A Framework for Automated Competitive Analysis of On-line Scheduling of Firm-Deadline Tasks

Krishnendu Chatterjee and Andreas Pavlogiannis
IST Austria (Institute of Science and Technology Austria)
Klosterneuburg, Austria
Email: {krish.chat, pavlogiannis}@ist.ac.at

Alexander Kößler and Ulrich Schmid
Embedded Computing Systems Group
Vienna University of Technology
Vienna, Austria
Email: {koe,s}@ecs.tuwien.ac.at

Abstract—**We present a flexible framework for the automated competitive analysis of on-line scheduling algorithms for firm-deadline real-time tasks based on multi-objective graphs: Given a taskset and an on-line scheduling algorithm specified as a labeled transition system, along with some optional safety, liveness, and/or limit-average constraints for the adversary, we automatically compute the competitive ratio of the algorithm w.r.t. a clairvoyant scheduler. We demonstrate the flexibility and power of our approach by comparing the competitive ratio of several on-line algorithms, including D^{over}, that have been proposed in the past, for various tasksets. Our experimental results reveal that none of these algorithms is universally optimal, in the sense that there are tasksets where other schedulers provide better performance. Our framework is hence a very useful design tool for selecting optimal algorithms for a given application.**

I. INTRODUCTION

We study the well-known problem of scheduling a sequence of dynamically arriving real-time task instances with firm deadlines on a single processor using a novel approach, namely, automated competitive analysis based on a corresponding multi-objective graph representation. In firm deadline scheduling, a task instance (a *job*) that is completed by its deadline contributes a positive utility value; a job that does not meet its deadline does not harm, but does not add any utility. The goal of the scheduling algorithm is to maximize the cumulated utility. Firm deadline tasks arise in various application domains, e.g., machine scheduling, multimedia and video streaming, QoS management in switches and data networks, and other systems that may suffer from overload [1].

Competitive analysis [2] has been the primary tool for studying the performance of such scheduling algorithms [3]. In general, it allows to compare the performance of an *on-line* algorithm $\mathcal{A}$, which processes a sequence of inputs without knowing the future, with what can be achieved by an optimal *off-line* algorithm $\mathcal{C}$ that does know the future (a *clairvoyant* algorithm): The *competitive factor* gives the worst-case performance ratio of $\mathcal{A}$ vs. $\mathcal{C}$ over all possible scenarios.

In a seminal paper [3], Baruah et al. proved that no on-line scheduling algorithm for single processors can achieve a competitive factor better than $1/4$ over a clairvoyant algorithm in *all* possible job sequences of *all* possible tasksets. The proof is based on constructing a specific job sequence,

which takes into account the on-line algorithm's actions and thereby forces any such algorithm to deliver a sub-optimal cumulated utility. For the special case of zero-laxity tasksets of uniform value-density, where utilities equal execution times, they also provided the on-line algorithm TD1 with competitive factor $1/4$, concluding that $1/4$ is a tight bound for this family of tasksets. In [3], the $1/4$ upper bound was also generalized, by showing that there exist tasksets with *importance ratio* k, defined as the ratio of the maximum over the minimum value-density in the taskset, in which no on-line scheduler can have competitive factor larger than $\frac{1}{(1+\sqrt{k})^2}$. In a subsequent work [1], the on-line scheduler D^{over} was introduced, which provides the performance guarantee of $\frac{1}{(1+\sqrt{k})^2}$ in any taskset with importance ratio k, showing that this bound is also tight.

Since the taskset arising in a particular application is usually known, our paper focuses on the competitive analysis problem for *given* tasksets: Rather than from *all* possible tasksets as in [3], the job sequences used for computing the *competitive ratio* are chosen from a taskset given as an input. There are two relevant problems for the automated competitive analysis for a given taskset: (1) The *synthesis* question asks to find an algorithm with optimal competitive ratio; and (2) the *analysis* question asks to compute the competitive ratio of a given on-line algorithm. In [4], we studied the synthesis problem and presented a reduction to a problem in graph games [5], which we showed to be NP-complete.

In this paper, we consider the analysis problem. More specifically, we provide a flexible, automated analysis framework that also supports additional constraints on the adversary, such as sporadicity constraints and longrun-average load. We show that the analysis problem (with additional constraints) can be reduced to a multi-objective graph problem, which can be solved in polynomial time. We also present several optimizations and an experimental evaluation of our algorithms that demonstrates the feasibility of our approach, which effectively allows to replace human ingenuity (required for finding worst-case scenarios) by computing power: Using our framework, the application designer can analyze different scheduling algorithms for the specific tasksets arising in her/his particular application, and compare their competitive ratio in order to select the best one.

Detailed contributions and paper organization:

1) In Section II, we define our scheduling problem along with the relevant additional constraints on the adversary.
2) In Section III, we introduce the labeled transition systems as a formal model for specifying on-line and off-line algorithms. In Section IV, we present the formal framework to specify the constraints on the adversary, and argue how it allows to model a wide variety of constraints. We also give an overview of all the steps involved in our approach.
3) In Section V, we present the *multi-objective graphs* used by our solution algorithm. Multiple objectives are required to represent the competitive analysis problem with various constraints.
4) In Section VI, we describe a theoretical reduction of the competitive analysis problem to solving a multi-objective graph problem, where the graph is obtained as a product of the on-line algorithm, an off-line algorithm, and the constraints specified as automata. Our algorithmic solution is polynomial in the size of the graph; however, the product graph can be large for representative tasksets.
5) In Section VII, we present both general and implementation-specific optimizations, which considerably reduce the size of the resulting graphs.
6) In Section VIII, we provide competitive ratio analysis results obtained by our method. More specifically, we present a comparative study of the performance of several existing firm deadline real-time scheduling algorithms. Our results show that, for different tasksets (even with no constraints), different algorithms achieve the highest competitive ratio (i.e., there is no universal optimal algorithm). Moreover, even for a fixed taskset and varying constraints on the adversary, different algorithms achieve the highest competitive ratio. This highlights the importance of our framework for selecting optimal algorithms for specific applications.

Related work: Algorithmic game theory [6] has been applied to classic scheduling problems since decades, primarily in economics and operations research, see e.g. [7] for just one example of some more recent work. It has also been applied for real-time scheduling of *hard* real-time tasks in the past: Besides Altisen et al. [8], who used games for synthesizing controllers dedicated to meeting all deadlines, Bonifaci and Marchetti-Spaccamela [9] employed graph games for automatic feasibility analysis of sporadic real-time tasks in multiprocessor systems: Given a set of sporadic tasks (where consecutive releases of jobs of the same task are separated at least by some sporadicity interval), the algorithms provided in [9] allow to decide, in polynomial time, whether some given scheduling algorithm will meet *all* deadlines. A partial-information game variant of their approach also allows to synthesize an optimal scheduling algorithm for a given taskset (albeit not in polynomial time). As these approaches do not generalize to competitive analysis of tasks with firm deadlines, we studied the related synthesis problem in [4].

Regarding firm deadline task scheduling in general, starting out from [3], classic real-time systems research has studied the competitive factor of both simple and extended real-time scheduling algorithms. The competitive analysis of simple algorithms (see Section VIII for the references) has been extended in various ways later on: Energy consumption [10], [11] (including dynamic voltage scaling), imprecise computation tasks (having both a mandatory and an optional part and associated utilities) [12], lower bounds on slack time [13], and fairness [14]. Note that dealing with these extensions involved considerable ingenuity and efforts w.r.t. identifying and analyzing appropriate worst case scenarios, which do not necessarily carry over even to minor variants of the problem. Maximizing cumulated utility while satisfying multiple resource constraints is also the purpose of the Q-RAM (QoS-based Resource Allocation Model) [15] approach.

II. PROBLEM DEFINITION

Real-time scheduling setting. We consider a finite set of tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, to be executed on a single processor. We assume a discrete notion of real-time $t = k\varepsilon$, $k \geqslant 1$, 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 countably many task instances (called *jobs*) $J_{i,j} := (\tau_i, j) \in \mathcal{T} \times \mathbb{N}^+$ (where $\mathbb{N}^+$ is the set of positive integers) over time (i.e., $J_{i,j}$ denotes that a job of task i is released at time j). All jobs, of all tasks, are independent of each other and can be preempted and resumed during execution without any overhead. Every task τ_i, for $1 \leqslant i \leqslant N$, is characterized by a 3-tuple $\tau_i = (C_i, D_i, V_i)$ consisting of its non-zero *worst-case execution time* $C_i \in \mathbb{N}^+$ (slots), its non-zero *relative deadline* $D_i \in \mathbb{N}^+$ (slots) and its non-zero *utility value* $V_i \in \mathbb{N}^+$ (rational utility values $V_1, \ldots, V_n$ can be mapped to integers by proper scaling). We denote with $D_{\max} = \max_{1 \leqslant i \leqslant N} D_i$ the maximum relative deadline in $\mathcal{T}$. Every job $J_{i,j}$ needs the processor for C_i (not necessarily consecutive) slots exclusively to execute to completion. All tasks have firm deadlines: only a job $J_{i,j}$ that completes within D_i slots, as measured from its release time, provides utility V_i to the system. A job that misses its deadline does not harm but provides zero utility. The goal of a real-time scheduling algorithm in this model is to maximize the *cumulated utility*, which is the sum of V_i times the number of jobs $J_{i,j}$ that can be completed by their deadlines, in a sequence of job releases generated by the *adversary*.

Notation on sequences. Let X be a finite set. For an infinite sequence $x = (x^\ell)_{\ell \geqslant 1} = (x^1, x^2, \ldots)$ of elements in X, we denote by x^ℓ the element in the ℓ-th position of x, and denote by $x(\ell) = (x^1, x^2, \ldots, x^\ell)$ the finite prefix of x up to position ℓ. We denote by X^∞ the set of all infinite sequences of elements from X. Given a function $f : X \to \mathbb{Z}$ (where $\mathbb{Z}$ is the set of integers) and a sequence $x \in X^\infty$, we denote with $f(x, k) = \sum_{\ell=1}^{k} f(x^\ell)$ the sum of the images of the first k elements.

Job sequences. 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* $\sigma = (\sigma^\ell)_{\ell \geqslant 1} \in \Sigma^\infty$, where $\Sigma = 2^{\mathcal{T}}$. If a task τ_i belongs to σ^ℓ, for $\ell \in \mathbb{N}^+$, 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 $J_{i,j}$ can be executed, and $d_{i,j} = j + D_i$ denotes its absolute *deadline*.

Admissible job sequences. We present a flexible framework where the set of admissible job sequences that the adversary can generate may be restricted. The set $\mathcal{J}$ of *admissible* job sequences from Σ^∞ can be obtained by imposing one or more of the following (optional) admissibility restrictions:

($\mathcal{S}$) Safety constraints, which are restrictions that hold in every finite prefix of a job sequence; e.g., they can be used to enforce job release constraints such as periodicity or sporadicity, and to impose temporal workload restrictions.

($\mathcal{L}$) Liveness constraints, which assert infinite repetition of certain patterns in a job sequence; e.g., they can be used to force the adversary to release a certain task infinitely often.

($\mathcal{W}$) Limit-average constraints, which restrict the long run average behavior of a job sequence; e.g., they can be used to enforce that the average load in the job sequences does not exceed a threshold.

Schedule. Given an admissible job sequence $\sigma \in \mathcal{J}$, the *schedule* $\pi = (\pi^\ell)_{\ell \geqslant 1} \in \Pi^\infty$, where $\Pi = ((\mathcal{T} \times \{0, \ldots, D_{\max} - 1\}) \cup \varnothing)$, computed by a real-time scheduling algorithm for σ, is a function that assigns at most one job for execution to every slot $\ell \geqslant 1$: π^ℓ is either $\varnothing$ (i.e., no job is executed) or else (τ_i, j) (i.e., the job $J_{i,\ell-j}$ of task τ_i released j slots ago is executed). The latter must satisfy the following constraints:

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

Note that our definition of schedules uses relative indexing in the scheduling algorithms: At time point ℓ, the algorithm for schedule π^ℓ uses index j to refer to slot $\ell - j$. Recall that $\pi(k)$ denotes the prefix of length $k \geqslant 1$ of π. We define $\gamma_i(\pi, k)$ to be the number of jobs of task τ_i that are completed by their deadlines in $\pi(k)$. The cumulated utility $V(\pi, k)$ (also called utility for brevity) achieved in $\pi(k)$ is defined as $V(\pi, k) = \sum_{i=1}^N \gamma_i(\pi, k) \cdot V_i$.

Competitive ratio. We are interested in evaluating the performance of deterministic *on-line* scheduling algorithms $\mathcal{A}$, which, at time ℓ, do not know any of the σ^k for $k > \ell$ when running on $\sigma \in \mathcal{J}$. In order to assess the performance of $\mathcal{A}$, we will compare the cumulated utility achieved in the schedule $\pi_\mathcal{A}$ to the cumulated utility achieved in the schedule $\pi_\mathcal{C}$ provided by an optimal *off-line* scheduling algorithm, called a *clairvoyant* algorithm $\mathcal{C}$, working on the same job sequence. Formally, given a taskset $\mathcal{T}$, let $\mathcal{J} \subseteq \Sigma^\infty$ be the set of all admissible job sequences of $\mathcal{T}$ that satisfy given (optional) safety, liveness, and limit-average constraints. For every $\sigma \in \mathcal{J}$, we denote with $\pi_\mathcal{A}^\sigma$ (resp. $\pi_\mathcal{C}^\sigma$) the schedule produced by $\mathcal{A}$ (resp. $\mathcal{C}$) under σ. The *competitive ratio* of the on-line algorithm $\mathcal{A}$ for the taskset $\mathcal{T}$ under the admissible job sequence set $\mathcal{J}$ is defined as

$$\mathcal{CR}_\mathcal{J}(\mathcal{A}) = \inf_{\sigma \in \mathcal{J}} \liminf_{k \to \infty} \frac{1 + V(\pi_\mathcal{A}^\sigma, k)}{1 + V(\pi_\mathcal{C}^\sigma, k)} \qquad (1)$$

that is, the worst-case ratio of the cumulated utility of the on-line algorithm versus the clairvoyant algorithm, under all admissible job sequences. Note that adding 1 in numerator and denominator simply avoids division by zero issues.

Remark 1. Since, according to the definition of the competitive ratio $\mathcal{CR}_\mathcal{J}$ in Equation (1), we focus on worst-case analysis, we do not consider randomized algorithms (such as Locke's best-effort policy [16]). Generally, for worst-case analysis, randomization can be handled by additional choices for the adversary. For the same reason, we do not consider scheduling algorithms that can use the unbounded history of job releases to predict the future (e.g., to capture correlations).

III. LTSs as Models for Algorithms

We will consider both on-line and off-line scheduling algorithms that are formally modeled as *labeled transition systems (LTSs)*: Every deterministic finite-state on-line scheduling algorithm can be represented as a deterministic LTS, such that every input job sequence generates a unique run that determines the corresponding schedule. On the other hand, an off-line algorithm can be represented as a non-deterministic LTS, which uses the non-determinism to guess the appropriate job to schedule.

Labeled transition systems (LTSs). Formally, a *labeled transition system* (LTS) is a tuple $L = (S, s_1, \Sigma, \Pi, \Delta)$, where S is a finite set of states, $s_1 \in S$ is the initial state, Σ is a finite set of input actions, Π is a finite set of output actions, and $\Delta \subseteq S \times \Sigma \times S \times \Pi$ is the transition relation. Intuitively, $(s, x, s', y) \in \Delta$ if, given the current state s and input x, the LTS outputs y and makes a transition to state s'. If the LTS is deterministic, then there is always a unique output and next state, i.e., Δ is a function $\Delta : S \times \Sigma \to S \times \Pi$. Given an input sequence $\sigma \in \Sigma^\infty$, a *run* of L on σ is a sequence $\rho = (p_\ell, \sigma_\ell, q_\ell, \pi_\ell)_{\ell \geqslant 1} \in \Delta^\infty$ such that $p_1 = s_1$ and for all $\ell \geqslant 2$, we have $p_\ell = q_{\ell-1}$. For a deterministic LTS, for each input sequence, there is a unique run.

Deterministic LTS for an on-line algorithm. For our analysis, on-line scheduling algorithms are represented as deterministic LTSs. Recall the definition of the sets $\Sigma = 2^\mathcal{T}$, and $\Pi = ((\mathcal{T} \times \{0, \ldots, D_{\max}-1\}) \cup \varnothing)$. Every deterministic on-line algorithm $\mathcal{A}$ that uses finite state space (for all job sequences) 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 $\Delta_\mathcal{A}$ correspond to the execution of $\mathcal{A}$ for 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 $r_\mathcal{A}(\delta) = V_i$ if the transition δ completes a job of task τ_i, and $r_\mathcal{A}(\delta) = 0$ otherwise. Given the unique run $\rho^\sigma = (\delta^\ell)_{\ell \geqslant 1}$ 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 $V(\rho^\sigma, k) = \sum_{\ell=1}^k r_\mathcal{A}(\delta^\ell)$.

Most scheduling algorithms (such as EDF, FIFO, DOVER, TD1) can be represented as a deterministic LTS. An illustration for EDF is given in the following example (see Appendix Section F for other examples).
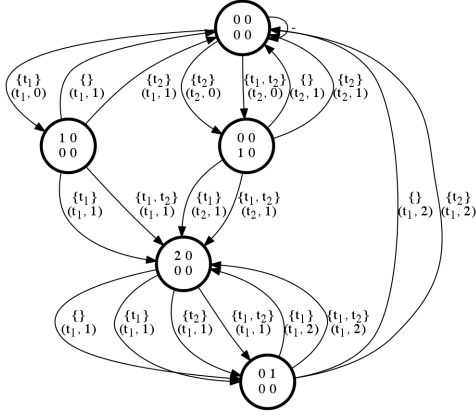
Fig. 1: EDF for $\mathcal{T} = \{\tau_1, \tau_2\}$ with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 2$, represented as a deterministic LTS.

Example 1. *Consider the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$, with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 2$. Figure 1 represents the EDF (Earliest Deadline First) scheduling policy as a deterministic LTS for $\mathcal{T}$. Each state is represented by a matrix M, such that $M[i,j]$, $1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant D_{\max} - 1$, denotes the remaining execution time of the job of task τ_i released j slots ago. Every transition is labeled with a set $T \in \Sigma$ of released tasks as well as with $(\tau_i, j) \in \Pi$, which denotes the unique job $J_{i,\ell-j}$ to be scheduled in the current slot ℓ. Released jobs with no chance of being scheduled are not included in the state space.*

The non-deterministic LTS. The clairvoyant algorithm $\mathcal{C}$ is formally a non-deterministic LTS $L_{\mathcal{C}} = (S_{\mathcal{C}}, s_{\mathcal{C}}, \Sigma, \Pi, \Delta_{\mathcal{C}})$ where each state in $S_{\mathcal{C}}$ is a $N \times (D_{\max} - 1)$ matrix M, such that for each time slot ℓ, the entry $M[i,j]$, $1 \leqslant i \leqslant N$, $1 \leqslant j \leqslant D_{\max} - 1$, denotes the remaining execution time of the job $J_{i,\ell-j}$ (i.e., the job of task i released j slots ago). For matrices M, M', subset $T \in \Sigma$ of newly released tasks, and scheduled job $P = (\tau_i, j) \in \Pi$, we have $(M, T, M', P) \in \Delta_{\mathcal{C}}$ iff $M[i,j] > 0$ and M' is obtained from M by

(1) inserting all $\tau_i \in T$ into M,
(2) decrementing the value at position $M[i,j]$, and
(3) shifting the contents of M by one column to the right.

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 $s_{\mathcal{C}}$ is represented by the zero $N \times (D_{\max} - 1)$ matrix, and $S_{\mathcal{C}}$ is the smallest $\Delta_{\mathcal{C}}$-closed set of states that contains $s_{\mathcal{C}}$ (i.e., if $M \in S_{\mathcal{C}}$ and $(M, T, M', P) \in \Delta_{\mathcal{C}}$ for some T, M' and P, we have $M' \in S_{\mathcal{C}}$). Finally, we associate with $L_{\mathcal{C}}$ a reward function $r_{\mathcal{C}} : \Delta_{\mathcal{C}} \to \mathbb{N}$ such that $r_{\mathcal{C}}(\delta) = V_i$ if the transition δ completes a task τ_i, and $r_{\mathcal{C}}(\delta) = 0$ otherwise.

IV. ADMISSIBLE JOB SEQUENCES AND OUR APPROACH

In this section we discuss our mechanisms for restricting the adversary to generate only certain admissible job sequences and then present our overall approach.

Admissible job sequences. Our framework allows to restrict the adversary to generate admissible job sequences $\mathcal{J} \subseteq \Sigma^{\infty}$,

which can be specified via different constraints. Since a constraint on job sequences can be interpreted as a language (which is a subset of infinite words Σ^{∞} here), we will use automata as acceptors of such languages. Since an automaton is a deterministic LTS with no output, all our constraints will be described as LTSs with an empty set of output actions. We allow the following types of contraints:

($\mathcal{S}$) Safety constraints are defined by a deterministic safety LTS $L_{\mathcal{S}} = (S_{\mathcal{S}}, s_{\mathcal{S}}, \Sigma, \varnothing, \Delta_{\mathcal{S}})$, with a distinguished *absorbing* reject state $s_r \in S_{\mathcal{S}}$. An absorbing state is a state that has outgoing transitions only to itself. Every job sequence σ defines a unique run $\rho_{\mathcal{S}}^{\sigma}$ in $L_{\mathcal{S}}$, such that either no transition to s_r appears in $\rho_{\mathcal{S}}^{\sigma}$, or every such transition is followed solely by self-transitions to s_r. A job sequence σ is *admissible* to $L_{\mathcal{S}}$, if $\rho_{\mathcal{S}}^{\sigma}$ does not contain a transition to s_r. To obtain a safety LTS that does not restrict $\mathcal{J}$ at all, we simply use a trivial deterministic $L_{\mathcal{S}}$ with no transition to s_r.

($\mathcal{L}$) Liveness constraints are defined by a deterministic liveness LTS $L_{\mathcal{L}} = (S_{\mathcal{L}}, s_{\mathcal{L}}, \Sigma, \varnothing, \Delta_{\mathcal{L}})$, with a distinguished *accept* state $s_a \in S_{\mathcal{L}}$. A job sequence σ is *admissible* to $L_{\mathcal{L}}$ if $\rho_{\mathcal{L}}^{\sigma}$ contains infinitely many transitions to s_a. For the case where there are no liveness constraint in $\mathcal{J}$, we use a LTS $L_{\mathcal{L}}$ consisting of state s_a only.

($\mathcal{W}$) Limit-average constraints are defined by a deterministic weighted LTS $L_{\mathcal{W}} = (S_{\mathcal{W}}, s_{\mathcal{W}}, \Sigma, \varnothing, \Delta_{\mathcal{W}})$ equipped with a weight function $w : \Delta_{\mathcal{W}} \to \mathbb{Z}^d$ that assigns a vector of weights to every transition. Given a threshold vector $\vec{\lambda} \in \mathbb{Q}^d$, where $\mathbb{Q}$ denotes the set of all rational numbers, a job sequence σ and the corresponding run $\rho_{\mathcal{W}}^{\sigma}$ of $L_{\mathcal{W}}$, the job sequence is *admissible* to $L_{\mathcal{W}}$ if $\liminf_{k \to \infty} \frac{1}{k} \cdot w(\rho_{\mathcal{W}}^{\sigma}, k) \leqslant \vec{\lambda}$.

Illustrations of admissible job sequences. We now illustrate the types of constraints that are supported by the above framework with some examples (see also Appendix Section B).

($\mathcal{S}$) *Safety constraints.* Safety constraints restrict the adversary to release job sequences, where every finite prefix satisfies some property (as they lead to the absorbing reject state s_r of $L_{\mathcal{S}}$ otherwise). Some well-known examples of safety constraints are (i) periodicity and/or sporadicity constraints, where there are fixed and/or a minumum time between the release of any two consecutive jobs of a given task, and (ii) absolute workload constraints [17], [18], where the total workload released in the last k slots, for some fixed k, is not allowed to exceed a threshold λ. For example, in case of absolute workload constraints, $L_{\mathcal{S}}$ simply encodes the workload in the last k slots in its state, and makes a transition to s_r whenever the workload exceeds λ.

($\mathcal{L}$) *Liveness constraints.* Liveness constraints force the adversary to release job sequences that satisfy some property infinitely often. For example, they could be used to guarantee that the release of some particular task τ_i does not eventually stall; the constraint is specified by a two-state LTS $L_{\mathcal{L}}$ that visits s_a whenever the current job set

includes τ_i. A liveness constraint can also be used to prohibit infinitely long periods of overload [3].

($\mathcal{W}$) *Limit-average constraints.* Consider a relaxed notion of workload constraints, where the adversary is restricted to generate job sequences whose *average* workload does not exceed a threshold λ. Since this constraint still allows "busy" intervals where the workload temporarily exceeds λ, it cannot be expressed as a safety constraint. To support such interesting average constraints of admissible job sequences, where the adversary is more relaxed than under absolute constraints, our framework explicitly supports limit-average constraints. Therefore, it is possible to express the average workload assumptions commonly used in the analysis of aperiodic task scheduling in soft-real time systems [19], [20]. Other interesting cases of limit-average constraints include restricting the average sporadicity, and, in particular, average energy: ensuring that the limit-average of the energy consumption is below a certain threshold is an important concern in modern real-time systems [10].

Remark 2. While in general constraints are encoded as independent automata, it is often possible to encode certain constraints directly in the non-deterministic LTS of the clairvoyant scheduler instead. In particular, this is true when restricting the limit-average workload, generating finite intervals of overload, and releasing a particular job infinitely often.

Overall approach for computing $\mathcal{CR}$. Our goal is to determine the worst-case competitive ratio $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ for a given on-line algorithm $\mathcal{A}$. The inputs to the problem are the given taskset $\mathcal{T}$, an on-line algorithm $\mathcal{A}$ specified as a deterministic LTS $L_{\mathcal{A}}$, and the safety, liveness, and limit-average constraints specified as deterministic LTSs $L_{\mathcal{S}}$, $L_{\mathcal{L}}$ and $L_{\mathcal{W}}$, respectively, which constrain the admissible job sequences $\mathcal{J}$. Our approach uses a reduction to a multi-objective graph problem, which consists of the following steps:

1) Construct a non-deterministic LTS $L_{\mathcal{C}}$ corresponding to the clairvoyant off-line algorithm $\mathcal{C}$. Note that since $L_{\mathcal{C}}$ is non-deterministic, for every admissible job sequence σ, there are many possible runs in $L_{\mathcal{C}}$, of course also including the runs with maximum cumulative utility.

2) Take the synchronous product LTS $L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}} \times L_{\mathcal{W}}$. By doing so, a path in the product graph corresponds to *identically* labeled paths in LTSs, and thus ensures that they agree on the same job sequence σ. This product can be represented by a multi-objective graph (see Section V).

3) Employ several optimizations in order to reduce the size of product graph (see Section VI and VII).

4) Determine $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ by reducing the computation of the ratio given in Equation (1) to solving a multi-objective problem on the product graph.

V. GRAPHS WITH MULTIPLE OBJECTIVES

In this section, we define various objectives on graphs and outline the algorithms to solve them. We later show how the competitive analysis of on-line schedulers reduces to the solution of this section.

Multi-graphs. A *multi-graph* $G = (V, E)$, 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. An *infinite path* ρ of G is an infinite sequence of edges $e^1, e^2, \ldots$ such that for all $i \geqslant 1$ with $e^i = (u^i, v^i)$, we have $v^i = u^{i+1}$. Every such path ρ induces a sequence of nodes $(u^i)_{i \geqslant 1}$, 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. Finally, we denote with Ω the set of all paths of G.

Objectives. Given a graph G, an objective Φ is a subset of Ω that defines the desired set of paths. We will consider safety, liveness, mean-payoff (limit-average), and ratio objectives, and their conjunction for multiple objectives.

Safety and liveness objectives. We consider safety and liveness objectives, both defined with respect to some subset of nodes $X, Y \subseteq V$. Given $X \subseteq V$, the *safety* objective defined as $\mathsf{Safe}(X) = \{\rho \in \Omega : \forall i \geqslant 1, \rho^i \notin X\}$, represents the set of all paths that never visit the set X. The *liveness* objective defined as $\mathsf{Live}(Y) = \{\rho \in \Omega : \forall j \exists i > j \text{ s.t. } \rho^i \in Y\}$ represents the set of all paths that visit Y infinitely often.

Mean-payoff and ratio objectives. We consider the mean-payoff and ratio objectives, defined with respect to a weight function and a threshold. A *weight function* $w : E \to \mathbb{Z}^d$ assigns to each edge of G a vector of d integers. A weight function naturally extends to paths, with $w(\rho, k) = \sum_{i=1}^{k} w(\rho^i)$. The *mean-payoff* of a path ρ is defined as:

$$\mathsf{MP}(w, \rho) = \liminf_{k \to \infty} \frac{1}{k} \cdot w(\rho, k);$$

i.e., it is the long-run average of the weights of the path. Given a weight function w and a threshold vector $\vec{\nu} \in \mathbb{Q}^d$, the corresponding objective is given as:

$$\mathsf{MP}(w, \vec{\nu}) = \{\rho \in \Omega : \mathsf{MP}(w, \rho) \leqslant \vec{\nu}\};$$

that is, the set of all paths such that the mean-payoff (or limit-average) of their weights is at most $\vec{\nu}$ (where we consider pointwise comparision for vectors). For weight functions w_1, $w_2 : E \to \mathbb{N}^d$, the *ratio* of a path ρ is defined as:

$$\mathsf{Ratio}(w_1, w_2, \rho) = \liminf_{k \to \infty} \frac{\vec{1} + w_1(\rho, k)}{\vec{1} + w_2(\rho, k)},$$

which denotes the limit infimum of the coordinate-wise ratio of the sum of weights of the two functions; $\vec{1}$ denotes the d-dimensional all-1 vector. Given weight functions w_1, w_2 and a threshold vector $\vec{\nu} \in \mathbb{Q}^d$, the ratio objective is given as:

$$\mathsf{Ratio}(w_1, w_2, \vec{\nu}) = \{\rho \in \Omega : \mathsf{Ratio}(w_1, w_2, \rho) \leqslant \vec{\nu}\}$$

that is, the set of all paths such that the ratio of cumulative rewards w.r.t w_1 and w_2 is at most $\vec{\nu}$.

Example 2. *Consider the multi-graph shown in Figure 2 with a weight function of dimension $d = 2$. Note that there are two edges from node 3 to node 5 (represented as edges $(3, 5, 1)$*

and $(3, 5, 2)$). In the graph we have a weight function with dimension 2. Note that the two edges from node 3 to node 5 have incomparable weight vectors.
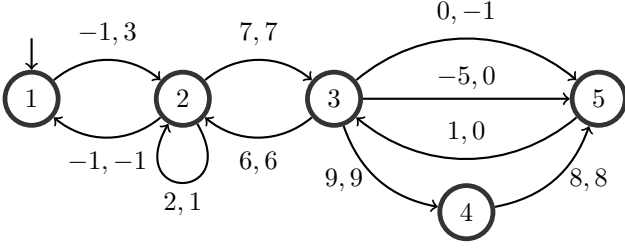
Fig. 2: An example of a multi-graph G.

Decision problem. The decision problem we consider is as follows: Given the graph G, an initial node $s \in V$, and an objective Φ (which can be a conjunction of several objectives), determine if there exists a path ρ that starts from s and belongs to Φ, i.e., $\rho \in \Phi$. For simplicity of presentation, we assume that every $u \in V$ is reachable from s (unreachable nodes can be discarded by preprocessing G in $O(m)$ time). We first present algorithms for each of safety, liveness, mean-payoff, and ratio obejctives separately, and then for their conjunction.

Algorithms for safety and liveness objectives.
1) *(Safety objectives).* The algorithm for the objective $\mathsf{Safe}(X)$ is straightforward. We first remove the set X of nodes, and iteratively remove nodes without outgoing edges. In the end, we obtain a graph $G = (V_X, E_X)$ such that $X \cap V_X = \varnothing$, and every node in V_X has an edge to a node in V_X. Thus, in the resulting graph, the objective $\mathsf{Safe}(X)$ is satisfied, and the algorithm answers yes iff $s \in V_X$. The algorithm requires $O(m)$ time.
2) *(Liveness objectives).* To solve for the objective $\mathsf{Live}(Y)$, initially perform an SCC (maximal strongly connected component) decomposition of G. We call an SCC V_{SCC} *live*, if (i) either $|V_{\mathsf{SCC}}| > 1$, or $V_{\mathsf{SCC}} = \{u\}$ and $(u, u) \in E$; and (ii) $V_{\mathsf{SCC}} \cap Y \neq \varnothing$. Then $\mathsf{Live}(Y)$ is satisfied in G iff there exists a live SCC V_{SCC} that is reachable from s (since every node in a live SCC can be visited infinitely often). Using for example the algorithm of [21] for the SCC decomposition also requires $O(m)$ time.

Algorithms for mean-payoff objectives. We distinguish between the case when the weight function has a single dimension ($d = 1$) versus the case when the weight function has multiple dimensions ($d > 1$).
1) *(Single dimension).* In the case of a single-dimensional weight function, a single weight is assigned to every edge, and the decision problem of the mean-payoff objective reduces to determining the mean weight of a minimum-weight simple cycle in G, as the latter also determines the mean-weight by infinite repetition. Using the algorithms of [22], [23], this process requires $O(n \cdot m)$ time. When the objective is satisfied, the process also returns a simple cycle C, as a witness to the objective. From C, a path $\rho \in \mathsf{MP}(w, \vec{\nu})$ is constructed by infinite repetitions of C.

2) *(Multiple dimensions).* When $d > 1$, the mean-payoff objective reduces to determining the feasibility of a linear program (LP). For $u \in V$, let $\mathsf{IN}(u)$ be the set of incoming, and $\mathsf{OUT}(u)$ the set of outgoing edges of u. As shown in [5], [24], G satisfies $\mathsf{MP}(w, \vec{\nu})$ iff the following set of constraints on $\vec{x} = (x_e)_{e \in E_{\mathsf{SCC}}}$ with $x_e \in \mathbb{Q}$ is satisfied simultaneously on some SCC V_{SCC} of G with induced edges $E_{\mathsf{SCC}} \subseteq E$.

$$
\begin{aligned}
x_e &\geqslant 0 & e \in E_{\mathsf{SCC}} \\
\sum_{e \in \mathsf{IN}(u)} x_e &= \sum_{e \in \mathsf{OUT}(u)} x_e & u \in V_{\mathsf{SCC}} \quad (2) \\
\sum_{e \in E_{\mathsf{SCC}}} x_e \cdot w(e) &\leqslant \vec{\nu} \\
\sum_{e \in E_{\mathsf{SCC}}} x_e &\geqslant 1
\end{aligned}
$$

The quantities x_e are intuitively interpreted as "flows". The first constraint specifies that the flow of each edge is non-negative. The second constraint is a flow-conservation constraint. The third constraint specifies that the objective is satisfied if we consider the relative contribution of the weight of each edge, according to the flow of the edge. The last constraint asks that the preceding constraints are satisfied by a non-trivial (positive) flow. Hence, when $d > 1$, the decision problem reduces to solving a LP, and the time complexity is polynomial [25].

Witness construction. The witness path construction from a feasible solution consists of two steps: (A) Construction of a multi-cycle from the feasible solution; and (B) Construction of an infinite witness path from the multi-cycle. We describe the two steps in detail. Formally, a *multi-cycle* is a finite set of cycles with multiplicity $\mathcal{MC} = \{(C_1, m_1), (C_2, m_2), \dots, (C_k, m_k)\}$, such that every C_i is a simple cycle and m_i is its multiplicity. The construction of a multi-cycle from a feasible solution $\vec{x}$ is as follows. Let $\mathcal{E} = \{e : x_e > 0\}$. By scaling each edge flow x_e by a common factor z, we construct the set $\mathcal{X} = \{(e, z \cdot x_e) : e \in \mathcal{E}\}$, with $\mathcal{X} \subset E_{\mathsf{SCC}} \times \mathbb{N}^+$. Then, we start with $\mathcal{MC} = \varnothing$ and apply iteratively the following procedure until $\mathcal{X} = \varnothing$: (i) find a pair $(e_i, m_i) = \arg\min_{(e_j, m_j) \in \mathcal{X}} m_j$, (ii) form a cycle C_i that contains e_i and only edges that appear in $\mathcal{X}$ (because of Equation (2), this is always possible), (iii) add the pair (C_i, m_i) in the multi-cycle $\mathcal{MC}$, (iv) subtract m_i from all elements (e_j, m_j) of $\mathcal{X}$ such that the edge e_j appears in C_i, (v) remove from $\mathcal{X}$ all $(e_j, 0)$ pairs, and repeat. Since V_{SCC} is an SCC, there is a path $C_i \rightsquigarrow C_j$ for all C_i, C_j in $\mathcal{MC}$. Given the multi-cycle $\mathcal{MC}$, the infinite path that achieves the weight at most $\vec{\nu}$ is not periodic, but generated by Procedure 1.

Algorithm for ratio objectives. We now consider ratio objectives, and present a reduction to mean-payoff objectives. Consider the weight functions w_1, w_2 and the threshold vector $\vec{\nu} = \frac{\vec{p}}{\vec{q}}$ as the component-wise division of vectors $\vec{p}, \vec{q} \in \mathbb{N}^d$.

Procedure 1: Multi-objective witness

Input: A graph $G = (V, E)$, and a multi-cycle
$\qquad \mathcal{MC} = \{(C_1, m_1), (C_2, m_2), \ldots, (C_k, m_k)\}$
Output: An infinite path $\rho \in \mathsf{MP}(w, \vec{\nu})$

```
1  ℓ ← 1
2  while True do
3      Repeat C₁ for ℓ · m₁ times
4      C₁ ⤳ C₂
5      Repeat C₂ for ℓ · m₂ times
6      ...
7      Repeat Cₖ for ℓ · mₖ times
8      Cₖ ⤳ C₁
9      ℓ ← ℓ + 1
10 end
```

We define a new weight function $w : E \to \mathbb{Z}^d$ such that for all $e \in E$, we have $w(e) = \vec{q} \cdot w_1(e) - \vec{p} \cdot w_2(e)$ (where $\cdot$ denotes component-wise multiplication). It is easy to verify that $\mathsf{Ratio}(w_1, w_2, \vec{\nu}) = \mathsf{MP}(w, \vec{0})$, and thus we solve the ratio objective by solving the new mean-payoff objective, as described above.

Algorithms for conjunctions of objectives. Finally, we consider the conjunction of a safety, a liveness, and a mean-payoff objective (note that we have already described a reduction of ratio objectives to mean-payoff objectives). More specifically, given a weight function w, a threshold vector $\vec{\nu} \in \mathbb{Q}$, and sets $X, Y \subseteq V$, we consider the decision problem for the objective $\Phi = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{MP}(w, \vec{\nu})$. The procedure is as follows:

1) Initially compute G_X from G as in the case of a single safety objective.
2) Then, perform an SCC decomposition on G_X.
3) For every live SCC V_{SCC} that is reachable from s, solve for the mean-payoff objective in V_{SCC}. Return yes, if $\mathsf{MP}(w, \vec{\nu})$ is satisfied in any such V_{SCC}.

If the answer to the decision problem is yes, then the witness consists of a live SCC V_{SCC}, along with a multi-cycle (resp. a cycle for $d = 1$). The witness infinite path is constructed as in Procedure 1, with the only difference that at end of each while loop a live node from Y in the SCC V_{SCC} is additionally visited. The time required for the conjunction of objectives is dominated by the time required to solve for the mean-payoff objective. Figure 2 provides a relevant example.

Example 3. *Consider the graph in Figure 2. Starting from node 1, the mean-payoff-objective* $\mathsf{MP}(w, \vec{0})$ *is satisfied by the multi-cycle* $\mathcal{MC} = \{(C_1, 1), (C_2, 2)\}$*, with* $C_1 = ((1, 2), (2, 1))$ *and* $C_2 = ((3, 5), (5, 3))$*. A solution to the corresponding LP is* $x_{(1,2)} = x_{(2,1)} = \frac{1}{3}$ *and* $x_{(3,5)} = x_{(5,3)} = \frac{2}{3}$*, and* $x_e = 0$ *for all other* $e \in E$*. Procedure 1 then generates a witness path for the objective. The objective is also satisfied in conjuction with* $\mathsf{Safe}(\{4\})$ *or* $\mathsf{Live}(\{4\})$*. In the latter case, a witness path additionally traverses the edges* $(3, 4)$ *and* $(4, 5)$ *before transitioning from* C_1 *to* C_2*.*

Theorem 1 summarizes the results of this section.

Theorem 1. *Let $G = (V, E)$ be a graph, $s \in V$, $X, Y \subseteq V$, $w : E \to \mathbb{Z}^d$, $w_1, w_2\; E \to \mathbb{N}^d$ weight functions, and $\vec{\nu} \in \mathbb{Q}^d$. Let $\Phi_1 = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{MP}(w, \vec{\nu})$ and $\Phi_2 = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(w_1, w_2, \vec{\nu})$. The decision problem of whether G satisfies the objective Φ_1 (resp. Φ_2) from s requires*
  1) *$O(n \cdot m)$ time, if $d = 1$.*
  2) *Polynomial time, if $d > 1$.*
If the objective Φ_1 (resp. Φ_2) is satisfied in G from s, then a finite witness (an SCC and a cycle for single dimension, and an SCC and a multi-cycle for multiple dimensions) exists and can be constructed in polynomial time.

VI. REDUCTION

We present a formal reduction of the computation of the competititve ratio of an on-line scheduling algorithm with constraints on job sequences to the multi-objective graph problem. The input consists of the taskset, a deterministic LTS for the on-line algorithm, and optional deterministic LTSs for the constraints.

Reduction. We first describe the process of computing the competitive ratio $\mathcal{CR}_{\mathcal{J}}(\mathcal{A})$ where $\mathcal{J}$ is a set of job sequences only subject to safety and liveness constraints. We later show how to handle limit-average constraints.

Given the deterministic and non-deterministic LTS $L_{\mathcal{A}}$ and $L_{\mathcal{C}}$ with reward functions $r_{\mathcal{A}}$ and $r_{\mathcal{C}}$, respectively, and optionally safety and liveness LTS $L_{\mathcal{S}}$ and $L_{\mathcal{L}}$, let $L = L_{\mathcal{A}} \times L_{\mathcal{C}} \times L_{\mathcal{S}} \times L_{\mathcal{L}}$ be their synchronous product (refer to Appendix Section A for the formal definition of synchronous product). Hence, L is a non-deterministic LTS $(S, s_1, \Sigma, \Pi, \Delta)$, and every job sequence σ yields a set of runs R in L, such that each $\rho \in R$ captures the joint behavior of $\mathcal{A}$ and $\mathcal{C}$ under σ. Note that for each such ρ the behavior of $\mathcal{A}$ is unchanged, but the behavior of $\mathcal{C}$ generally varies, due to non-determinism. Let $G = (V, E)$ be the multi-graph induced by L, that is, $V = S$ and $(M, M', j) \in E$ for all $1 \leqslant j \leqslant i$ iff there are i transitions $(M, T, M', P) \in \Delta$. Let $w_{\mathcal{A}}$ and $w_{\mathcal{C}}$ be the weight functions that assign to each edge of G the reward that the respective algorithm obtains from the corresponding transition in L. Let X be the set of states in G whose $L_{\mathcal{S}}$ component is s_r, and Y the set of states in G whose $L_{\mathcal{L}}$ component is s_a. It follows that for all $\nu \in \mathbb{Q}$, we have that $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ iff the objective $\Phi_{\nu} = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(w_{\mathcal{A}}, w_{\mathcal{C}}, \nu)$ is satisfied in G from the state s_1. As the dimension in the ratio objective is one, Case 1 of Theorem 1 applies, and we obtain the following:

Lemma 1. *Given the product graph $G = (V, E)$ of n nodes and m edges, a rational $\nu \in \mathbb{Q}$, and a set of job sequences $\mathcal{J}$ admissible to safety and liveness LTSs, determining whether $\mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant \nu$ requires $O(n \cdot m)$ time.*

Since $0 \leqslant \mathcal{CR}_{\mathcal{J}}(\mathcal{A}) \leqslant 1$, the problem of determining the competitive ratio reduces to finding $v = \sup\{\nu \in \mathbb{Q} : \Phi_{\nu}$ is satisfied in $G\}$. Because this value corresponds to the ratio of the corresponding rewards obtained in a simple cycle in G, it follows that v is the maximum of a finite set, and can be determined exactly by an *adaptive binary search* (see Appendix Section C and Algorithm 2 for details).

Finally, we turn our attention to limit-average constraints and the LTS $L_\mathcal{W}$. We follow a similar approach as above, but this time including $L_\mathcal{W}$ in the synchronous product, i.e., $L = L_\mathcal{A} \times L_\mathcal{C} \times L_\mathcal{S} \times L_\mathcal{L} \times L_\mathcal{W}$. Let $w_\mathcal{A}$ and $w_\mathcal{C}$ be weight functions that assign to each edge $e \in E$ in the corresponding multi-graph a vector of $d+1$ weights as follows. In the first dimension, $w_\mathcal{A}$ and $w_\mathcal{C}$ are defined as before, assigning to each edge of G the corresponding rewards of $\mathcal{A}$ and $\mathcal{C}$. In the remaining d dimensions, $w_\mathcal{C}$ is always 1, whereas $w_\mathcal{A}$ equals the value of the weight function w of $L_\mathcal{W}$ on the corresponding transition. Let $\vec{\lambda}$ be the threshold vector of $L_\mathcal{W}$. It follows that for all $\nu \in \mathbb{Q}$, we have that $\mathcal{CR}_\mathcal{J}(\mathcal{A}) \leqslant \nu$ iff the objective $\Phi_\nu = \mathsf{Safe}(X) \cap \mathsf{Live}(Y) \cap \mathsf{Ratio}(w_\mathcal{A}, w_\mathcal{C}, (\nu, \vec{\lambda}))$ is satisfied in G from the state s that corresponds to the initial state of each LTS, where $(\nu, \vec{\lambda})$ is a $d+1$-dimension vector, with ν in the first dimension, followed by the d-dimension vector $\vec{\lambda}$. As the dimension in the ratio objective is greater than one, Case 2 of Theorem 1 applies, and we obtain the following:

Lemma 2. *Given the product graph $G = (V, E)$ of n nodes and m edges, a rational $\nu \in \mathbb{Q}$, and a set of job sequences $\mathcal{J}$ admissible to safety, liveness, and limit average LTSs, determining whether $\mathcal{CR}_\mathcal{J}(\mathcal{A}) \leqslant \nu$ requires polynomial time.*

Again, since $0 \leqslant \mathcal{CR}_\mathcal{J}(\mathcal{A}) \leqslant 1$, the competitive ratio is determined by an adaptive binary search (similar to Algorithm 2 in the Appendix). However, this time $\mathcal{CR}_\mathcal{J}(\mathcal{A})$ is not guaranteed to be realized by a simple cycle (the witness path in G is not necessarily periodic, see Procedure 1), and is only approximated within some desired error threshold $\epsilon > 0$.

VII. Optimized Reduction

In Section VI, we have established a formal reduction from determining the competitive ratio of an on-line scheduling algorithm in a constrained adversarial environment to solving multiple objectives on graphs. In the current section, we present several optimizations in this reduction that significantly reduce the size of the generated LTSs.

Clairvoyant LTS. Recall the clairvoyant LTS $L_\mathcal{C}$ with reward function $r_\mathcal{C}$ from Section III that non-deterministically models a scheduler. Now we encode the off-line algorithm as a non-deterministic LTS $L'_\mathcal{C} = (S'_\mathcal{C}, s'_\mathcal{C}, \Sigma, \varnothing, \Delta'_\mathcal{C})$ with reward function $r'_\mathcal{C}$ that lacks the property of being a scheduler, as information about released and scheduled jobs is lost. However, it preserves the property that, given a job sequence σ, there exists a run $\rho^\sigma_\mathcal{C}$ in $L_\mathcal{C}$ iff there exists a run $\widehat{\rho}^\sigma_\mathcal{C}$ in $L'_\mathcal{C}$ with $V(\rho^\sigma_\mathcal{A}, k) = V(\widehat{\rho}^\sigma_\mathcal{A}, k)$ for all $k \in \mathbb{N}^+$. That is, there is a bisimulation between $L_\mathcal{C}$ and $L'_\mathcal{C}$ that preserves rewards.

Intuitively, the clairvoyant algorithm need not partially schedule a task, i.e., it will either discard it immediately, or schedule it to completion. Hence, in every release of a set of tasks T, $L'_\mathcal{C}$ non-deterministically chooses a subset $T' \subseteq T$ to be scheduled, as well as allocates the future slots for their execution. Once these slots are allocated, $L'_\mathcal{C}$ is not allowed to preempt those in favor of a subsequent job.

The state space $S'_\mathcal{C}$ of $L'_\mathcal{C}$ consists of binary strings of length $D_{\max}$. For a binary string $B \in S'_\mathcal{C}$, we have $B[i] = 1$ iff the i-th slot in the future is allocated to some released job, and

$s'_\mathcal{C} = \vec{0}$. Informally, the transition relation $\Delta'_\mathcal{C}$ is such that, given a current subset $T \subseteq \Sigma$ of released jobs, there exists a transition δ from B to B' only if B' can be obtained from B by non-deterministically choosing a subset $T' \subseteq T$, and for each task $\tau_i \in T'$ allocating non-deterministically C_i free slots in B. Finally, set $r'_\mathcal{C} = \sum_{\tau_i \in T'} V_i$.

By definition, $|S'_\mathcal{C}| \leqslant 2^{D_{\max}}$. In laxity-restricted tasksets, we can obtain an even tighter bound. Let $L_{\max} = \max_{\tau_i \in \mathcal{T}}(D_i - C_i)$ be the maximum laxity in $\mathcal{T}$, and $I : S'_\mathcal{C} \to \{\perp, 1, \ldots, D_{\max} - 1\}^{L_{\max}+1}$ a function such that $I(B) = (i_1, \ldots, i_{L_{\max}+1})$ are the indexes of the first $L_{\max}+1$ zeros in B. That is, $i_j = k$ iff $B[k]$ is the j-th zero location in B, and $i_j = \perp$ if there are less than j free slots in B.

Claim 1. *The function I is bijective.*

Proof. Fix a tuple $(i_1, \ldots, i_{L_{\max}+1})$, and let $B \in S'_\mathcal{C}$ be any state such that $I(B) = (i_1, \ldots, i_{L_{\max}+1})$. We consider two cases.

1) If $i_{L_{\max}+1} = \perp$, there are less than $L_{\max}+1$ empty slots in B, all uniquely determined by $(i_1, \ldots, i_k)$, for some $k \leqslant L_{\max}$.
2) If $i_{L_{\max}+1} \neq \perp$, then all $i_j \neq \perp$, and thus any job to the right of $i_{L_{\max}+1}$ would have been stalled for more than $L_{\max}$ positions. Hence, all slots to the right of $i_{L_{\max}+1}$ are free in B, and B is also unique.

Hence, $I(B)$ always uniquely determines B, as desired. $\square$

For $x, k \in \mathbb{N}^+$, denote with $\mathsf{Perm}(x, k) = x \cdot (x-1) \ldots (x-k+1)$ the number of k-permutations on a set of size x.

Lemma 3. *Let $\mathcal{T}$ be a taskset with maximum deadline $D_{\max}$, and $L_{\max} = \max_{\tau_i \in \mathcal{T}}(D_i - C_i)$ be the maximum laxity. Then, $|S'_\mathcal{C}| \leqslant \min(2^{D_{\max}}, \mathsf{Perm}(D_{\max}, L_{\max} + 1))$.*

Hence, for zero and small laxity environments [3], as e.g. arising in wormhole switching in NoCs [26], $S'_\mathcal{C}$ has polynomial size in $D_{\max}$ (also see Appendix Section E for zero-laxity tasksets with large $D_{\max}$).

Clairvoyant LTS generation. We now turn our attention on efficiently generating the clairvoyant LTS $L'_\mathcal{C}$ as described in the previous paragraph. There is non-determinism in two steps: both in choosing the subset $T' \subseteq T$ of the currently released tasks for execution, and in allocating slots for executing all tasks in T'. Given a current state B and T, this non-determinism leads to several identical transitions δ to a state B'. We have developed a recursive algorithm called ClairvoyantSuccessor (Algorithm 1) that generates each such transition δ exactly once.

The intuition behind ClairvoyantSuccessor is as follows. It has been shown that the earliest deadline first (EDF) policy is optimal in scheduling job sequences where every released task can be completed [27]. By construction, given a job sequence σ_1, $L'_\mathcal{C}$ non-deterministically chooses a job sequence σ_2, such that for all ℓ, we have $\sigma_2^\ell \subseteq \sigma_1^\ell$, and all jobs in σ_2 are scheduled to completion by $L'_\mathcal{C}$. Therefore, it suffices to consider a transition relation $\Delta'_\mathcal{C}$ that allows at least all possible choices that admit a feasible EDF schedule on every possible σ_2, for any generated job sequence σ_1.

In more detail, ClairvoyantSuccessor is called with a current state B, a subset of released tasks T and an index k, and returns the set $\mathcal{B}$ of all possible successors of B that schedule a subset $T' \subseteq T$, and every job of T' is executed later than k slots in the future. This is done by extracting from T the task τ with the earliest deadline, and proceeding as follows: The set $\mathcal{B}$ is obtained by constructing a state B' that considers all the possible ways to schedule τ to the right of k (including the possibility of not scheduling τ at all), and recursively finding all the ways to schedule $T \setminus \{\tau\}$ in B', to the right of the rightmost slot allocated for task τ.

Finally, we exploit the following two observations to further reduce the state space of $L'_{\mathcal{C}}$. First, we note that as long as there is some load in the state of $L'_{\mathcal{C}}$ (i.e., at least one bit of B is one), the clairvoyant algorithm gains no benefit by not executing any job in the current slot. Hence, besides the zero state $\bar{0}$, every state B must have $B[1] = 1$. In most cases, this restriction reduces the state space by at least 50%. Second, it follows from our claims on the off-line EDF policy of the clairvoyant scheduler that for every two scheduled jobs J and J', it will never have to preempt J for J' and vice versa. A consequence of this is that, for every state B and every continuous segment of zeros in B that is surrounded by ones (called a *gap*), the gap must be able to be completely filled with some jobs that start and end inside the gap. This reduces to solving a knapsack problem [28] where the size of the knapsack is the length of the gap, and the set of items is the whole taskset $\mathcal{T}$ (with multiplicities). We note that the problem has to be solved on identical inputs a large number of times, and techniques such as *memoization* are employed to avoid multiple evaluations of the same input.

These two improvements were found to reduce the state space by a factor up to 90% in all examined cases (see Section VIII and Table II), and despite the non-determinism, in all reported cases the generation of $L_{\mathcal{C}}$ was done in less than a second.

On-line state space reduction. Typically, most on-line scheduling algorithms do "lazy dropping" of the jobs, where a job is dropped only when its deadline passes. To keep the state-space of the LTS small, it is crucial to only store those jobs that have the possibility of being scheduled, at least partially, under some sequence of future task releases. We do so by first creating the LTS naively, and then iterating through its states. For each state s and job $J_{i,j}$ in s with relative deadline D_i, we perform a *depth-limited search* originating in s for D_i steps, looking for a state s' reached by a transition that schedules $J_{i,j}$. If no such state is found, we merge state s to s'', where s'' is identical to s without job $J_{i,j}$.

Algorithm 1: ClairvoyantSuccessor

Input: A set $T \subseteq \mathcal{T}$, state B, index $1 \leqslant k \leqslant D_{\max}$
Output: A set $\mathcal{B}$ of successor states of B

1   **if** $T = \varnothing$ **then** **return** $\{B\}$;
2   $\tau \leftarrow \arg\min_{\tau_i \in T} D_i$, $C \leftarrow$ execution time of τ
3   $T' \leftarrow T \setminus \{\tau\}$
    // Case 1: τ is not scheduled
4   $\mathcal{B} \leftarrow$ ClairvoyantSuccessor(T', B, k)
    // Case 2: τ is scheduled
5   $\mathcal{F} \leftarrow$ set of free slots in B greater than k
6   **foreach** $F \subseteq \mathcal{F}$ *with* $|F| = C$ **do**
7      $B' \leftarrow$ Allocate F in B
8      $k' \leftarrow$ rightmost slot in F
9      $\mathcal{B}' \leftarrow$ ClairvoyantSuccessor(T', B', k')
       // Keep only non-redundant states
10   **foreach** $B'' \in \mathcal{B}'$ **do**
11   **if** $B''[1] = 1$ *and* knapsack$(B'', \mathcal{T})$ **then**
12          $\mathcal{B} \leftarrow \mathcal{B} \cup \{B''\}$
13   **end**
14   **end**
15   **end**
16   **return** $\mathcal{B}$


VIII. EXPERIMENTAL RESULTS

We have implemented our approach for automated competitive ratio analysis, and applied it to a range of case studies: four well-known scheduling policies, namely, EDF (Earliest Deadline First), SRT (Shortest Remaining Time), SP (Static Priorities), and FIFO (First-in First-out), as well as some more elaborate algorithms that provide non-trivial performance guarantees, in particular, DSTAR [29] and DOVER [1] (see also TD1 [3] in Appendix Section E), are analyzed under a variety of tasksets. Our implementation is done in Python and C, and uses the lp_solve [30] package for linear programming solutions. All experiments are run on a standard 2010 computer with a 3.2GHz CPU and 4GB of RAM running Linux.

Varying tasksets without constraints. The algorithm DOVER was proved in [1] to have optimal competitive factor, i.e., optimal competitive ratio under the worst-case taskset. However, our experiments reveal that this performance guarantee is not universal, in the sense that DOVER is outperformed by other schedulers for specific tasksets. This observation applies to all on-line algorithms examined: As shown in Figure 3, even without constraints on the adversary, for every scheduling algorithm, there are tasksets in which it achieves the highest competitive ratio among all others. Note that this high variability of the optimal on-line algorithm across tasksets makes our automated analysis framework an interesting tool for the application designer.

Fixed taskset with varying constraints. We also consider fixed tasksets under various constraints (such as sporadicity or workload restrictions) for admissible job sequences. Figure 4 shows our experimental results for workload safety constraints, which again reveal that, depending on workload constraints, we can have different optimal schedulers. Finally, we consider limit-average constraints and observe that varying these constraints can also vary the optimal scheduler for a fixed taskset:
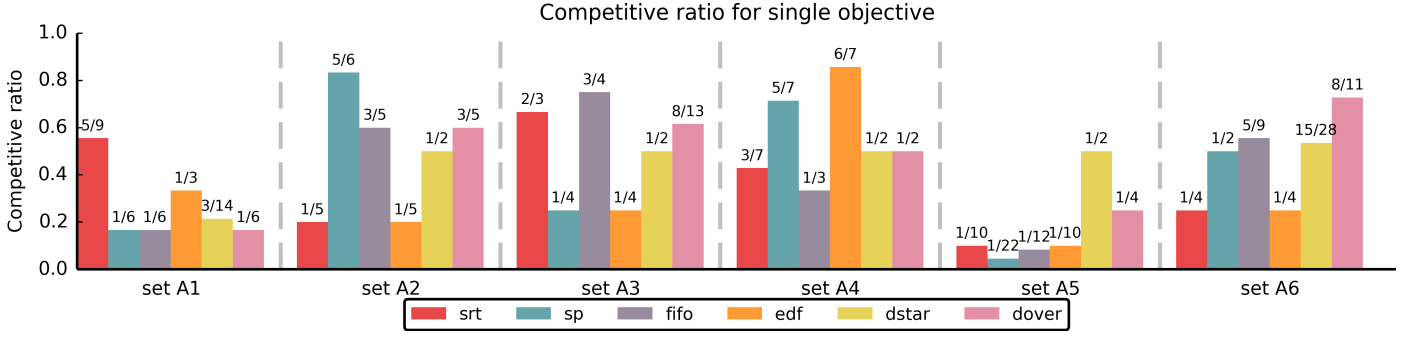
Fig. 3: The competitive ratio of the examined algorithms in various tasksets under no constraints; the tasksets A1-A6 are available in the Appendix Section D. Every examined algorithm is optimal in some taskset, among all others.
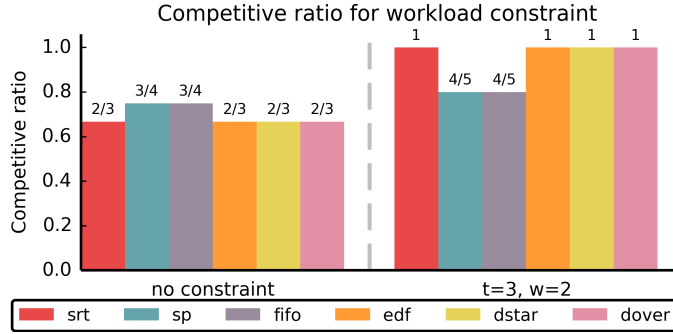
Fig. 4: Restricting the absolute workload generated by the adversary typically increases the competitive ratio, and can vary the optimal scheduler. On the left, the performance of each scheduler is evaluated without restrictions: FIFO, SP behave best. When restricting the adversary to at most 2 units of workload in the last 3 rounds, FIFO and SP become suboptimal, and are outperformed by other schedulers. The taskset is available in the Appendix Section D.

	1.5	1	0.8	0.6	0.4	0.3	0.1	0.078	0.05
fifo	✓	✓	✓	✓	✓				✓
sp	✓						✓		✓
srt	✓				✓	✓	✓	✓	✓

Table I: Columns show the mean workload restriction. The check-marks indicate that the corresponding scheduler is optimal for that mean workload restriction, among the six schedulers we examined. We see that the optimal scheduler can vary as the restrictions are tighter, and in a non-monotonic way. EDF, DSTAR and DOVER were not optimal in any case and hence not mentioned. The taskset is available in the Appendix Section D.

As Table I shows, the optimal scheduler can vary highly and non-monotonically with stronger limit-average workload restrictions.

Running times. Table II summarizes some key parameters of our various tasksets, and gives some statistical data on the observed running times in our respective experiments. Even though the considered tasksets are small, the very short running times of our prototype implementation reveal the

Name	N	D_{max}	Size (nodes)		Time (s)	
			Clairv.	Product	Mean	Max
set B01	2	7	19	823	0.04	0.05
set B02	2	8	26	1997	0.39	0.58
set B03	2	9	34	4918	10.02	15.21
set B04	3	7	19	1064	0.14	0.40
set B05	3	8	26	1653	0.66	2.05
set B06	3	9	34	7705	51.04	136.62
set B07	4	7	19	1711	2.13	6.34
set B08	4	8	26	3707	13.88	34.12
set B09	4	9	44	10040	131.83	311.94
set B10	5	7	19	2195	5.73	16.42
set B11	5	8	32	9105	142.55	364.92
set B12	5	9	44	16817	558.04	1342.59

Table II: Scalability of our approach for tasksets of various sizes N and D_{max}. For each taskset, the size of the state space of the clairvoyant scheduler is shown, along with the mean size of the product LTS, and the mean and maximum time to solve one instance of the corresponding ratio objective.

principal feasibility of our approach. We believe that further application-specific optimizations, augmented by abstraction and symmetry reduction techniques, will allow to scale to larger applications.

IX. CONCLUSIONS

We presented a flexible framework for automatically analyzing the competitive ratio of on-line scheduling algorithms for an input firm-deadline taskset, which also supports various forms of constraints for admissible job sequences. Our experimental results demonstrate that it allows to solve small-sized problem instances efficiently. Moreover, they highlight the importance of our fully automated approach, as there is neither a "universally" optimal algorithm for all tasksets (even in the absence of additional constraints) nor an optimal algorithm for different constraints in the same taskset. Thanks to the flexibility of our approach, it can be extended in various ways (multiple processors, algorithm-specific constraints like energy restrictions, more general deadlines, etc.). Part of our future research will be devoted to incorporating such features. We will also study abstraction and symmetry reduction techniques that will allow our approach to scale to larger instances.

References

[1] G. Koren and D. Shasha, "D^{over}: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems," *SIAM J. Comp.*, 1995.

[2] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the competitiveness of on-line real-time task scheduling," *Real-Time Syst.*, 1992.

[4] K. Chatterjee, A. Kößler, and U. Schmid, "Automated analysis of real-time scheduling using graph games," in *HSCC'13*, 2013.

[5] Y. Velner, K. Chatterjee, L. Doyen, T. A. Henzinger, A. Rabinovich, and J.-F. Raskin, "The complexity of multi-mean-payoff and multi-energy games," *CoRR*, 2012.

[6] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*. Cambridge University Press, 2007.

[7] E. Koutsoupias, "Scheduling without payments," in *SAGT'11*, 2011.

[8] K. Altisen, G. Gößler, and J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," *Real-Time Systems*, 2002.

[9] V. Bonifaci and A. Marchetti-Spaccamela, "Feasibility analysis of sporadic real-time multiprocessor task systems," *Algorithmica*, 2012.

[10] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Trans. Comput.*, 2004.

[11] V. Devadas, F. Li, and H. Aydin, "Competitive analysis of online real-time scheduling algorithms under hard energy constraint," *Real-Time Syst.*, 2010.

[12] S. K. Baruah and M. E. Hickey, "Competitive on-line scheduling of imprecise computations," *IEEE Trans. Comput.*, 1998.

[13] S. K. Baruah and J. R. Haritsa, "Scheduling for overload in real-time systems," *IEEE Trans. Comput.*, 1997.

[14] M. A. Palis, "Competitive algorithms for fine-grain real-time scheduling," in *RTSS'04*, 2004.

[15] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *RTSS'97*, 1997.

[16] C. D. Locke, "Best-effort decision-making for real-time scheduling," Ph.D. dissertation, CMU, Pittsburgh, PA, USA, 1986.

[17] S. J. Golestani, "A framing strategy for congestion management," *IEEE J.Sel. A. Commun.*, vol. 9, 1991.

[18] R. Cruz, "A calculus for network delay. I. network elements in isolation," *IEEE Trans. Information Theory*, vol. 37, no. 1, 1991.

[19] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS'98*, 1998.

[20] J. R. Haritsa, M. J. Carey, and M. Livny, "On being optimistic about real-time constraints," in *PODS'90*, 1990.

[21] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, 1972.

[22] R. M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Mathematics*, 1978.

[23] O. Madani, "Polynomial value iteration algorithms for deterministic MDPs," in *UAI'02*, 2002.

[24] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin, "Generalized mean-payoff and energy games." in *FSTTCS*, 2010.

[25] L. G. Khachiyan, "A polynomial algorithm in linear programming," *Doklady Akademii Nauk SSSR*, vol. 244, 1979.

[26] Z. Lu and A. Jantsch, "Admitting and ejecting flits in wormhole-switched networks on chip." *IET Computers & Digital Techniques*, vol. 1, 2007.

[27] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," in *IFIP Congress*, 1974.

[28] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*. Springer US, 1972.

[29] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, "On-line scheduling in the presence of overload," in *FOCS'91*, 1991.

[30] M. Berkelaar, K. Eikland, and P. Notebaert, *lpsolve : Open source (Mixed-Integer) Linear Programming system*, version 5.0.0.0, May 2004.

## A. Synchronous product of LTSs

We present the formal definition of synchronous product of two LTSs. We consider two LTSs $L_1 = (S_1, s_1, \Sigma, \Pi, \Delta_1)$ and $L_2 = (S_2, s_2, \Sigma, \Pi, \Delta_2)$. The *synchronous product* of L_1 and L_2 is an LTS $L = (S, s, \Sigma, \Pi', \Delta)$ such that:

1) $S \subseteq S_1 \times S_2$,
2) $s = (s_1, s_2)$,
3) $\Pi' = \Pi \times \Pi$, and
4) $\Delta \subseteq S \times \Sigma \times S \times \Pi'$ such that $((q_1, q_2), T, (q_1', q_2'), (P_1, P_2)) \in \Delta$ iff $(q_1, T, q_1', P_1) \in \Delta_1$ and $(q_2, T, q_2', P_2) \in \Delta_2$.

The set of states S is the smallest Δ-closed subset of $S_1 \times S_2$ that contains s (i.e., $s \in S$, and for each $q \in S$, if there exist $q' \in S_1 \times S_2$, $T \in \Sigma$ and $P \in \Pi'$ such that $(q, T, q', P) \in \Delta$, then $q' \in S$). That is, the synchronous product of L_1 with L_2 captures the joint behavior of L_1 and L_2 in every input sequence $\sigma \in \Sigma^\infty$ (L_1 and L_2 synchronize on input actions). Note that if both L_1 and L_2 are deterministic, so is there synchronous product. The synchronous product of $k > 2$ LTSs $L_1, \ldots, L_k$ is defined iteratively as the synchronous product of L_1 with the synchronous product of $L_2, \ldots, L_k$.

B. Examples of constraint LTSs

Figures 5, 6 and 7 show examples of constraint LTSs for a taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $C_1 = C_2 = 1$.
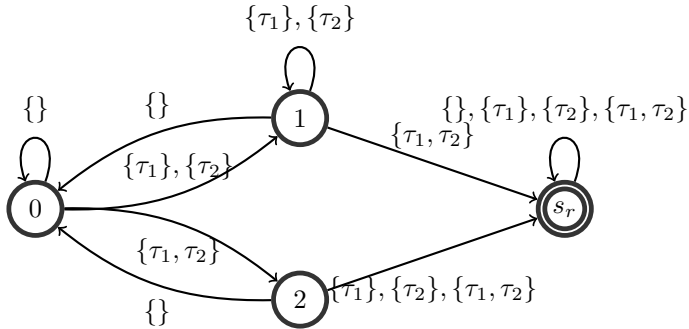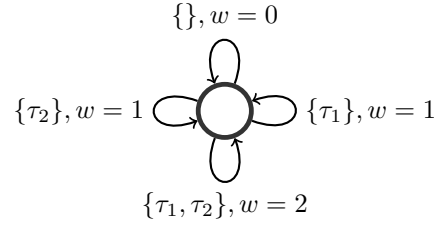
Fig. 5: Example of a safety LTS $L_\mathcal{S}$ that restricts the adversary to at most 2 units of workload in the last 2 rounds.
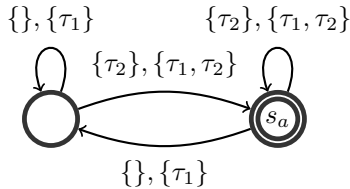
Fig. 6: Example of a liveness LTS $L_\mathcal{L}$ that forces τ_2 to be released infinitely often.

C. Adaptive Binary Search

Algorithm AdaptiveBinarySearch (Algorithm 2) implements an *adaptive* binary search for the competitive ratio in the

Fig. 7: Example of a limit-average LTS $L_\mathcal{W}$ that tracks the average workload of jobs released by the adversary.

interval $[0, 1]$. The algorithm maintains an interval $[\ell, r]$ such that $\ell \leqslant \mathcal{CR}_\mathcal{J}(\mathcal{A}) \leqslant r$ at all times, and exploits the nature of the problem for refining the interval as follows: First, if the current objective $\nu \in [\ell, r]$ (typically, $\nu = (\ell + r)/2$) is satisfied in G i.e., Lemma 1 answers "yes" and provides the current minimum cycle C as a witness, the value r is updated to the ratio ν' of the on-line and off-line rewards in C, which is typically less than ν. This allows to reduce the current interval for the next iteration from $[\ell, r]$ to $[\ell, \nu']$, with $\nu' \leqslant \nu$, rather than $[\ell, \nu]$ (as a simple binary search would do). Second, since $\mathcal{CR}_\mathcal{J}(\mathcal{A})$ corresponds to the ratio of rewards on a simple cycle in G, if the current objective $\nu \in [\ell, r]$ is not satisfied in G, the algorithm assumes that $\mathcal{CR}_\mathcal{J}(\mathcal{A}) = r$ (i.e, the competitive ratio equals the right endpoint of the current interval), and tries $\nu = r$ in the next iteration. Hence, as opposed to a naive binary search, the adaptive version has the advantages of (i) returning the exact value of $\mathcal{CR}_\mathcal{J}(\mathcal{A})$ (rather than an approximation), and (ii) being faster.

Algorithm 2: AdaptiveBinarySearch

Input: Graph $G = (V, E)$ and weight functions $w_\mathcal{A}$, $w_\mathcal{C}$
Output: $\min_{C \in G} \frac{w_\mathcal{A}(C)}{w_\mathcal{C}(C)}$

1   $\ell \leftarrow 0$, $r \leftarrow 1$, $\nu \leftarrow \frac{(\ell+r)}{2}$
2   **while** *True* **do**
3   Solve G for obj. Φ_ν and find min simple cycle C
4     $\nu_1 \leftarrow w_\mathcal{A}(C)$, $\nu_2 \leftarrow w_\mathcal{C}(C)$
5   **if** $\nu = \frac{\nu_1}{\nu_2}$ **then**
6   **return** ν
7   **else**
8   **if** $\nu > \frac{\nu_1}{\nu_2}$ **then**
9         $r \leftarrow \frac{\nu_1}{\nu_2}$, $\nu \leftarrow \frac{(\ell+r)}{2}$
10   **else**
11        $\ell \leftarrow \nu$, $r \leftarrow \min\left(\frac{\nu_1}{\nu_2}, r\right)$, $\nu \leftarrow r$
12   **end**
13   **end**
14 **end**

D. Tasksets used in the reported experiments

Table III lists the tasksets A1-A6 used for Figure 3, Table IV gives the tasksets used in the experiments reported in Figure 4 and Table I. In all cases, tasks are ordered by their static priorities, which determine the SP scheduler, as well as the

Name	C_i	D_i	V_i		Name	C_i	D_i	V_i
set A1	1	2	3		set A4	1	2	3
$k = 6$	4	6	2		$k = 3$	2	3	2
	1	3	3			1	6	1
	3	4	3		set A5	2	2	1
set A2	2	3	5		$k = 4$	6	6	10
$k = 5$	2	2	1			1	1	2
set A3	2	2	1		set A6	1	5	5
$k = 4$	1	5	2		$k = 5$	2	2	4
	1	5	2			1	1	1

Table III: Tasksets of Figure 3

C_i	D_i	V_i		C_i	D_i	V_i
1	1	3		2	7	3
1	2	3		5	5	2
1	1	1		5	6	1

Table IV: Taskset of Figure 4 (left) and Table I (right).

way ties are broken by other schedulers. In Table III, along with each taskset, its *importance ratio* k is shown, defined as

$$k = \max_{\tau_i, \tau_j \in \mathcal{T}} \frac{V_i / C_i}{V_j / C_j}$$

E. Competitive ratio of TD1

We have also considered the performance of the online scheduler TD1 in zero laxity tasksets with uniform value-density (i.e., for each task τ_i, we have $C_i = D_i = V_i$). Following [3], we have constructed a series of tasksets parameterized by some positive real $\eta < 4$, which guarantee that the competitive ratio of every online scheduler is upper bounded by $\frac{1}{\eta}$. Given η, each taskset consists of tasks τ_i such that C_i is given by the following recurrence, as long as $C_{i+1} > C_i$.

$$(i)\ C_0 = 1 \qquad (ii)\ C_{i+1} = \eta \cdot C_i - \sum_{j=0}^{i} C_j$$

In [3], TD1 was shown to have competitive factor $\frac{1}{4}$, and hence a competitive ratio that approaches $\frac{1}{4}$ from above, as $\eta \to 4$ in the above series of tasksets. Table V shows the competitive ratio of TD1 in this series of tasksets. Each taskset is represented as a set $\{C_i\}$, where each C_i is given by the above recurrence, rounded up to the first integer. We indeed see that the competitive ratio drops until it stabilizes to $\frac{1}{4}$.

Finally, note that the zero-laxity restriction allows us to process tasksets where $D_{\max}$ is much higher than what we report in Table II. The results of Table V were produced in less than a minute in total.

Name	η	Taskset	Comp. Ratio
set C1	2	$\{1, 1\}$	**1**
set C2	3	$\{1, 2, 3\}$	**1/2**
set C3	3.1	$\{1, 3, 7, 13, 19\}$	**7/25**
set C4	3.2	$\{1, 3, 7, 13, 20, 23\}$	**1/4**
set C5	3.3	$\{1, 3, 7, 14, 24, 33\}$	**1/4**
set C6	3.4	$\{1, 3, 7, 14, 24, 34\}$	**1/4**

Table V: Competitive ratio of TD1

F. Examples of on-line schedulers as LTSs

We now present more examples of on-line schedulers represented as deterministic LTSs. Consider the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 1$ already used for the EDF example in Figure 1. Figure 8 shows the EDF, SP, FIFO, and SRT policies represented as deterministic LTSs.
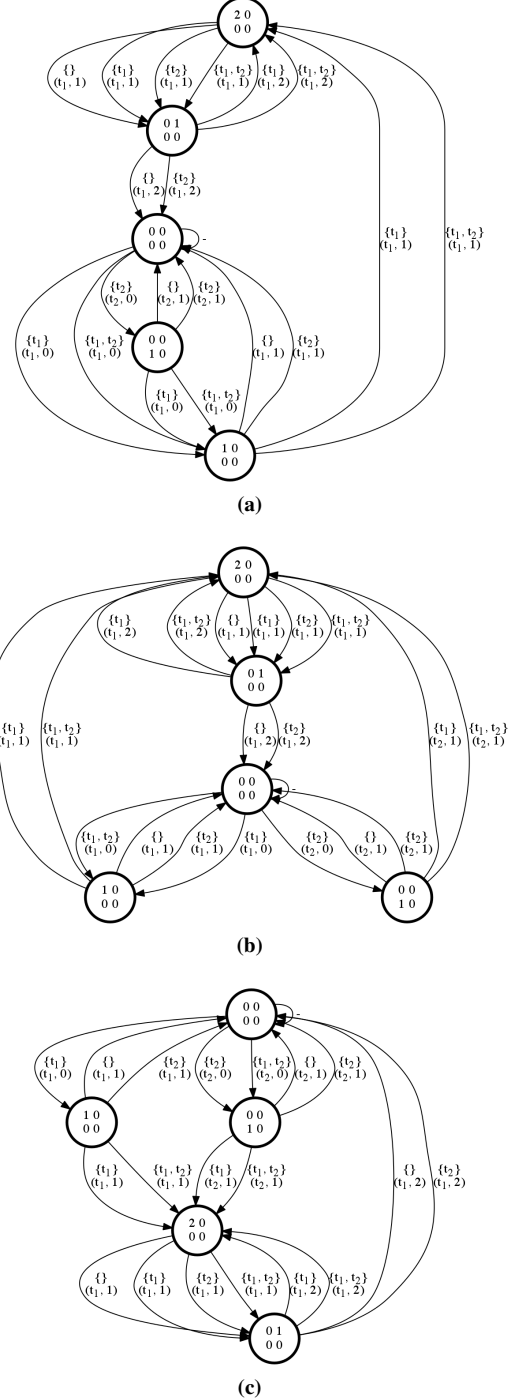
Fig. 8: The SP (a), FIFO (b) and SRT (c), on-line scheduling algorithms for the taskset $\mathcal{T} = \{\tau_1, \tau_2\}$ with $D_1 = 3$, $D_2 = 2$ and $C_1 = C_2 = 1$, represented as LTSs.