Precedence-Aware Automated Competitive Analysis of Real-Time Scheduling

Andreas Pavlogiannis, Nico Schaumberger, Ulrich Schmid, and Krishnendu Chatterjee

***Abstract*—We consider a real-time setting where an environment releases sequences of firm-deadline tasks, and an online scheduler chooses on-the-fly the ones to execute on a single processor so as to maximize cumulated utility. The *competitive ratio* is a well-known performance measure for the scheduler: it gives the worst-case ratio, among all possible choices for the environment, of the cumulated utility of the online scheduler versus an offline scheduler that knows these choices in advance. Traditionally, competitive analysis is performed by hand, while automated techniques are rare and only handle *static* environments with *independent* tasks. We present a quantitative-verification framework for *precedence-aware* competitive analysis, where task releases may depend on preceding scheduling choices, i.e., the environment can respond to scheduling decisions *dynamically*. We consider two general classes of precedences: 1) *follower* precedences force the release of a dependent task upon the completion of a set of precursor tasks, while and 2) *pairing* precedences modify the characteristics of a dependent task provided the completion of a set of precursor tasks. Precedences make competitive analysis challenging, as the online and offline schedulers operate on diverging sequences. We make a formal presentation of our framework, and use a GPU-based implementation to analyze ten well-known schedulers on precedence-based application examples taken from the existing literature: 1) a handshake protocol (HP); 2) network packet-switching; 3) query scheduling (QS); and 4) a sporadic-interrupt setting. Our experimental results show that precedences and task parameters can vary drastically the best scheduler. Our framework thus supports application designers in choosing the best scheduler among a given set automatically.**

***Index Terms*—Competitive analysis, precedence scheduling, quantitative verification.**

I. Introduction

In THIS work, we consider competitive analysis of real-time scheduling of a finite set of firm-deadline tasks on a single processor, in a discrete-time model. Each task is primarily characterized by its worst-case execution time (WCET), a *relative deadline*, and a *utility value* here. The environment releases a possibly infinite sequence of task instances, called *jobs*, and in each time slot, a *scheduler* selects one time unit of one of the released jobs that are not yet completed. If a job is completed before the deadline, its utility is attributed to the system; a job that does not meet its firm deadline does not harm, but does not add any utility. Firm-deadline tasks arise in various application domains, e.g., machine scheduling [1], multimedia and video streaming [2], QoS management in bounded-delay data network switches [3] or networks-on-chip [4], and disk scheduling [5], and scheduling algorithms like D^{over} [6] have been designed that work very well even in the presence of severe overload.

Competitive analysis [7] compares the performance of an *online* algorithm $\mathcal{A}$, which processes a sequence of inputs online (without knowing the future), with what can be achieved by an optimal *offline* algorithm $\mathcal{C}$ (that does know the future). In particular, for a given taskset, the *competitive ratio* is the worst-case cumulative utility ratio of $\mathcal{A}$ versus $\mathcal{C}$ over all admissible job release sequences that can be generated for that taskset. Hence, competitive analysis is an important component in the design of real-time applications: maximizing the competitive ratio means choosing the scheduler with the best worst-case performance guarantees for the given application. On the other hand, competitive analysis is particularly cumbersome, as determining the sequence that pushes the online scheduler to its worst performance is a highly nontrivial task [8].

Researchers have hence started to use formal methods to automate the process of determining the competitive ratio [9]–[13]. In high level terms, the analysis is modeled as a quantitative verification task, where the implementation is the online algorithm and the specification is the target value of the competitive ratio. For a given job sequence, one computes the ratio of the cumulative utility obtained by the online algorithm and the offline algorithm in that job sequence. The analysis determines whether the implementation satisfies the specification for all job sequences, or whether there exists a job sequence for which the specification is violated.

This simple approach breaks in the presence of *precedence relations*, which capture dependencies between completed and released jobs. Consider, for example, the setting of scheduling requests for blocks of a disk issued by different client processes in an operating system [5]. The requests issued by a client that reads a file sequentially should be executed in

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported by the Austrian Science Foundation (FWF) under the NFN RiSE/SHiNE under Grant S11405 and Grant S11407. This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Andreas Pavlogiannis.)*

Andreas Pavlogiannis is with the Department of Computer Science, Aarhus University, 8000 Aarhus, Denmark (e-mail: pavlogiannis@cs.au.dk).

Nico Schaumberger and Ulrich Schmid are with the Institute of Computer Engineering, TU Wien, 1040 Vienna, Austria (e-mail: nico.schaumberger@gmail.com; s@ecs.tuwien.ac.at).

Krishnendu Chatterjee is with the Institute of Science and Technology Austria, Klosterneuburg, 3400 Austria (e-mail: krishnendu.chatterjee@ist.ac.at).

Digital Object Identifier 10.1109/TCAD.2020.3012803

first-in first-out (FIFO) order.[1] This can be accomplished either by requesting clients to issue the next request only after the previous one has been completed, in a stop-and-wait fashion, or else by a precedence-aware disk scheduler that does not reorder the requests. In general, precedences imply that scheduling decisions made in the present affect tasks released in the future, i.e., the environment responds *dynamically* to the scheduler. Consequently, when computing the competitive ratio, the online and offline algorithms no longer work on the *same* job sequence, but rather on job sequences that *diverge*. Unfortunately, existing automated competitive analysis techniques can only handle *static* environments, where such precedence dependencies are not expressible. We address this challenge in this work.

Main Contributions: In this article, we develop a quantitative-verification framework for the competitive analysis of real-time schedulers for firm deadline tasks that may contain nonpreemptible sections and precedences, i.e., future task releases that depend on preceding scheduling choices. Starting out from the framework [13], we developed an approach that can handle two general forms of precedences.

1) *Follower Precedences:* Condition the release of a specific task on the completion time of one or more precursor tasks; they can be used to express sequential processing requirements. For example, if some interrupt occurs, a special clean-up job shall be released within 5 to 10 time units after completing the execution of the interrupt handler job.
2) *Pairing Precedences:* Dynamically select the actual task to be executed upon the release of a specific task, depending on the relation of its release time to the completion of an earlier task. A simple example is a periodic server task, which services asynchronous client requests. The execution time requirement of a job of the server task depends on whether a client job has been completed some time before its release or not.

Handling such precedences is challenging, both with respect to the theoretical formalization and the practical performance. As we have mentioned above, precedences cause online and offline algorithms to work on *diverging* job sequences when computing the competitive ratio. The formalization challenge arises in allowing divergence to occur only due to precedences. On the practical side, previous works exploit the fact that earliest deadline first (EDF) is optimal in nonoverloaded environments. This allows an efficient encoding of the offline scheduler, which limits the nondeterminism that has to be explored by the verification algorithm. In the presence of precedences, EDF is no longer optimal and full-blown nondeterminism has to be explored. We respond to this challenge by developing a parallel verification algorithm implemented in CUDA and executed on a GPU. Experiments with tasksets shaped along several application examples demonstrate that our framework computes in a few seconds highly nontrivial competitive ratios.

[1]Our example focuses on the application-level interface to the disk subsystem, where blocks are delivered sequentially. The actual disk accesses issued at lower layers, according to a caching policy, may of course be nonsequential.

Related Work: Sheikh *et al.* [14] considered the problem of nonpreemptively scheduling periodic *hard* real-time tasks (where all jobs must make their deadlines). Altisen *et al.* [9] used games for synthesizing controllers dedicated to meeting all deadlines in systems with shared resources. Bonifaci and Marchetti-Spaccamela [10] employed graph games for automatic feasibility analysis of sporadic real-time tasks in multiprocessor systems. All these approaches do not generalize to competitive analysis of tasks with firm deadlines, however, which were addressed in [11]–[13].

Real-time-systems research on firm-deadline task scheduling essentially started out from [8], and has been generalized in several directions: Energy consumption [15], [16], imprecise computation tasks [17], lower bounds on slack time [18], and fairness [19]. Maximizing cumulated utility while satisfying multiple resource constraints is also the purpose of the QoS-based resource allocation model (Q-RAM) [20] approach.

Precedences have been a natural way to capture task dependencies in a variety of applications, such as: 1) query scheduling (QS) [21] and operator-scheduling [22] for stream processing; 2) scheduling disk requests [5], interrupt-handling in multitasking operating systems [23]; and 3) network-on-chip switch scheduling [24].

Paper Organization: In Section II, we describe our model of firm-deadline scheduling with precedences. Section III describes the reduction to graphs employed in our framework, Section IV explains how our competitive analysis works in detail. In Section V, we present a a parallel implementation of Madani's minimum average cycle algorithm [25], and present our experimental results.

II. Model

A. System Model

Firm-deadline scheduling deals with instances (called *jobs*) of a set of *tasks* $Ts = \{\tau_1, \ldots, \tau_N\}$, primarily characterized by their WCET Et_i, *relative deadline* Dl_i, and *utility value* Ut_i, which are to be scheduled on a single processor. A job from τ_i that is released by the environment at time t and completed by $t+Dl_i$ contributes Ut_i to the cumulative utility of the system; a job that does not meet its deadline does not harm, but does not add any utility. The goal of a scheduler is to maximize the cumulated utility. We allow the environment to respond dynamically to scheduling choices. In particular, the completion of a set of tasks can:

1) force the release of another task or
2) alter the characteristics of a future task, according to the precedences specified in Section II-C.

Although the execution time of two jobs of the same task can be different, depending on their context, we consider it constant to avoid complicating our notation further. More specifically, like most real-time scheduling research, we assume that tasks have a WCET and that every job takes exactly this WCET to complete. Nonwithstanding some scheduling anomalies [26], this choice typically yields the worst-case behavior of a real-time scheduler. Since the competitive ratio also studies worst-case behaviors, using the WCET is also a sound choice for any scheduler that can only

TABLE I
GLOSSARY OF OUR NOTATION

$\mathrm{Ts} = \mathrm{Ts}_b \cup \mathrm{Ts}_f \cup \mathrm{Ts}_p$	The full taskset, consisting of the baseline, the follower, and the paired tasksets, respectively
Et_τ, Dl_τ,Ut_τ	WCET, relative deadline and utility of task τ
$J_{i,j}$	The job of task τ_i at time j
$\Sigma = 2^{\mathrm{Ts}}$	The set of sets of tasks releases that may occur in a single slot
$\Pi = ((\mathrm{Ts} \times \{0, \ldots, \mathrm{Dl}_{\max} - 1\}) \cup \{\perp\})$	The set of jobs that may be scheduled in a single slot (uses relative indexing)
$\sigma = (\sigma^\ell)_{\ell \geq 1} \in \Sigma^\infty$	A (infinite) job release sequence
$\pi = (\pi^\ell)_{\ell \geq 1} \in \Pi^\infty$	A (infinite) schedule
$\#\mathrm{CompJobs}_i(\pi)$	The number of jobs of task τ_i that are completed by the finite schedule π
$\mathrm{Ut}(\pi) = \sum_{i=1}^{N} \#\mathrm{CompJobs}_i(\pi) \cdot \mathrm{Ut}_i$	The utility achieved by the finite schedule π
$\mathcal{A}(\sigma) = (\tau_i, j)$	The scheduling choice of scheduler $\mathcal{A}$ for the current slot, given the finite job sequence σ
$\mathcal{A}[\sigma] = (\mathcal{A}(\sigma(\ell)))_{\ell \geq 1} \in \Pi^\infty$	The infinite schedule of scheduler $\mathcal{A}$ given the infinite job sequence σ

benefit when a job does not meet its WCET. Our framework could very easily be extended to explicitly allow for variable execution times of the tasks, however. To avoid extra unnecessary notation, we simply point out here that variable execution times can be modeled by defining multiple variants of the same task with different WCETs.

The schedulers we consider have the following characteristics.

1) *Nonrandomized:* Although randomization might help for average performance, it offers no benefit to the competitive ratio, which characterizes worst-case performance.
2) *Deterministic:* As we analyze practical schedulers, nondeterminism is not allowed in the real-time setting.
3) *Bounded-Space:* We require that the memory usage of the scheduler is bounded, i.e., it does not grow arbitrarily over time. This is only a weak limitation, as all real-time schedulers we are aware of satisfy this property.
4) *No Overhead:* We assume that scheduling choices and context switching are instantaneous and incur no overhead. Incorporating those algorithm-specific features is left for future work.

B. Basic Definitions

In this section, we formally introduce our scheduling model and set up relevant notation. To facilitate comparability of our approaches and results, our exposition closely follows [12]. For easy reference, we summarize most of our notation in Table I.

Notation on Sequences: Let X be a finite set. For a finite sequence $x = (x^\ell)_{\ell \geq 1} = (x^1, x^2, \ldots, x^k) \in X^*$ of elements in X, we let $|x| = k$. We denote by x^ℓ the element in the ℓth position of x, and by $x(\ell) = (x^1, x^2, \ldots, x^\ell)$ the finite prefix of x up to position ℓ. Given a function $f : X \to \mathbb{Z}$ (where $\mathbb{Z}$ is the set of integers) and a finite sequence $x \in X^*$, we denote by $f(x) = \sum_{\ell=1}^{|x|} f(x^\ell)$.

Real-Time Scheduling Setting: We consider a finite set of tasks $Ts = \{\tau_1, \ldots, \tau_N\}$, to be executed on a single processor. We assume a discrete notion of real time $t = k\varepsilon$, $k \geq 1$, where $\varepsilon > 0$ is both the unit time and the smallest unit of preemption (called a *slot*). Since both task releases and scheduling activities occur at slot boundaries only, all timing values are specified as positive integers. Every task τ_i releases at most countably infinitely many task instances (called *jobs*) $J_{i,j} := (\tau_i, j) \in Ts \times \mathbb{N}^+$ (where $\mathbb{N}^+$ is the set of positive integers) over time (i.e., $J_{i,j}$ denotes that a job of task τ_i is released at time j). Every task τ_i, is characterized by a four-tuple $\tau_i = (Et_i, Dl_i, Ut_i, Np_i)$, as illustrated in Fig. 1.

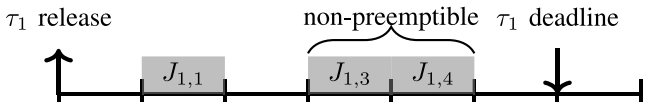

Fig. 1. Illustration of a task $\tau_1 = (3, 6, 3, [2, 3])$. The task is released at time 0, and thus has absolute deadline at time $0 + 6 = 6$. It is scheduled in slots 1, 3, and 4, with the notation $J_{i,j}$ denoting the scheduling of the job of task τ_i that was released j time slots ago. We have a single preemption at the end of slot 2, while the second and third units of the task are nonpreemptible. The task is completed at the end of slot 5, at which point it confers utility 3 to the system (not shown).

1) $Et_i \in \mathbb{N}^+$ is the WCET of τ_i in number of slots.
2) $Dl_i \in \mathbb{N}$ is the *relative deadline* of τ_i in number of slots.
3) $Ut_i \in \mathbb{N}$ is the *utility value* of τ_i (rational utility values can be mapped to integers by proper scaling).
4) $Np_i = \{[l_1, l_2], \ldots, [l_{2k-1}, l_{2k}]\}$, where $l_{2k} \leq Et_i$ and $l_{\ell-1} < l_\ell$ for all ℓ, are the *nonpreemptible* sections of τ_i. If the l_ℓth unit of job a $J_{i,j}$ is executed in some slot t, then the same job must be executed throughout the interval $[t, t + l_\ell - l_{\ell-1}]$.

We denote by $Dl_{\max} = \max_{1 \leq i \leq N} Dl_i$ the maximum relative deadline in Ts, and by $Ut_{\max} = \max_{1 \leq i \leq N} Ut_i$ the maximum utility in Ts. Every job $J_{i,j}$ needs the processor for Et_i slots exclusively to execute to completion. These slots are not necessarily consecutive, unless specified so by Np_i. All tasks have firm deadlines: only a job $J_{i,j}$ that completes by time $j + Dl_i$ provides utility Ut_i to the system. A job that misses its deadline does not harm but provides zero utility. $Dl_i = 0$ is used for a task that is never scheduled.

The goal of a real-time scheduling algorithm is to maximize the *cumulated utility*, which is the sum of Ut_i times the number of jobs $J_{i,j}$ that can be completed by their deadlines, in a sequence of job releases generated by an *adversary*.

Job Sequences and Schedules: When generating a job sequence, the adversary releases at most one new job from every task in every slot. Formally, the adversary generates an infinite *job sequence* $\sigma = (\sigma^\ell)_{\ell \geq 1} \in \Sigma^\infty$, where $\Sigma = 2^{Ts}$. If a task τ_i belongs to σ^ℓ, for some $\ell \in \mathbb{N}^+$, then a (single) new job $J_{i,j}$ of task i is released at the beginning of slot ℓ: $j = \ell$ denotes the *release time* of $J_{i,j}$, which is the earliest time that $J_{i,j}$ can be executed, and $d_{i,j} = j + Dl_i$ denotes its *absolute deadline*. We also consider the set of finite job sequences Σ^*, which consists of finite prefixes of infinite job sequences.

An infinite *schedule* is a sequence $\pi = (\pi^\ell)_{\ell \geq 1} \in \Pi^\infty$, where $\Pi = ((Ts \times \{0, \ldots, Dl_{\max} - 1\}) \cup \{\perp\})$. Intuitively, $\pi^\ell = (\tau_i, j)$ signifies that the job $J_{i,\ell-j}$ of task τ_i released j slots ago is executed in slot ℓ, while $\pi^\ell = \perp$ signifies that no job is executed in slot ℓ. We also consider the set of finite schedules Π^*, which contains finite prefixes of infinite schedules. Given a finite schedule π, we define $\#\text{CompJobs}_i(\pi)$ to be the number of jobs of task Ts_i that are completed by their deadlines in π. The *cumulated utility* $Ut(\pi)$ (also called *utility* for brevity) achieved in π is defined as $Ut(\pi) = \sum_{i=1}^{N} \#\text{CompJobs}_i(\pi) \cdot Ut_i$.

Schedulers and Cumulated Utility: A *scheduler* is a function $\mathcal{A}\colon \Sigma^* \to \Pi$, i.e., it maps the sequence of job releases σ up to the current slot to the job that will be executed in the current slot (or no job at all). Let $\ell = |\sigma|$. If $\mathcal{A}(\sigma) = (\tau_i, j)$, the following constraints must be satisfied.

1) $\tau_i \in \sigma^{\ell-j}$ (the job has been released).
2) $j < Dl_i$ (the job's deadline has not passed).
3) $|\{k : k > 0 \text{ and } \pi^{\ell-k} = (\tau_i, j') \text{ and } k + j' = j\}| < Et_i$ (the job has not been completed).

A finite job sequence $\sigma \in \Sigma^*$ yields a *schedule* $\mathcal{A}[\sigma] = (\mathcal{A}(\sigma(\ell)))_{\ell \geq 1} \in \Pi^*$ for $\mathcal{A}$, i.e., we have $\mathcal{A}^\ell[\sigma] = \mathcal{A}(\sigma(\ell))$ for all ℓ. The *cumulated utility* of scheduler $\mathcal{A}$ on σ is $Ut_{\mathcal{A}}(\sigma) = Ut(\mathcal{A}[\sigma])$. An infinite job sequence $\sigma \in \Sigma^\infty$ yields an infinite schedule $\mathcal{A}[\sigma] = (\mathcal{A}(\sigma(\ell)))_{\ell \geq 1} \in \Pi^\infty$.

Our goal is to capture realistic scenarios in which the set of task instances released in each slot is not completely arbitrary, but adheres to certain precedences that capture the interaction of the scheduler with the environment until the current time.

C. Scheduling With Precedences

We allow for two types of generic precedences to be expressible in our framework: 1) follower and 2) pairing. To express such precedences, we partition the taskset as $Ts = Ts_b \cup Ts_f \cup Ts_p$.

1) Ts_b is a set of *baseline* tasks.
2) Ts_f is a set of *follower* tasks, arising in follower precedences.
3) Ts_p is a set of *paired* tasks, arising in pairing precedences. We consider an injective *ground function* $f\colon Ts_p \to Ts_b \cup Ts_f$ that grounds every paired task to a baseline or follower task.

We now proceed with an operational description of the above precedences. Each such precedence *fires* whenever the schedule meets certain criteria. When a precedence fires, certain restrictions are imposed on the adversary regarding future task releases. Each such precedence holds within a variable horizon, until it is *met*; at that point it stops being effective until the next time it fires. In particular, we have the following precedences.

1) Pairing Precedences: This type of precedences specifies that whenever a set of tasks is completed by the scheduler, then the next release of a certain task modifies the properties of that task. Formally, each *pairing precedence* is specified by a tuple $\text{Pair} = (\tau_i, [t_1, t_2], \{\tau_{l_1}, \ldots, \tau_{l_k}\})$, where: 1) $\tau_i \in Ts_p$; 2) $\tau_{l_\ell} \in Ts$; and 3) $1 \leq t_1 \leq t_2$ and $t_2 \leq \infty$. The precedence fires in the current slot t' if the following conditions are met.

1) An instance J_{l_ℓ, j_ℓ} of some task τ_{l_ℓ} is completed in the current slot.
2) An instance $J_{l_{\ell'}, j_{\ell'}}$ of all tasks $\tau_{l_{\ell'}}$, with $\ell' \neq \ell$ has been completed since the last time Pair was met.

When the precedence fires, a future release of an instance of $f(\tau_i)$ is *paired* with $\tau_{l_1}, \ldots, \tau_{l_k}$, effectively lifting it to a release of τ_i instead. That is, provided the precedence fires in slot t', if an instance of task $f(\tau_i) \in Ts_b \cup Ts_f$ is released in the interval $[t' + t_1, t' + t_2]$, it is lifted to an instance of $\tau_i \in Ts_p$. The precedence is met the first time τ_i is released in the interval $[t' + t_1, t' + t_2]$, or at time $t' + t_2$ if no release of $f(\tau_i)$ occurs in that interval. A task $\tau_i \in Ts_p$ can only be released if a pairing precedence $\text{Pair} = (\tau_i, [t_1, t_2], \{\tau_{l_1}, \ldots, \tau_{l_k}\})$ has fired but not already met.

2) Follower Precedences: This type of precedences specifies that whenever a set of tasks is completed by the scheduler, then another task has to be released in the system. Formally, each *follower precedence* is specified by a tuple $\text{Follow} = (\tau_i, [t_1, t_2], \{\tau_{l_1}, \ldots, \tau_{l_k}\})$, where 1) $\tau_i \in Ts_f$; 2) $\tau_{l_\ell} \in Ts$; and 3) $-\infty \leq t_1 \leq t_2$ and $t_2 < \infty$. The criteria for whether the precedence fires in the current slot t' are the same as in the case of pairing precedences. The precedence specifies that if it fires in slot t', an instance of task τ_i has to be released in the interval $[t' + t_1, t' + t_2]$. The precedence is met the first time τ_i is released in that interval. A task $\tau_i \in Ts_f$ can only be released if a follower precedence $\text{Follow} = (\tau_i, [t_1.t_2], \{\tau_{l_1}, \ldots, \tau_{l_k}\})$ has fired but not already met.

Note that $t_1 \leq 0$ allows to model situations where the follower task is released *before* the constraint fires. Together with a scheduler like EDF* which does not schedule already released follower jobs before the preceding ones have been completed, this allows to model classic precedence constraints [27] like in sequential disk request processing [5].

Combinations of Precedences: For each pairing and follower precedence, the tasks τ_{l_i} are called *precursor tasks*, and τ_i is called the *dependent task*. Precedences can be combined in arbitrary ways. For example, a *fork* in the precedence relation can be expressed by two precedences with a common set of precursor tasks, while a *join* in the precedence relation can be expressed by two precedences with a common dependent task.

D. Examples of Precedence-Aware Scheduling

Using the precedences introduced above, one can model a wide range of the task dependencies that are typically found in real applications. Here, we present some examples, which will also guide our experiments in Section V.

Serving Sporadic Interrupts: Many applications contain a "consumer" task, which is periodically executed, and a sporadically executed "producer" task. By sporadic, we mean nonperiodic tasks with firm deadlines with some minimal release separation time. If an instance of the producer task has been completed before the release of the consumer task, the latter has to do some extra service work. Examples are a post-processing thread for events signaled by interrupt-service routines in embedded systems [23] or efficient disk

scheduling [5]. In our framework, this can be represented by a producer task $\tau_P \in Ts_b$ with small Et_P, Dl_P, and large Ut_P, and a consumer task that is split into the unpaired version $\tau_C \in Ts_b$ with moderate Et_C, Dl_C, Ut_C (representing the situation where no extra work needs to be done) and a paired version $\tau'_C \in Ts_p$ with large Et'_C, Dl'_C, Ut'_C. The pairing precedence is $(\tau'_C, [1, \infty], \{\tau_P\})$.

Handshake Protocols (HPs): A natural setting for a follower precedence is the well-known stop-and-wait acknowledgment protocol for reliable data communication between two hosts that share a mutually exclusive communication channel [28]. If host A wants to send a sequence of messages to host B, it sends a message and waits until an acknowledgment message has been received from B, before it sends the next message. We model this using a task $\tau_m \in Ts_b$ that represents the message sent from A to B, and a follower task $\tau_f \in Ts_f$ that represents the acknowledgment message returned by B. They are connected by the follower precedence $(\tau_f, [\delta_{\min}, \delta_{\max}], \{\tau_m\})$, where $[\delta_{\min}, \delta_{\max}]$ is the range of message transmission delays. The deadlines Dl_m and Dl_f are chosen in accordance with the desired timeout for retransmission.

Packet Switching (PS): Network switches schedule incoming packets for some destination node, see [24] for a network-on-chip application. Packets arrive via different input links here, and are forwarded to the appropriate output link. Typically, packets are split into a header fragment and $k \geq 1$ data fragments. Since forwarding fragments that are late (e.g., due to link congestion) is of no use for many network services, this can be modeled in our framework as follows: we distinguish 1) a nonpreemptible header task $\tau_h \in Ts_b$; 2) $k-1 \geq 0$ data fragment tasks $\tau_1, \ldots, \tau_{k-1} \in Ts_b$; and 3) a last data fragment task $\tau_\ell \in Ts_b$, all with zero utility. For $\tau_1, \ldots, \tau_{k-1} \in Ts_b$ and $\tau_\ell \in Ts_b$, there are also 4) paired tasks $\tau'_1, \ldots, \tau'_{k-1} \in Ts_p$ with zero utility; and 5) a paired task $\tau'_\ell \in Ts_p$ with nonzero utility. Their dependencies are expressed by the pairing precedences $(\tau'_1, [1, t], \{\tau_h\})$, $(\tau'_i, [1, t], \{\tau'_{i-1}\})$ for $1 \leq i \leq k-1$, and $(\tau'_\ell, [1, t], \{\tau'_{k-1}\})$ (for $k > 1$), resp. $(\tau'_\ell, [1, t], \{\tau_h\})$ (for $k = 1$). These precedences model the fact that a packet can provide value only if all data fragment tasks up to the last one are paired; note that the value of t can be used to limit the maximum acceptable fragment separation time.

Query Scheduling: More complex follower precedences arise in query-scheduling, e.g., as studied in [21] for data streams. The goal is to schedule query tasks, which depend on the results of various predecessor tasks (such as reading new data, evaluating new data, and processing new requests). For example, consider a setting where the completion of either (but not both) τ_1 or τ_2 leads to the release of τ_3 within $[1, t]$ slots in the future. This can be modeled in our framework by means of three precedences: 1) $\text{Follow}_1 = (\tau_3, [1, t], \{\tau_1\})$; 2) $\text{Follow}_2 = (\tau_3, [1, t], \{\tau_2\})$; and 3) $\text{Pair} = (\tau'_3, [1, t], \{\tau_1, \tau_2\})$, where the ground function maps $f(\tau'_3) = \tau_3$; herein, τ'_3 is a special task that has $Dl_l = Ut_l = 0$, and is not precursor of any precedence. Intuitively, 1) and 2) force the adversary to release τ_3 if τ_1 *or* τ_2 has been completed by the scheduler, while 3) avoids the double release of τ_3 if *both* τ_1 and τ_2 have been completed.

E. Labeled Transition Systems

Our framework relies on labeled transition systems (LTSs), as in [12].

Labeled Transition Systems: Formally, a LTS is a tuple $L = (S, s, \Theta, \Xi, \Delta)$, where S is a finite set of states, $s \in S$ is the initial state, Θ is a finite set of input actions, Ξ is a finite set of output actions, and $\Delta \subseteq S \times \Theta \times S \times \Xi$ is the transition relation. Intuitively, $(s^1, x, s^2, y) \in \Delta$ if, given the current state s^1 and input x, the LTS outputs y, and makes a transition to state s^2. If the LTS is deterministic, then there is always a unique output and next state, i.e., Δ is a function $\Delta\colon S \times \Theta \to S \times \Xi$. Given an input sequence $\alpha \in \Theta^\infty$, a *run* of L on α is a sequence $\rho = (p_\ell, \alpha_\ell, q_\ell, \beta_\ell)_{\ell \geq 1} \in \Delta^\infty$ such that $p_1 = s$ and for all $\ell \geq 2$, we have $p_\ell = q_{\ell-1}$. For a deterministic LTS, for each input sequence, there is a unique run.

Safety Automata: A *safety automaton* is an LTS $\mathcal{S} = (S, s, \Theta, \emptyset, \Delta)$, (i.e., it does not perform any output actions), with a distinguished set $X \subseteq S$ of *rejecting* states. The automaton *accepts* an infinite sequence $\alpha \in \Theta^\infty$ if there is a corresponding run that does not contain any state from X.

Precedences as Safety Monitors: Each precedence C is specified as a *safety monitor*, defined as a safety automaton $\mathcal{S}^C$ where the set of input actions is $\Theta = \Sigma \times \Pi$. Hence, $\mathcal{S}^C$ concurrently keeps track of the tasks released at each slot, and monitors the scheduling decisions of the real-time scheduler. Recall that whenever some precedence fires, it has to be met within $[t_1, t_2]$ slots, for $t_2 = \infty$ or some some fixed $t_2 \in \mathbb{N}$ that is specific to the precedence. The safety monitor keeps track of when C is fired, and enters a rejecting state if C is fired but not met within $[t_1, t_2]$ slots. On the other hand, if C is met, the monitor immediately resets to its initial state and monitors when C is fired again.

The set of all precedences is monitored by a *global safety monitor* $\overline{\mathcal{S}}$, which runs all safety monitors, $\mathcal{S}^C$ in parallel, and enters a rejecting state iff any of $\mathcal{S}^C$ enters a rejecting state. In addition, $\overline{\mathcal{S}}$ ensures that a pairing task $\tau_i \in Ts_p$ is released only if a pairing precedence $\text{Pair} = (\tau_i, [t_1, t_2], \{\tau_{l_1}, \ldots, \tau_{l_k}\})$ has fired and has not already been met. Given a real-time scheduler $\mathcal{A}$ and a global safety monitor $\overline{\mathcal{S}}$, an infinite job sequence σ is *compatible* with $\overline{\mathcal{S}}$ and $\mathcal{A}$, written $\sigma \models \mathcal{A}, \overline{\mathcal{S}}$ if for every $k \geq 1$, the monitor accepts the input $(\sigma(k), \mathcal{A}[\sigma(k)])$.

Example of Precedences as Monitor Automata: Fig. 2 shows the monitor automaton for the pairing precedence $\text{Pair} = (\tau'_1, [1, 2], \{\tau\})$, where $\tau = (1, 2, V, \emptyset)$. The starting state is 00, and state ab with $a, b \in \{0, 1\}$ encodes released but not scheduled jobs of τ: in state 01 (resp. 10), such a job has been released in the last slot (resp. in the last but one slot), while state 11 encodes a release in both slots. States f_1 and f_2 are the two states where the precedence has fired and can thus be met by the release of τ'_1, which brings the automaton back to 00. The release of τ'_1 in any other state leads to rejection rej.

F. Competitiveness of Real-Time Schedulers

We are now ready to give a formal definition of the competitive ratio of a real-time scheduler subject to precedences. Recall that the underlying challenge is to have a definition that allows divergent job sequences for the online and offline

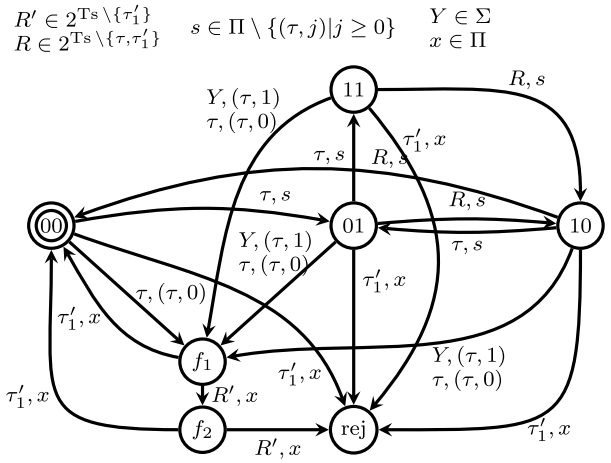

Fig. 2. Monitor automaton for the pairing precedence Pair $= (\tau_1', [1, 2], \{\tau\})$, where $\tau = (1, 2, V, \emptyset)$. The starting state is 00. Edges are labeled as Y, x, where $Y \in \Sigma$ is a set of tasks released and $x \in \Pi$ a job to be scheduled; τ, x (resp. τ_1', x) denotes a set of tasks released that contain τ (resp. τ_1'), and Y, s denotes the scheduling of job s of some task other than τ.

scheduler, provided this divergence is the result of the environment responding dynamically to scheduling decisions. More specifically, we allow a job to be seen only by the online resp. offline scheduler iff the job comes from a dependent task in a precedence that has fired by the corresponding scheduler. Moreover, if the precedence is a pairing precedence, which modifies a baseline task, then the corresponding baseline job must be made available to the other scheduler.

Below we make the above intentions formal, using: 1) the notion of compatibility of a job sequence with a schedule and 2) a notion of compatibility between the job sequences themselves, defined below.

Compatible Job Sequences: Recall the ground function f. Given a set of tasks $X \subseteq Ts$, we denote by $f(X) = \{f(\tau) \colon \tau \in X \cap Ts_p\}$. Two infinite job sequences σ_1 and σ_2 are called *compatible*, denoted by $\sigma_1 \bowtie \sigma_2$, if for every $\ell \in \mathbb{N}$, we have

$$\left(\sigma_1^{\ell} \cup f(\sigma_1^{\ell})\right) \cap Ts_b = \left(\sigma_2^{\ell} \cup f\left(\sigma_2^{\ell}\right)\right) \cap Ts_b. \tag{1}$$

Informally, the baseline tasks that are either released directly or occur through their paired tasks coincide in σ_1 and σ_2.

Competitive Ratio: Given a real-time scheduler $\mathcal{A}$ and a set of precedences expressed as a global safety monitor $\overline{\mathcal{S}}$, the *competitive ratio* of $\mathcal{A}$ on the taskset *Ts* given $\overline{\mathcal{S}}$ is defined as

$$\mathcal{CR}(\mathcal{A}) = \inf_{\substack{\mathcal{B}, \sigma_{\mathcal{A}}, \sigma_{\mathcal{B}} \colon \\ \sigma_{\mathcal{A}} \bowtie \sigma_{\mathcal{B}} \\ \sigma_{\mathcal{A}} \models \mathcal{A}, \overline{\mathcal{S}} \\ \sigma_{\mathcal{B}} \models \mathcal{B}, \overline{\mathcal{S}}}} \liminf_{k \to \infty} \frac{1 + Ut_{\mathcal{A}}(\sigma_{\mathcal{A}}(k))}{1 + Ut_{\mathcal{B}}(\sigma_{\mathcal{B}}(k))} \tag{2}$$

where $\mathcal{B}$ ranges over real-time schedulers. In words, the competitive ratio is the smallest ratio of the utilities cumulated by $\mathcal{A}$ over the utilities cumulated by any other real-time scheduler $\mathcal{B}$ on a pair of corresponding job sequences that are compatible with each other and with the precedences specified by the monitor $\overline{\mathcal{S}}$.

Note that (2) allows $\sigma_{\mathcal{A}}$ and $\sigma_{\mathcal{B}}$ to diverge, as a result of the environment responding dynamically to the divergent scheduling choices of the respective schedulers. At the same time, our definition allows *only* for such divergence to happen, e.g., if a task τ is released for $\mathcal{A}$ but not $\mathcal{B}$, then 1) τ is a dependent task in a precedence that has fired for $\mathcal{A}$ and 2) if that precedence is a pairing precedence, the corresponding baseline task must be released for $\mathcal{B}$.

Remark 1: We have $\mathcal{CR}(\mathcal{A}) \leq 1$, witnessed by taking $\mathcal{B} = \mathcal{A}$.

III. Graphs With Safety and Quantitative Objectives

In this section, we present algorithms related to safety and quantitative objectives on graphs. In the next section, we reduce the competitive analysis problem subject to precedence constraints to solving for such objectives on the appropriate graphs. Our description again follows [12] as much as possible.

Multigraphs: A *multigraph* $G = (V, E)$, hereinafter called simply a *graph*, consists of a finite set V of N *nodes*, and a finite set of M *directed multiple edges* $E \subset V \times V \times \mathbb{N}^+$. For brevity, we will refer to an edge (u, v, i) as (u, v), when i is not relevant. We consider graphs in which for all $u \in V$, we have $(u, v) \in E$ for some $v \in V$, i.e., every node has at least one outgoing edge, and let $s \in V$ be a distinguished initial node of G. A *finite path* ρ of G is a finite sequence of edges $e^1, e^2, \ldots, e^k$ such that for all $1 \leq i < k$ with $e^i = (u^i, v^i)$, we have $v^i = u^{i+1}$. Every such ρ induces a sequence of nodes $(u^i)_{i \geq 1}$, which we will also call a path, when the distinction is clear from the context, and ρ^i refers to u^i instead of e^i. Infinite paths are defined similarly, and we denote by Ω the set of all infinite paths of G that start in the distinguished node s.

Objectives: An objective Φ is a subset of Ω that defines a desired set of paths starting in the distinguished node s. Here, we consider safety, mean-payoff, and ratio objectives.

Safety Objectives: Given a set $X \subseteq V$, the *safety* objective for X is defined as $\mathrm{Safe}(X) = \{\rho \in \Omega : \forall i \geq 1, \rho^i \notin X\}$, i.e., it represents the set of all paths that never visit the set X.

Mean-Payoff Objectives: A *weight function* $W : E \to \mathbb{Z}$ assigns to each edge of G an integer weight. A weight function naturally extends to finite paths, with $W(\rho) = \sum_{i=1}^{k} W(\rho^i)$. The *mean-payoff* of an infinite path ρ is defined as

$$\mathrm{PathMP}(W, \rho) = \liminf_{k \to \infty} \frac{1}{k} \cdot W(\rho(k)) \tag{3}$$

i.e., it is the long-run average of the weights of the path. Given a weight function W and a rational threshold $\lambda \in \mathbb{Q}$, the corresponding mean-payoff objective is given as

$$MP(W, \lambda) = \{\rho \in \Omega \colon \mathrm{PathMP}(W, \rho) \leq \lambda\}. \tag{4}$$

In words, $MP(W, \lambda)$ is the set of all paths such that the mean-payoff (or limit-average) of their weights is bounded by λ. We assume without loss of generality that $\lambda \leq W_{\max}$, where $W_{\max}$ is the maximum weight assigned to an edge by W. Indeed, by definition, the objective cannot be satisfied if $\lambda > W_{\max}$.

Ratio Objectives: Given two $W_1, W_2 : E \to \mathbb{N}$, the *ratio* of a path ρ is defined as

$$\mathrm{PathRatio}(W_1, W_2, \rho) = \liminf_{k \to \infty} \frac{1 + W_1(\rho(k))}{1 + W_2(\rho(k))} \tag{5}$$

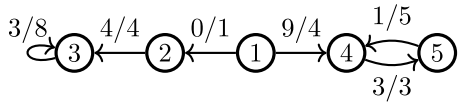

Fig. 3. Graph G with two weight functions W_1/W_2.

which denotes the limes inferior of the ratio of the sum of weights of the two functions. Given weight functions W_1, W_2 and a rational threshold $\lambda \in \mathbb{Q}$, the corresponding ratio objective is given as

$$\text{Ratio}(W_1, W_2, \lambda) = \{\rho \in \Omega : \text{PathRatio}(W_1, W_2, \rho) \leq \lambda\} \tag{6}$$

that is, the set of all paths such that the ratio of cumulative rewards with respect to W_1 and W_2 is bounded by λ.

Decision Problems: In this work, we consider the following decision problems on a multigraph $G = (V, E)$.

1) Given a set of nodes $X \subseteq V$, a weight function $W : E \to \mathbb{Z}$ and a rational threshold $\lambda \in \mathbb{Q}$, decide whether $\text{Safe}(X) \cap MP(W, \lambda) \neq \emptyset$, i.e., there exists an infinite path of G that does not visit any node in X and has mean-payoff bounded by λ.
2) Given a set of nodes $X \subseteq V$, two weight functions $W_1, W_2 : E \to \mathbb{Z}$ and a rational threshold $\lambda \in \mathbb{Q}$, decide whether $\text{Safe}(X) \cap \text{Ratio}(W_1, W_2, \lambda) \neq \emptyset$, i.e., there exists an infinite path of G that does not visit any node in X and the ratio of cumulative rewards with respect to W_1 and W_2 is bounded by λ.

If the corresponding objective is satisfied, a *witness* for the objective is an (ultimately periodic) infinite path ρ.

Example: Consider the weighted graph G in Fig. 3 with 1 the initial node. The objective $\text{Safe}(\{2\}) \cap \text{Ratio}(W_1, W_2, (2/5))$ is not satisfied, as the only periodic infinite path with ratio bounded by $(2/5)$ is $\rho_1 = 1, 2, 3, 3 \ldots$ However, this path violates the safety constraint. On the other hand, the objective $\text{Safe}(\{2\}) \cap \text{Ratio}(W_1, W_2, (3/5))$ is satisfied, as witnessed by the infinite path $\rho_2 = 1, 4, 5, 4, 5, \ldots$ with $\text{PathRatio}(W_1, W_2, \rho_2) = (1/2)$.

Solutions to the Decision Problems: We now turn our attention to the algorithms for solving each of the two decision problems. We first describe the approach to the safety with mean-payoff objective, and afterwards show how to reduce safety with ratio objectives to safety with mean-payoff objectives.

Safety With Mean-Payoff: Consider a $\text{Safe}(X) \cap MP(W, \lambda)$ objective, and let $W^{\max}$ be the maximum weight assigned by W. We solve the decision problem for the objective as follows.

1) For every $x \in X$, we remove all edges of G outgoing from x, and insert a single self-loop edge (x, x). In addition, we assign weight $W^{\max} + 1$ to (x, x).
2) We remove all nodes that are not reachable from s.
3) For every two nodes u, v, we remove all multiple edges $(u, v, i) \in E$ except the one with the smallest weight.
4) We solve the minimum-mean cycle (MMC) problem on the modified graph $G' = (V, E')$ with modified weight function W', using a standard algorithm, e.g., [25], [29]. We return True iff the MMC has mean weight at most λ.

Lemma 1: The above process returns True iff $\text{Safe}(X) \cap MP(W, \lambda)$ is satisfied in G.

Safety With Ratio: Consider a $\text{Safe}(X) \cap \text{Ratio}(W_1, W_2, \lambda)$ objective, and let $\lambda = p/q$, for an irreducible fraction p/q. We construct a new weight function $W : E \to \mathbb{Z}$, where for every edge $e \in E$, we have

$$W(e) = p \cdot W_1(e) - q \cdot W_2(e). \tag{7}$$

It is easy to verify that $\text{Safe}(X) \cap \text{Ratio}(W_1, W_2, p/q) = \text{Safe}(X) \cap MP(W, 0)$, and thus we have a reduction to the safety with mean-payoff objective.

Since the MMC on a graph with n nodes and m' edges (without multiplicities) can be solved in $O(n \cdot m')$ time [25], [29], we arrive at the following lemma.

Theorem 1: Consider a multigraph $G = (V, E)$ with n nodes and m edges. The 1) safety with mean-payoff and 2) safety with ratio objectives on G can be solved in $O(m + n \cdot m')$ time, where m' is the number of edges in G without multiplicities.

IV. Competitive Analysis

In this section we show how to compute the competitive ratio of a real-time scheduler $\mathcal{A}$ subject to follower and pairing precedences, as defined in (2).

A. Schedulers as Labeled Transition Systems

Schedulers as LTSs: For our analysis, real-time schedulers are represented as deterministic LTSs. Recall the definition of the sets $\Sigma = 2^{Ts}$ and $\Pi = ((Ts \times \{0, \ldots, Dl_{\max} - 1\}) \cup \{\perp\})$. Every real-time scheduler $\mathcal{A}$ that uses finite state space can be represented as a deterministic LTS $L_{\mathcal{A}} = (S_{\mathcal{A}}, s_{\mathcal{A}}, \Sigma, \Pi, \Delta_{\mathcal{A}})$, where the states $S_{\mathcal{A}}$ correspond to the state space of $\mathcal{A}$, and $\Delta_{\mathcal{A}}$ corresponds to the execution of $\mathcal{A}$ in one slot. Note that, due to relative indexing, for every current slot ℓ, the schedule π^{ℓ} of $\mathcal{A}$ contains elements from the set Π, and $(\tau_i, j) \in \pi^{\ell}$ uniquely determines the job $J_{i,\ell-j}$. Finally, we associate with $L_{\mathcal{A}}$ a reward function $r_{\mathcal{A}} : \Delta_{\mathcal{A}} \to \mathbb{N}$ such that $r_{\mathcal{A}}(\delta) = Ut_i$ if the transition δ completes a job of task τ_i, and $r_{\mathcal{A}}(\delta) = 0$ otherwise. Given the unique run $\rho^{\sigma} = (\delta^{\ell})_{\ell \geq 1}$ of $L_{\mathcal{A}}$ for the job sequence σ, where δ^{ℓ} denotes the transition taken at the beginning of slot ℓ, the cumulated utility in the prefix of the first k transitions in ρ^{σ} is $Ut(\rho^{\sigma}, k) = \sum_{\ell=1}^{k} r_{\mathcal{A}}(\delta^{\ell})$.

The Nondeterministic LTS: We model *all* real-time schedulers $\mathcal{B}$ as a *nondeterministic* LTS $L_{\mathcal{N}} = (S_{\mathcal{N}}, s_{\mathcal{N}}, \Sigma, \Pi, \Delta_{\mathcal{N}})$ where each state in $S_{\mathcal{N}}$ is a $N \times (Dl_{\max} - 1)$ matrix M, such that for each time slot t, the entry $M[i, j]$, $1 \leq i \leq N$, $1 \leq j \leq Dl_{\max} - 1$, denotes the remaining execution time of the job $J_{i,t-j}$ (i.e., the job of task i released j slots ago). In addition, the state encodes the *last-job index* (ι, κ) of the previously executed job, i.e., it indexes the entry of M that contains the job that was executed in the previous round. This information helps the scheduler to respect nonpreemptible sections. For matrices M, M', subset $T \in \Sigma$ of newly released tasks, and scheduled job $P = (\tau_i, j) \in \Pi$, we have $(M, T, M', P) \in \Delta_{\mathcal{N}}$ iff: 1) $M[i, j] > 0$ and 2) if the last-job index (ι, κ) is such that κ is in one of the intervals of Np_{ι}, then $(i, j) = (\iota, \kappa)$. The new matrix M' and new last-job index is obtained by

1) inserting all $\tau_i \in T$ into M;
2) decrementing the value at position $M[i, j]$;
3) shifting the contents of M one column to the right;
4) updating the last-job index to $(i, j+1)$.

That is, M' corresponds to M after inserting all released tasks in the current state, executing a pending task for one unit of time, and reducing the relative deadlines of all tasks currently in the system. The initial state $s_{\mathcal{N}}$ is represented by the zero $N \times (D_{\max} - 1)$ matrix, and $S_{\mathcal{N}}$ is the smallest $\Delta_{\mathcal{N}}$-closed set of states that contains $s_{\mathcal{N}}$ (i.e., if $M \in S_{\mathcal{N}}$ and $(M, T, M', P) \in \Delta_{\mathcal{N}}$ for some T, M', and P, we have $M' \in S_{\mathcal{N}}$). Finally, we associate with $L_{\mathcal{N}}$ a reward function $r_{\mathcal{N}} : \Delta_{\mathcal{N}} \to \mathbb{N}$ such that $r_{\mathcal{N}}(\delta) = Ut_i$ if the transition δ completes a task τ_i, and $r_{\mathcal{N}}(\delta) = 0$ otherwise.

B. Computing the Competitive Ratio

We are now ready to describe the computation of the competitive ratio of a real-time scheduler subject to precedences.

Synchronous Product: Recall the notion of job sequence compatibility $\sigma_1 \bowtie \sigma_2$ from Section II-F. It is straightforward to express compatibility as a *compatibility safety automaton* $\mathcal{S}_{\bowtie} = (S_{\bowtie}, s_{\bowtie}, \Sigma \times \Sigma, \emptyset, \Delta_{\bowtie})$, which, in each step, checks whether (1) is satisfied. Consider the following.

1) A real-time scheduler $\mathcal{A}$ represented as an LTS $L_{\mathcal{A}} = (S_{\mathcal{A}}, s_{\mathcal{A}}, \Sigma, \Pi, \Delta_{\mathcal{A}})$.
2) The nondeterministic LTS $L_{\mathcal{N}} = (S_{\mathcal{N}}, s_{\mathcal{N}}, \Sigma, \Pi, \Delta_{\mathcal{N}})$.
3) A global safety monitor $\overline{\mathcal{S}} = (S_C, s_C, \Sigma \times \Pi, \emptyset, \Delta_C)$.
4) The compatibility safety automaton $\mathcal{S}_{\bowtie} = (S_{\bowtie}, s_{\bowtie}, \Sigma \times \Sigma, \emptyset, \Delta_{\bowtie})$.

We define the *safety product automaton* of $\mathcal{A}$, $L_{\mathcal{N}}$, $\overline{\mathcal{S}}$, and $\mathcal{S}_{\bowtie}$ as the nondeterministic safety automaton $\mathcal{P} = (S_{\mathcal{P}}, s_{\mathcal{P}}, \Sigma \times \Sigma, \emptyset, \Delta_{\mathcal{P}})$. The set of states is $S_{\mathcal{P}} = S_{\mathcal{A}} \times S_{\mathcal{N}} \times S_C \times S_C \times S_{\bowtie}$, and the initial state is $s_{\mathcal{P}} = (s_{\mathcal{A}}, s_{\mathcal{N}}, s_C, s_C, s_{\bowtie})$. A state of $\mathcal{P}$ is rejecting if any of its last three components are rejecting. In words, given an input pair of job sequences $(\sigma_{\mathcal{A}}, \sigma_{\mathcal{N}}) \in \Sigma^{\infty} \times \Sigma^{\infty}$, the safety product automaton $\mathcal{P}$ runs in parallel $L_{\mathcal{A}}$ and $L_{\mathcal{N}}$, so that the corresponding LTS produces a schedule $\mathcal{A}[\sigma_{\mathcal{A}}]$ and a set of schedules $\mathcal{N}[\sigma_{\mathcal{N}}]$. In addition, $\mathcal{P}$ runs $\mathcal{S}_{\bowtie}$ on the pair $(\sigma_{\mathcal{A}}, \sigma_{\mathcal{N}})$ to ensure that the two job sequences are compatible. Finally, $\mathcal{P}$ runs two copies of $\overline{\mathcal{S}}$, on inputs $(\sigma_{\mathcal{A}}, \mathcal{A}(\sigma_{\mathcal{A}}))$ and $(\sigma_{\mathcal{N}}, \mathcal{N}(\sigma_{\mathcal{N}}))$, respectively, to ensure that the precedences are met for each job sequence.

Formally, we have $(s^1_{\mathcal{P}}, (x_1, x_2), s^2_{\mathcal{P}}) \in \Delta_{\mathcal{P}}$, where $s^i_{\mathcal{P}} = (s^i_{\mathcal{A}}, s^i_{\mathcal{N}}, s^i_C, s^i_C, s^i_{\bowtie})$ for each $i \in \{1, 2\}$, if there exist $y_1, y_2 \in \Pi$ such that the following conditions hold.

1) $(s^1_{\mathcal{A}}, x_1, s^2_{\mathcal{A}}, y_1) \in \Delta_{\mathcal{A}}$.
2) $(s^1_{\mathcal{N}}, x_2, s^2_{\mathcal{N}}, y_2) \in \Delta_{\mathcal{N}}$.
3) $(s^1_C, (x_1, y_1), s^2_C), (s^1_C, (x_2, y_2), s^2_C) \in \Delta_C$.
4) $(s^1_{\bowtie}, (x_1, x_2), s^2_{\bowtie}) \in \Delta_{\bowtie}$.

Finally, we associate with $\mathcal{P}$ a pair-reward function $r_{\mathcal{P}} = (r_{\mathcal{A}}, r_{\mathcal{N}}) : \Delta_{\mathcal{P}} \to \mathbb{Z} \times \mathbb{N}$ such that $r_{\mathcal{P}}(s^1, (x_1, x_2), s^2) = (r_1, r_2)$, where

$$r_1 = r_{\mathcal{A}}\left(s^1_{\mathcal{A}}, x_1, s^2_{\mathcal{A}}, y_1\right), \quad \text{for } y_1 \text{ s.t. } \left(s^1_{\mathcal{A}}, x_1, s^2_{\mathcal{A}}, y_1\right) \in \Delta_{\mathcal{A}}$$
$$r_2 = \max_{y_2 \colon (s^1_{\mathcal{N}}, x_2, s^2_{\mathcal{N}}, y_2) \in \Delta_{\mathcal{N}}} r_{\mathcal{N}}\left(s^1_{\mathcal{N}}, x_2, s^2_{\mathcal{N}}, y_2\right). \tag{8}$$

Remark 2: Consider an input (σ_1, σ_2) and an infinite run ρ of $\mathcal{P}$ on (σ_1, σ_2). If $\mathcal{P}$ accepts (σ_1, σ_2), for every $k \geq 1$, we have

$$r_{\mathcal{P}}(\sigma_1(k), \sigma_2(k)) = \left(Ut_{\mathcal{A}}(\sigma_1(k)), \sup_{\mathcal{B}} Ut_{\mathcal{B}}(\sigma_2(k))\right) \tag{9}$$

i.e., the first component of $r_{\mathcal{P}}$ corresponds to the cumulated utility of the real-time scheduler $\mathcal{A}$ on the job sequence $\sigma_1(k)$, whereas the second component of $r_{\mathcal{P}}$ corresponds to the maximum cumulated utility of any real-time scheduler $\mathcal{B}$.

Computing the Competitive Ratio: Given a taskset $Ts = Ts_b \cup Ts_f \cup Ts_p$, a real-time scheduler $\mathcal{A}$ represented as an LTS $L_{\mathcal{A}}$ and a set of precedences, we construct: 1) the nondeterministic LTS $L_{\mathcal{N}}$; 2) the global safety monitor $\overline{\mathcal{S}}$; and 3) the compatibility safety automaton $\mathcal{S}_{\bowtie}$. Afterwards, we construct the safety product automaton $\mathcal{P}$ with a set of rejecting states X. The automaton naturally induces a multigraph $G = (V, E)$ where $V = S_{\mathcal{P}}$ is the set of states of $\mathcal{P}$ and E corresponds to the transitions of $\mathcal{P}$. Given a threshold $\lambda \leq 1$, we decide whether $\mathcal{CR}(\mathcal{A}) \leq \lambda$ by solving for the objective $\text{Safe}(X) \cap \text{Ratio}(W_1, W_2, \lambda)$, where $r_{\mathcal{P}} = (W_1, W_2)$ is the reward function of the safety product automaton. Remark 2 and Theorem 1 lead to the following lemma.

Lemma 2: Given a rational threshold $\lambda \leq 1$, deciding whether $\mathcal{CR}(\mathcal{A}) \leq \lambda$ can be done in $O(m + n \cdot m')$ time, where 1) $n = |S_{\mathcal{P}}|$; 2) $m = |\Delta_{\mathcal{P}}|$; and 3) m' is the number of states $u, v \in S_{\mathcal{P}}$ such that $(u, (x^1, x^2), v) \in \Delta_{\mathcal{P}}$, for some $x^1, x^2 \in \Sigma$.

Exact Computation of the Competitive Ratio: Lemma 2 shows how to decide whether the competitive ratio is below a given rational threshold. Since we are dealing with integer weights, the exact competitive ratio is a rational number, and thus can be computed exactly by performing a binary search for λ on the interval $[0, 1]$ [as, by Remark 1, $\mathcal{CR}(\mathcal{A}) \leq 1$]. Observe that, since utilities are integer values, if we write λ as an irreducible fraction p/q, then $q \leq n \cdot Ut^{\max}$, where $n = |S_{\mathcal{P}}|$ and $Ut_{\max}$ is the maximum utility in the taskset. The minimum distance between two rational values with denominator at most q is at least $1/(q \cdot (q-1))$. Thus, the binary search terminates in $O(q(q-1)) = O(\log(n \cdot Ut_{\max}))$ iterations.

Theorem 2: The competitive ratio $\mathcal{CR}(\mathcal{A})$ can be computed in $O((m + n \cdot m') \cdot \log(n \cdot Ut_{\max}))$ time, where 1) $n = |S_{\mathcal{P}}|$; 2) $m = |\Delta_{\mathcal{P}}|$; and 3) m' is the number of states $u, v \in S_{\mathcal{P}}$ such that $(u, (x^1, x^2), v) \in \Delta_{\mathcal{P}}$, for some $x^1, x^2 \in \Sigma$.

The polynomial upper bound of Theorem 2 is in terms of the size of the product automaton $\mathcal{P}$. In terms of the size of a succinct description of $\mathcal{P}$, which is typically polylogarithmic in the number of states (e.g., as a circuit [30]), the corresponding complexity upper bounds become exponential.

V. Implementation and Experiments

In this section, we report on a prototype implementation and experimental evaluation of our framework. It has been implemented in Python and C, with the most performance-critical part written in CUDA 9 for parallelization on a GPU.

A. Parallel Minimum Mean Cycle

Prototype implementations of existing frameworks such as [13] are usually written in convenient programming languages, such as Python, which tend to be slow. Whereas a C implementation of performance-critical parts already allows a substantial speedup, it was not sufficient for handling the very large graphs generated already by relatively small examples with precedences in our setting.

We therefore implemented the most crucial step of our automatic competitive analysis, namely, finding a MMC in a weighted graph, in CUDA for execution on a GPU. More specifically, we parallelized Madani's sequential value iteration algorithm [25], which consists of three phases.

1) Initialization: For each node $s \in V$, the algorithm maintains:

1) a value $\text{val}(s) \in \mathbb{Z}$, which gives the cumulative weight of the minimum weight path starting from s found so far;
2) the optimal edge $ec(s) \in V$, which gives the direct successor of s on the current minimum weight path;
3) a backwards edge ("super-edge" in [25]) $se(s) = (o, k, \omega)$, where $o \in V$ is the origin of the current minimum length path, k is its length, and ω its weight (which also defines the weight of the super-edge from o to s).

Two instances (old and new) of these array data structures are needed, which are used alternatingly by the value iteration. For the first iteration, in the old array, all $\text{val}(s)$ are set to 0 and $ec(s)$ to an arbitrary outgoing edge from s.

2) First Round: The first round consists of n value iterations, where $\text{val}'(s)$ (in the new array) of each node $s \in V$ is first initialized to $\text{val}'(s) = \text{val}(s) + W(s, ec(s))$, where $W(s, t)$ is the weight of the edge (s, t). This is performed by one processor per node s in a CUDA compute kernel. Then, another CUDA compute kernel with one processor per edge is used to update

$$\text{val}'(s) = \min_{t \in \text{out}(s)} (\text{val}(s), \text{val}(t) + W(s, t)) \tag{10}$$

where $\text{out}(s)$ is the out-neighbors of node s. If $\text{val}'(s) < \text{val}(s)$, the edge choice is updated to

$$ec'(s) = \text{argmin}_{t \in \text{out}(s)} (\text{val}(s), \text{val}(t) + W(s, t)). \tag{11}$$

When all processors have completed an iteration, the new and old arrays switch places for the next one.

3) Second Round: This round is identical to the first round, except that each value iteration contains another step at the end. It maintains the super-edges $se(s)$ and detects cycles, and is performed by a CUDA compute kernel with one processor per node: Each node s just sets, in the new data array

$$se'(s) = (se(t).o, (se(t).k) + 1, se(t).\omega + W(s, t)) \tag{12}$$

where $t = ec(s)$. If a node s that just set $se(s)'$ finds that $se'(s).o = s$, it has detected a cycle. If it has a smaller mean weight than the previously best MMC, it becomes the new MMC. In the initialization phase, the MMC is set to the weight of the smallest self-loop. All self-loops can then be removed from the graph, as none of those can appear in a longer MMC.

Although we did not engineer the performance of our implementation in any way, e.g., by analyzing thread divergence and improving synchronization, it already provided a speedup of up to two orders of magnitude over a sequential implementation.

B. Online Algorithms

We implemented ten scheduling algorithms from the literature.

Earliest Deadline First (EDF): This classic algorithm schedules the job with the earliest absolute deadline [31]. Our implementation breaks ties based on the lowest task identifier.

Earliest Deadline First (EDF):* This variant of EDF explicitly handles precedences by modifying deadlines. As in [27], jobs are scheduled in the order of their the *active deadlines* using EDF. The active deadline of a released job $J_{i,\ell}$ of τ_i is set to the minimum of the nominal absolute deadline and the absolute deadline of any of its dependent jobs $J_{j,\ell}$ of τ_j. Since the on-line algorithm does not know the release times of any future $J_{j,\ell}$ when $J_{i,\ell}$ is released, our EDF* conservatively assumes that they are all released together with $J_{i,\ell}$.

First-In First-Out: This classic algorithm always schedules the job with the earliest release time. Ties are broken according to the lowest task identifier.

Static Priorities (SPs): This algorithm assigns fixed priorities to tasks, and schedules the job of the task with the highest priority [31]. In all our experiments, tasks with lower identifier have higher priority. In the case of multiple pending jobs of the same task, our implementation favors the job with the earliest release time.

Dynamic Priorities (DPs): This is a precedence-aware version of SP, where all precursor tasks for a precedence constraint have priority over tasks without this property. Jobs from tasks with elevated priority are ordered by higher base priority and, in case of jobs from the same task, by earlier release times.

Smallest Remaining Time (SRT): This algorithm schedules the job with the smallest remaining workload. Ties are broken first by the earlier absolute deadline.

Profit Density (PD): This algorithm is based on [21], and schedules the job with the highest PD, where PD is defined as the utility of the task divided by the remaining workload of the job. We have made the algorithm precedence-aware, by increasing the PD of a precursor task if a dependent task with a currently not met precedence has a higher Specifically, if Ut_d (resp. Et_d) is the value (resp. workload) of the dependent task instance and Ut_p (resp. Et_p) the value (resp. remaining workload) of the precursor task instance considered for scheduling, then the PD of this instance is $\max(Ut_p/Et_p, [Ut_p + Ut_d/Et_p + Et_d])$.

Smallest Slack Time (SST): This algorithm, also known as least-laxity first, is taken from [32]. It schedules the job with the SST, which is defined as the relative deadline minus the remaining workload. Ties are broken first by lower task identifiers and then by earlier release times.

D-Over (D^{over}): This algorithm, taken from [6], behaves like EDF in underloaded conditions, and has additional rules

TABLE II
SIX TASKSETS WITH PRECEDENCES. PRIMED TASKS ARE THE PAIRED VERSIONS OF THE CORRESPONDING UN-PRIMED TASKS

Taskset	Task	Et	Dl	Ut	Precedences
QS	τ_1	2	3	2	
	τ_2	1	1	1	
	τ_3	2	4	0	$\text{Follow}(\tau_3, [1,1], \{\tau_1\})$
	τ_4'	-	0	0	$\text{Pair}(\tau_4', [1,1], \{\tau_2, \tau_3\})$
	τ_4	1	1	10	$\text{Follow}(\tau_4, [1,1], \{\tau_1, \tau_2\})$ $\text{Follow}(\tau_4, [1,1], \{\tau_1, \tau_3\})$
SI	τ_1	1	1	1	
	τ_2	3	6	4	
	τ_2'	2	4	3	$\text{Pair}(\tau_2', [1,\infty], \{\tau_1\})$
SI'	τ_1	1	1	10	
	τ_2	1	4	11	
	τ_2'	1	2	9	$\text{Pair}(\tau_2', [1,\infty], \{\tau_1\})$
HP	τ_1	2	2	1	
	τ_2	1	1	1	
	τ_3	2	3	6	$\text{Follow}(\tau_3, [1,1], \{\tau_1\})$ $\text{Follow}(\tau_3, [1,1], \{\tau_2\})$
HP'	τ_1	1	1	10	
	τ_2	1	2	9	
	τ_3	1	4	11	$\text{Follow}(\tau_3, [1,1], \{\tau_1\})$ $\text{Follow}(\tau_3, [1,1], \{\tau_2\})$
PS	τ_1	1	2	0	
	τ_2	-	0	0	
	τ_2'	4	6	4	$\text{Pair}(\tau_2', [1,\infty], \{\tau_1\})$
	τ_3	2	3	0	non-preemptible
	τ_4	-	0	0	
	τ_4'	5	8	8	$\text{Pair}(\tau_4', [1,\infty], \{\tau_3\})$

involving the pending jobs' utility values for handling periods of overload. D^{over} is known to be optimal in terms of competitiveness for independent tasksets.

D-Star (D^*): This algorithm, which has been described in [33], is an early predecessor of D^{over}. It also behaves like EDF in underloaded conditions, and employs some utility-based rules (shaped along the competitive analysis presented in [8]) in periods of overload.

C. Tasksets

The tasksets used in our experiments have been shaped along the applications presented in Section II-C; the descriptions below will hence be kept brief. The parameters of the particular tasks and the precedences in these tasksets are listed in Table II.

Packet Switching: This models the behavior of a simple switch [24], for two different packets. Each packet consists of a nonpreemptible precursor task τ_1 (resp. τ_3) that represents the header fragment, and a dependent task τ_2' (resp. τ_4') that represents the data fragment. Header tasks have zero utility. Utility is only gained if the header fragment is scheduled before the data fragment. This is modeled using one pairing precedence for each packet, where the unpaired version τ_2 (resp. τ_4) of the dependent task has utility zero.

Handshake Protocols: This models a stop-and-wait data communication protocol on a serial bus, for two different packets. A precursor task τ_1 (resp. τ_2) models the sending of a payload message, while a follower task τ_3 represents the acknowledgment (which is the same for both τ_1 and τ_2) that is released one unit of time after completion of the respective precursor task. We use two versions HP and HP' of this taskset.

Sporadic Interrupt (SI): This models a periodic worker task that responds to an occasionally occurring event, e.g., in disk scheduling [5] or interrupt handling [23]. The interrupt is represented by a short precursor task τ_1, while the worker task is a longer dependent task in a pairing precedence: the unpaired version τ_2 represents idling, whereas the paired version τ_2' represents the processing of a preceding interrupt. We use two versions SI and SI' of this taskset.

Query Scheduling: This represents a more complex monitoring application, following [21]. The completion of one or two queries, modeled as tasks τ_1 and τ_2, triggers the release of two supervisor/monitor tasks: τ_3 is the result of a follower precedence on the completion of τ_1, whereas τ_4 is triggered by a follower precedence on the completion of both τ_1 and either τ_2 or τ_3. An additional pairing precedence makes sure that only one job of τ_4 is released if both τ_2 and τ_3 are completed.

D. Results

We have performed our experiments on the tasksets and real-time schedulers listed above, on an Intel Core i7-5820K-based computer (1.2 GHz) with 32 GB RAM, equipped with an EVGA GeForce GTX Titan SuperClocked 12 GB GPU (3072 cores, 1.1 GHz). The results on the competitive ratio are shown in Fig. 4. We discuss our findings in terms of the competitive ratio of each scheduler, as well as the running times of our analysis in each setting.

Competitive Ratio: Our findings are as follows.

1) There is no universally best algorithm, for any type of precedence: As the results for SI resp. SI' and HP resp. HP' reveal, for both follower and pairing precedences, just changing some task parameters changes the best online algorithm.
2) Precedence-aware online algorithms, such as PD and EDF* do not always outperform nonprecedence-aware ones. For example, in HP', the nonprecedence-aware algorithm SP is the best. Nevertheless, PD is the best in challenging tasksets, such as PS, where it is the only algorithm that provides a nonzero competitive ratio, and even has competitive ratio of 1 in that taskset.
3) Algorithms that rely on a "latest start time notification," such as D^{over} and D^*, are particularly harmed by the presence of nonpreemptible sections. The adversary can release jobs in such a way that the latest start time interrupt for high value jobs occurs while the algorithm is in a nonpreemptible section. This is clearly visible in the zero competitive ratio of these algorithms for the taskset PS.

Running Times: Overall, for all tasksets and all schedulers in our experiments, it took about 280 s to generate all LTSs, and about 1600 s to compute the competitive ratios. Thus, all 60 scheduling scenarios (six tasksets times ten schedulers) were handled in about 31 min. The largest graph (with 380048 nodes and 4805645 edges) was constructed for D^* on the taskset PS; most other graphs consisted of at most a few thousand nodes and edges, with the parallel runtime to find the MMC being only a few seconds.

Scalability: To get a better understanding of the complexity of our approach and the effect of our parallelization of

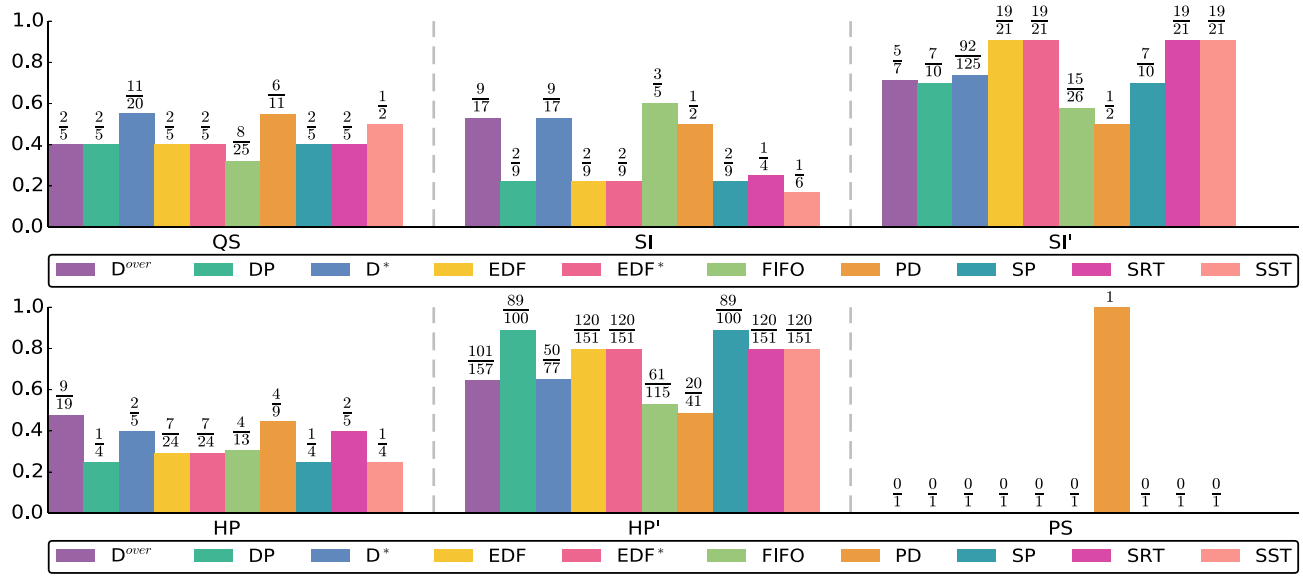

Fig. 4. Competitive ratio of each scheduler on each studied taskset. Fractions show the exact value.

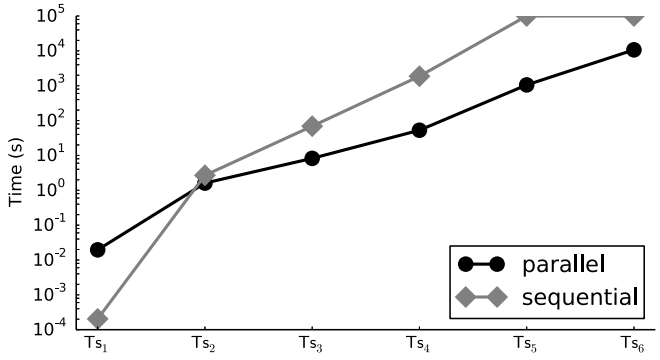

Fig. 5. Scalability plot on the tasksets of Table III.

TABLE III
SIX TASKSETS USED IN OUR SCALABILITY EXPERIMENTS. EACH OF THE SIX TASKSETS $\{Ts_i\}_i$ CONSISTS OF UP TO SIX TASKS $\{\tau_i\}_i$. FOR EACH $i \in \{1, 3, 5\}$ WE HAVE A PRECEDENCE Follow$(\tau_i, [1, 1], \{\tau_{i+1}\})$. COLUMNS ARE LABELED AS τ_i, Ut_i. ENTRIES FOR A TASK τ_i CORRESPOND TO (Et_i, Dl_i). AN ENTRY $(-, -)$ DENOTES THAT THE TASK IS NOT PRESENT IN THE RESPECTIVE TASKSET

	$\tau_1, 10$	$\tau_2, 11$	$\tau_3, 9$	$\tau_4, 12$	$\tau_5, 5$	$\tau_6, 13$
Ts_1	$(1, 2)$	$(1, 2)$	$(-, -)$	$(-, -)$	$(-, -)$	$(-, -)$
Ts_2	$(1, 2)$	$(1, 2)$	$(1, 3)$	$(2, 4)$	$(-, -)$	$(-, -)$
Ts_3	$(1, 2)$	$(1, 2)$	$(2, 5)$	$(2, 4)$	$(-, -)$	$(-, -)$
Ts_4	$(1, 2)$	$(2, 3)$	$(2, 5)$	$(3, 6)$	$(-, -)$	$(-, -)$
Ts_5	$(1, 3)$	$(2, 4)$	$(2, 5)$	$(3, 7)$	$(-, -)$	$(-, -)$
Ts_6	$(1, 2)$	$(1, 2)$	$(2, 5)$	$(2, 4)$	$(2, 4)$	$(3, 7)$

Madani's algorithm, we have performed some simple scalability experiments, using the tasksets of Table III. Fig. 5 shows the maximum time taken for computing the competitive ratio on each taskset. Timeout was set to 10^5 s. As stated in Section IV-B, although the running time is polynomial in the state-space of the corresponding LTSs, it is typically exponential in taskset parameters, such as number of tasks and maximum deadline. This exponential dependency is clearly visible in Fig. 5. We also observe that the speedup increases for more demanding tasksets, since the parallel implementation of Madani's algorithm can better utilize the specific architecture of the GPU and the CUDA compute kernels. Naturally, this speedup is bounded by the machine cores.

Scalability Optimizations: Here, we discuss here some optimizations that can lead to further scalability. Besides increasing parallelization, we identify four different ways for doing so below. The first two are related to modeling choices, while the latter two deal with algorithmic optimizations.

1) *Limiting the Preemption Points:* The state space of both LTSs $L_{\mathcal{A}}$ and $L_{\mathcal{B}}$ decreases considerably with fewer preemption points in the tasks. Hence, scalability can be improved by incorporating longer no-preemption sections.
2) *Sporadicity of Task Releases:* In our framework, an instance of a task may be released in every slot. In actual applications, task releases are expected to be much more sporadic, which reduces the scheduling choices in every step. Sporadicity restrictions can be easily encoded in our framework using the global safety monitor $\overline{\mathcal{S}}$, which will result in significant size reduction for both the online LTS $L_{\mathcal{A}}$ and offline LTS $L_{\mathcal{B}}$.
3) *Short Witnesses:* In our experiments, although the product graphs have hundreds of thousands of nodes, the length of the cycle that witnesses the competitive ratio is always in the order of tens of nodes. This suggests parameterizing the underlying MMC problem by the length of the witness cycle, and developing new algorithms that look for small witnesses more efficiently.
4) *Limiting Nondeterminism:* Recall the nondeterminism LTS $L_{\mathcal{B}}$ that models all possible schedulers. We can aggressively reduce its size by identifying scheduling choices that are always suboptimal. For example, consider the case where a job $J_{i,j}$ has been released, and the corresponding task τ_i is not a precursor task in any precedence. Then the scheduler does not benefit from

staying idle in any slot, as long as $J_{i,j}$ is not completed (or its deadline passes). Hence, the corresponding states of $L_{\mathcal{B}}$ can be pruned away, thereby reducing the size of the graph over which the competitive ratio is computed.

Optimizations in 1) and 2) are particularly useful when small differences between the competitiveness of various schedulers are not that important. For example, the practitioner can start with a coarse-grained modeling of the scheduling setting, where preemption points are few and tasks arrive very sporadically. Running our framework on this model will create a first approximation of the behavior of each scheduler, and might allow for disregarding schedulers that perform poorly, leaving a smaller pool of schedulers to choose from. In a refinement step, preemption points and sporadicity restrictions are refined, to further distinguish between the remaining schedulers, and so on. The process stops when the granularity of the model suffices to make a confident choice of the scheduler.

VI. Conclusion

In this work, we provided the foundations for the automated competitive analysis of real-time scheduling algorithms for firm-deadline tasks with precedence constraints. We defined the competitive ratio in this setting, despite the fact that the online and offline algorithms work on diverging job sequences, and showed how the various components of our framework can be implemented using LTSs. We developed a parallel version of Madani's MMC finding algorithm using CUDA on a GPU. Finally, we performed an experimental evaluation of ten scheduling algorithms on tasksets capturing various precedence-aware application examples. Our findings confirm the importance of a framework for *automated* competitive analysis for the design of a particular application, since the choice of the best algorithm for a given taskset and its precedences cannot be done manually.

In future work, we will explore extensions of our framework to also compute additional algorithm-specific performance measures, such as the number of context switches, scheduling overheads, or energy consumption, which could then be traded against the competitiveness ratio of a scheduler.

References

[1] B. D. Gupta and M. A. Palis, "Online real-time preemptive scheduling of jobs with deadlines on multiple machines," *J. Sched.*, vol. 4, no. 6, pp. 297–312, 2001.

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. 19th IEEE Real Time Syst. Symp. (RTSS'98)*, Madrid, Spain, 1998, pp. 4–13.

[3] M. Englert and M. Westermann, "Considering suppressed packets improves buffer management in QoS switches," in *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, New Orleans, LA, USA, Jan. 2007, pp. 209–218.

[4] Z. Lu and A. Jantsch, "Admitting and ejecting flits in wormhole-switched networks on chip," *IET Comput. Digit. Techn.*, vol. 1, no. 5, pp. 546–556, Sep. 2007.

[5] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with Fahrrad," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 13–25, Apr. 2008.

[6] G. Koren and D. Shasha, "D^{over}: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems," *SIAM J. Comp.*, vol. 24, no. 2, pp. 318–339, 1995.

[7] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[8] S. Baruah *et al.*, "On the competitiveness of on-line real-time task scheduling," *Real Time Syst.*, vol. 4, pp. 125–144, Jun. 1992.

[9] K. Altisen, G. Gößler, and J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," *Real Time Syst.*, vol. 23, no. 1, pp. 55–84, 2002.

[10] V. Bonifaci and A. Marchetti-Spaccamela, "Feasibility analysis of sporadic real-time multiprocessor task systems," *Algorithmica*, vol. 63, no. 4, pp. 763–780, 2012.

[11] K. Chatterjee, A. Kößler, and U. Schmid, "Automated analysis of real-time scheduling using graph games," in *Proc. 16th Int. Conf. Hybrid Syst. Comput. Control (HSCC'13)*, 2013, pp. 163–172.

[12] K. Chatterjee, A. Pavlogiannis, A. Kößler, and U. Schmid, "A framework for automated competitive analysis of on-line schedul ing of firm-deadline tasks," in *Proc. IEEE 35th Real Time Syst. Symp. (RTSS)*, Rome, Italy, Dec. 2014, pp. 118–127.

[13] K. Chatterjee, A. Pavlogiannis, A. Kößler, and U. Schmid, "Automated competitive analysis of real-time scheduling with graph games," *Real Time Syst.*, vol. 54, no. 1, pp. 166–207, Jan. 2018.

[14] A. A. Sheikh, O. Brun, P. E. Hladik, and B. J. Prabhu, "A best-response algorithm for multiprocessor periodic scheduling," in *Proc. 23rd Euromicro Conf. Real Time Syst.*, Porto, Portugal, 2011, pp. 228–237.

[15] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Trans. Comput.*, vol. 53, no. 5, pp. 584–600, May 2004.

[16] V. Devadas, F. Li, and H. Aydin, "Competitive analysis of online real-time scheduling algorithms under hard energy constraint," *Real Time Syst.*, vol. 46, pp. 88–120, Jul. 2010.

[17] S. K. Baruah and M. E. Hickey, "Competitive on-line scheduling of imprecise computations," *IEEE Trans. Comput.*, vol. 47, no. 9, pp. 1027–1032, Sep. 1998.

[18] S. K. Baruah and J. R. Haritsa, "Scheduling for overload in real-time systems," *IEEE Trans. Comput.*, vol. 46, no. 9, pp. 1034–1039, Sep. 1997.

[19] M. A. Palis, "Competitive algorithms for fine-grain real-time scheduling," in *Proc. 25th IEEE Int. Real Time Syst. Symp. (RTSS'04)*, Lisbon, Portugal, 2004, pp. 129–138.

[20] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for QoS management," in *Proc. Real Time Syst. Symp. (RTSS'97)*, San Francisco, CA, USA, 1997, pp. 298–307.

[21] Y. Zhou, J. Wu, and A. K. Leghari, "Multi-query scheduling for time-critical data stream applications," in *Proc. 25th Int. Conf. Sci. Stat. Database Manag.*, 2013, pp. 1–12.

[22] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator scheduling for memory minimization in data stream systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2003, pp. 253–264.

[23] 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, Mar. 1994.

[24] A. Burns, L. S. Indrusiak, and Z. Shi, "Schedulability analysis for real time on-chip communication with wormhole switching," *Int. J. Embed. Real Time Commun. Syst.*, vol. 1, no. 2, pp. 1–22, Apr. 2010.

[25] O. Madani, "Polynomial value iteration algorithms for deterministic MDPs," in *Proc. 18th Conf. Uncertainty Artif. Intell. (UAI'02)*, 2002, pp. 311–318.

[26] J. Reineke *et al.*, "A definition and classification of timing anomalies," in *Proc. 6th Int. Workshop Worst Case Execution Time Anal. (WCET)*, vol. 4, 2006.

[27] J. Blazewicz, "Scheduling dependent tasks with different arrival times to meet deadlines," in *Proc. Int. Workshop Org. Commision Eur. Commun. Model. Perform. Eval. Comput. Syst.*, 1977, pp. 57–65.

[28] A. S. Tanenbaum, *Computer Networks*, 4th ed. Boston, MA, USA: Prentice-Hall, 2003.

[29] R. M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Math.*, vol. 23, no. 3, pp. 309–311, 1978.

[30] H. Galperin and A. Wigderson, "Succinct representations of graphs," *Inf. Control*, vol. 56, no. 3, pp. 183–198, 1983.

[31] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[32] L. Sha *et al.*, "Real time scheduling theory: A historical perspective," *Real Time Syst.*, vol. 28, nos. 2–3, pp. 101–155, Nov. 2004.

[33] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, "On-line scheduling in the presence of overload," in *Proc. 32nd Annu. Symp. Found. Comput. Sci. (FOCS'91)*, San Juan, Puerto Rico, 1991, pp. 100–110.