtime of the proposed analysis for CW-EDF$^+$ relative to the number of tasks in a task set. **(d)** Runtime of the proposed analysis for CW-EDF$^+$ relative to the number of jobs in a hyperperiod. In insets (c) and (d), each data point corresponds to one task set. Note the logarithmic scale due to the wide range of observed runtimes.

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 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. Otherwise, for work-conserving schedulers, the OI is $2H + O^{max}$, where O^{max} is the maximum offset [13]. For non-work-conserving algorithms, however, the problem of choosing a safe OI in the presence of 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.

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 functions f and g have polynomial runtimes). 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 from this paper 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 microsecond 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 assigned each task a bounded release jitter of at most 5% of its period. We further conducted additional experiments without release jitter; due to the space limitations these are reported in the appendix (Figs. 9 and 10).

Experiments were conducted on an Intel Xeon E7-8857 v2 machine clocked at 3 GHz with 1.2 TiB RAM; the analysis was implemented as a single-threaded Java program. Fig. 6 reports the observed schedulability ratio as well as the average and individual runtime of the analysis. In total, we generated 9,000 task sets for this experiment (1,000 each for $U \in \{0.1, \ldots, 0.9\}$).

Fig. 6-(a) shows the schedulability ratio under the four considered policies as a function of total utilization U. First, note that the difference between our new exact and the prior sufficient schedulability analysis for NP-FP exceeds 20% in the range from 0.3 to 0.5, which shows that our analysis is able to reclaim considerable pessimism in the state of the art.

Another key observation is that CW-EDF$^+$and P-RM are extremely efficient in scheduling non-preemptive tasks even when they 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 in a more efficient and careful way, and can hence correctly schedule many more workloads. Notably, prior to this work, response-time bounds were available for neither P-RM nor CW-EDF.

Fig. 6-(b) depicts the average runtime of the analyses; we provide a zoomed-in version of this diagram in Fig. 8 in the appendix. In the experiment, we stopped Algorithm 2 as soon as we observed a non-schedulable job, which is why the average time consumption of our NP-FP and NP-EDF schedulability tests remains low at high utilizations. CW-EDF$^+$ exhibits by far the largest runtimes since its IIP function is significantly more expensive to compute. Overall, the observed runtimes range from a few seconds to a few minutes, which is acceptable for an offline, design-time analysis.

Figs. 6-(c) and (d) show the runtime of Algorithm 2 applied to CW-EDF$^+$ w.r.t. the number of tasks in a task set and jobs in a hyperperiod, respectively. Each point corresponds to one generated task set. First observe that, while runtimes increase with an increasing number of tasks (or jobs), the growth is relatively slow in the beginning, and especially with regard to the increase in the number of jobs. This observation emphasizes that the merge step in Algorithm 2 helps significantly in reducing the number of concurrently explored paths.

More experiments on non-harmonic task sets and a larger number of jobs are reported in the appendix. According to our observations, a harmonic (or close to harmonic) task set, such as the one that we used in the experiments discussed in this section, affect the schedulability of an algorithm in two ways.

First, since harmonic tasks release jobs together with all shorter-period tasks, the number of eligible jobs that must be analyzed increases, while in a non-harmonic task set, the complexity is much lower since job releases are not often synchronous. As a result, for non-harmonic tasks, the number of concurrent paths in the graph reduces and the runtime lessens and becomes dominated by the number of jobs in the hyperperiod (rather than the number of tasks).

Second, a harmonic task set with release jitter has a reduced schedulability if the scheduling algorithm is work-conserving because, due to release jitters, a task with a shorter period may be released after and hence blocked by a task with a longer period and (much) larger execution cost. This indicates that industrial task sets that often have harmonic periods can benefit from non-work-conserving schedulers such as CW-EDF$^+$.

Overall, we conclude that the proposed analysis is practical for realistic workload sizes, that the proposed exact analysis for periodic tasks (with jitter) 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, 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.

Stigge and Yi [17] 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. In order to verify the schedulability of such tasks, Stigge and Yi [17] search all possible scenarios while gradually pruning the number of test cases that will certainly not lead to a deadline miss. They have reported experiments with up to 20 tasks. Our approach differs from theirs [17] 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 based for preemptive sporadic tasks scheduled by global EDF based on a finite-state machine modeling all possible combinations of arrival times and execution sequences was introduced by Baker and Cirinei [7]. 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 scheduled on a multiprocessor global FP scheduling 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 powerful, 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, to verify the schedulability of up to 25 tasks with up to 2,500 jobs in their hyperperiod, it took from less than one minute for NP-FP, NP-EDF, and P-RM to on average less than 10 minutes for CW-EDF$^+$. Moreover, it provides the first general—and even exact—schedulability analysis for FJP non-work-conserving 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 25 tasks can be analyzed with our method in a few seconds to a few minutes, depending on the choice scheduling policy.

We plan to extend this method to multiprocessors under non-preemptive global scheduling 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 Systems*, 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 Systems*, 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] M. Stigge and W. Yi, "Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities," in *Real-Time Systems*, vol. 51, no. 6, 2015, pp. 639–674.

Appendix

Idle-Time Insertion Policy for CW-EDF

Critical-window EDF (CW-EDF) [10] is a non-work-conserving scheduling algorithm that is designed for periodic task sets with no release jitter. We apply a few changes to accomodate release jitter and to make the resulting IIP compliant with Definition 2. To avoid confusion, we refer to this slightly tweaked variant of CW-EDF as CW-EDF$^+$.

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

As it is the case with P-RM, the JIIP predicate f of CW-EDF$^+$ is defined as $f(J_i, t, \mathcal{J}^S) = t \leq g(J_i, t, \mathcal{J}^S)$. Next, we explain how g is obtained.

First, define a set of *influencing jobs*, denoted by $\mathcal{I}$, whose schedulability 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 next not-scheduled job of τ_x (if any) that satisfies

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

Note that $r_x^{min} < t$ is possible. If for some task there is no remaining non-scheduled job, the task is simply omitted from consideration.

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

$$\beta_{I_i} = \begin{cases} d_{I_i} - C_{I_i}^{max} & I_i = |\mathcal{I}|, \\ \min\{\beta_{I_{i-1}}, d_{I_i}\} - C_{I_i}^{max} & \text{otherwise.} \end{cases} \tag{10}$$

If there is no influencing job, β_{I_1} is simply set to ∞. Finally, the function g is defined as $g(J_i, t, \mathcal{J}^S) = \beta_{I_1} - C_i^{max}$. 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. As a result, g satisfies Condition (i) in Definition 2.

Additional Results

We present additional, more detailed results regarding the experiment discussed in Sec. IV.

- Fig. 7 illustrates the time consumption of our test for NP-FP and P-RM. As it can be seen in the figure, the test is must faster for NP-FP than in the case of CW-EDF$^+$.
- For convenience, we also report a zoomed-in version of Fig. 6-(b), where we have limited the Y-axis to 100 seconds in Fig. 8. Here, the drop in the average time consumption of NP-FP and NP-EDF is due to the fact that most unschedulable task sets can be identified as such in the earlier stages of graph generation. The time consumption reduces since we stop the graph expansion if we observe a deadline miss in a path after its expansion.
- Fig. 9 illustrates the time consumption of CW-EDF$^+$for random task sets that do not have release jitter. The setup of this experiment is similar to what is explained in Sec. IV and follows the automotive benchmark provided by Kramer et al. [12]. As it can be seen, without release jitter, the time consumption of the algorithm is significantly lower due to the fact that there are fewer eligible successors on average (since jitter introduces many more opportunities to reorder jobs).

In future work, we plan to explore how to further optimize, and potentially parallelize, our implementation of Algorithm 2. We also plan to make our implementation freely available as an open-source library.

Additional Scalability Experiment

To better understand the effect of the number of jobs on the performance of Algorithm 2, we have conducted another experiment on randomly generated periodic task sets. In this experiment, we randomly generated 400 periodic task sets each having three to eight tasks (with a uniform distribution). For this experiment, periods are selected randomly from $[1, 1000]$ following a log-uniform distribution, which results in non-harmonic task sets with extremely large hyperperiods.

Each generated task set was sorted such that T_1 is the smallest period. Next, a random execution time was assigned to the task with the smallest period T_1. With C_1^{max} as the total WCET of the task that has period T_1, we then selected $C_i^{max} \in [0.001, 2(T_1 - C_1^{max})]$ with uniform distribution as the WCET of the other tasks. In this experiment, release jitter was assumed to be at most 5% of a task's period, deadlines were equal to the period, and C_i^{min} was chosen uniformly at random from $[0.5C_i^{max}, C_i^{max}]$. We discarded any task set that had a total utilization exceeding 1 or more than 50,000 jobs in its hyperperiod.

Fig. 10 shows the runtime of our analysis applied to CW-EDF$^+$ as a function of the number of jobs in the hyperperiod. This figure reveals a striking, almost polynomial relation between the runtime of Algorithm 2 and the number of jobs $|\mathcal{J}|$. A formal characterization of the runtime complexity of Algorithm 2 requires a considerable analytical development and is beyond the scope of this paper.

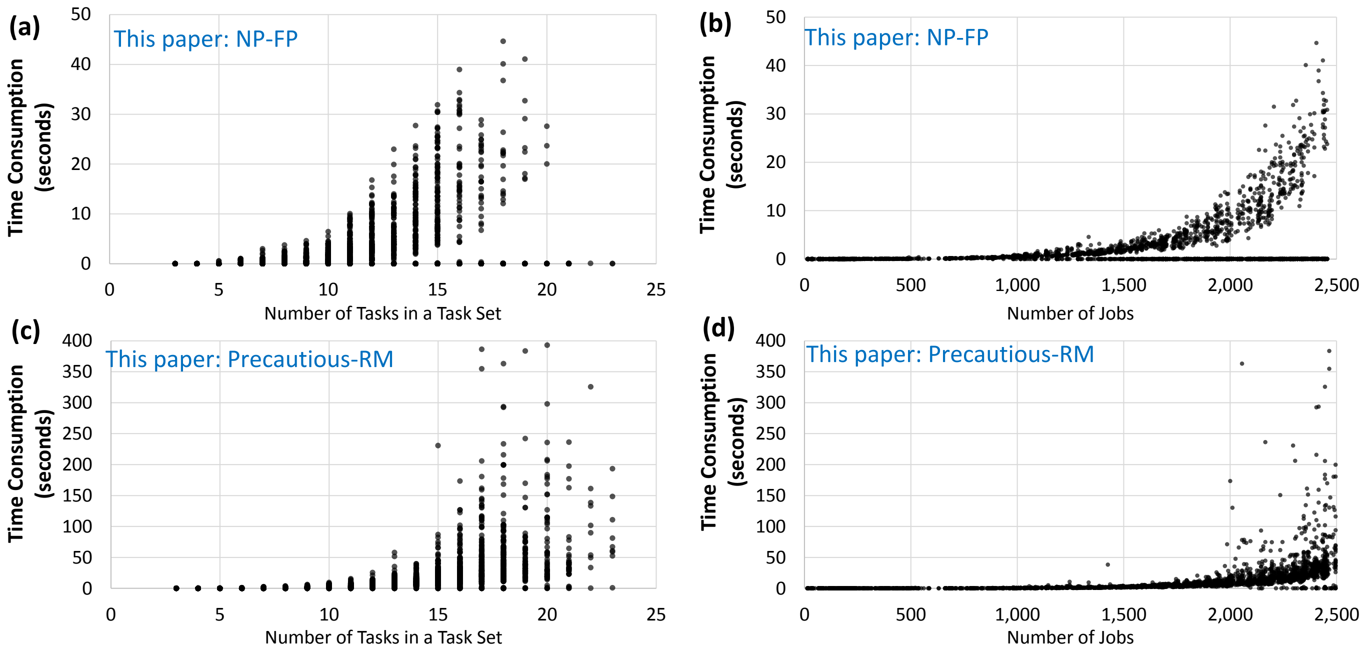

Fig. 7. Time consumption of our test when applied to NP-FP and P-RM in the experiment reported in Sec. IV. Each point corresponds to one task set.

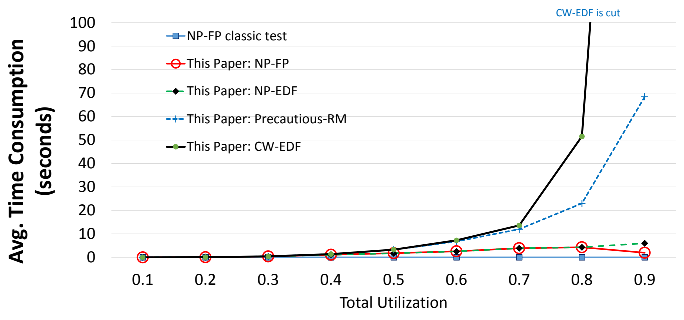

Fig. 8. A zoomed-in version of Fig. 6-(b). The Y-axis is limited to 100s.

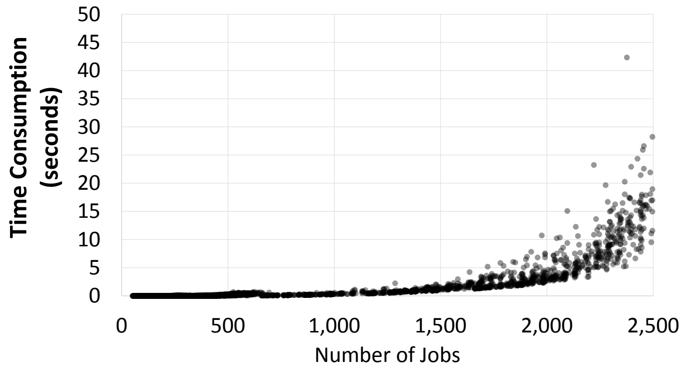

Fig. 9. Runtime of our analysis for CW-EDF$^+$ as a function of the number of jobs in the first experiment without release jitter. Each dot corresponds to one task set; the number of jobs corresponds to one hyperperiod.

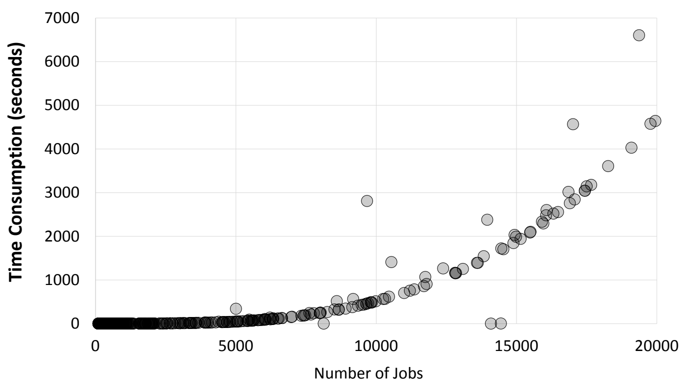

Fig. 10. Runtime of our analysis for CW-EDF$^+$ as a function of the number of jobs in the second experiment with extremely large hyperperiods. Each dot corresponds to one task set; the number of jobs corresponds to one hyperperiod.