HAL
archives-ouvertes.fr

Scheduling Games for Concurrent Systems

Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab

▶ **To cite this version:**

Kasper Dokter, Sung-Shik Jongmans, Farhad Arbab. Scheduling Games for Concurrent Systems. 18th International Conference on Coordination Languages and Models (COORDINATION), Jun 2016, Heraklion, Greece. pp.84-100, 10.1007/978-3-319-39519-7_6 . hal-01631719

HAL Id: hal-01631719
https://hal.inria.fr/hal-01631719

Submitted on 9 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

Scheduling Games for Concurrent Systems

Kasper Dokter[1], Sung-Shik Jongmans[2,3], and Farhad Arbab[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, Netherlands
[2] Open University of the Netherlands, Heerlen, Netherlands
[3] Radboud University Nijmegen, Nijmegen, Netherlands

Abstract. A scheduler is an algorithm that assigns at any time a set of processes to a set of processors. Processes usually interact with each other, which introduces dependencies amongst them. Typically, such dependencies induce extra delays that the scheduler needs to avoid. Specific types of applications, like streaming applications, synthesize a scheduler from a formal model that is aware of these interactions. However, such interaction-specific information is not available for general types of applications. In this paper, we propose an interaction aware scheduling framework for generic concurrent applications. We formalize the amount of work performed by an application as constraints. We use these constraints to generate a graph, and view scheduler synthesis as solving a game on this graph that is played between the scheduler and the application. We illustrate that our framework is expressive enough to subsume an established scheduling framework for streaming programs.

1 Introduction

A scheduler of a concurrent application is an algorithm that assigns at any time *processes* of the application to a set of *processors* to execute them. The processes in a concurrent application interact with each other, which introduces dependencies amongst them. For example, a consumer process cannot execute if it requires data not yet provided by a producer process. Typically, such dependencies induce extra delays that the scheduler needs to avoid. For specific types of applications, like streaming applications [18], formal models exist that are aware of the interactions among their processes. Such models are then used to synthesize schedulers that optimize the execution of their applications with respect to a scheduling goal, such as latency or power consumption [4, 15]. For general types of applications, like web servers [10], no a priori detailed information about the interactions among their constituent processes is available to the scheduler. In such cases, a general-purpose round-robin scheduler is typically used to execute the application on the available processors. However, we cannot expect such schedulers to optimize our scheduling goals, because they cannot anticipate the dependencies among application processes.

In this paper, we propose an interaction-aware scheduling framework that enables scheduler synthesis for *generic* concurrent applications, by explicitly modelling interactions among processes. In particular, this framework consists of

two elements: a novel formal model of concurrent applications and a scheduler synthesis approach built on top of this formal model.

We base our formal model of concurrent applications on *constraint automata* [3], a general model of concurrency developed by Baier et al. (originally as a formal semantics for the coordination language Reo [2]). Basically, the idea is to model a concurrent application as a set of constraint automata, one for every process in the application. In this approach, every constraint automaton models the behaviour of a process at the level of its interactions with its environment (i.e., other processes). Using a special *composition operator*, we obtain a interaction-aware model for the entire concurrent application.

The existing theory of constraint automata focuses on processes and their interactions; it does not yet facilitate modelling the amount of work that processes need to carry out. However, such information is essential for scheduling. In this paper, we therefore extend transition labels in constraint automata with a declarative constraint that describes the work that needs to be done as part of a transition. These *job constraints* essentially generalize simple *weights* as in *weighted automata* [11], primarily to support true concurrency in composition. We call the resulting extension of constraint automata *work automata*, and we extend the composition operator on constraint automata to work automata accordingly. Work automata, then, constitute a formal model of concurrent applications in which both interaction among processes and work inside processes can be expressed, in a compositional *and* general manner.

Next, we use work automata in our interaction-aware scheduler synthesis. Given a formal model of a concurrent application as a set of work automata, our interaction-aware scheduler synthesis approach consists of two steps. In the first step, we use our composition operator on work automata to construct a work automaton for the entire concurrent application. The resulting work automaton models exactly the work of each process and the dependencies between the work. In the second step, we model the scheduler synthesis problem as a token *game* on a graph played between the scheduler and the application. The scheduler assigns the processes of the application to a heterogeneous set of processors, and the application non-deterministically selects a possible execution of the application. We apply results about the existence and quality of optimal strategies in *mean payoff games* [7,12] to find schedules that minimize the use of context-switches. Finally, we illustrate that our framework is expressive enough to subsume an established scheduling framework for streaming applications.

The structure of the paper is as follows: In Section 2, we introduce job constraints and define work automata. In Section 3, we define the graph on which a scheduling game is played. In Section 4, we apply our scheduling framework to streaming applications. In Section 5, we conclude and discuss future work.

2 Concurrent applications

As a starting point, we use a system of communicating automata to model interaction among processes in a concurrent application. To define the scheduling

problem for this system of automata, we annotate each transition with an expression that models the workload of that transition. In Section 2.1, we recall the definition of constraint automata. In Section 2.2, we introduce job constraints, which model the work of the processes in a concurrent application. In Section 2.3, we define work automata by adding job constraints to constraint automata. In Section 2.4, we informally discuss the semantics of work automata. In Section 2.5, we extend constraint automata composition to work automata.

2.1 Preliminaries on constraint automata

Baier et al. proposed constraint automata to model interaction amongst processes in a concurrent application [3]. A constraint automaton is a tuple $\mathcal{A} = (Q, \mathcal{P}, \to)$, where Q is a set of states, $\mathcal{P}$ is a set of ports, called the interface, and $\to \subseteq Q \times 2^{\mathcal{P}} \times Q$ is a transition relation. Informally, $\mathcal{A}$ is a labeled transition system with labels, called *synchronization constraints*, consisting of subsets $N \subseteq \mathcal{P}$. A synchronization constraint $N \subseteq \mathcal{P}$ describes the interaction of $\mathcal{A}$ with its environment: ports in N synchronize, while ports outside of N block. Note that $\emptyset \subseteq \mathcal{P}$ models an internal action of the automaton. Originally, in addition to a synchronization constraint, every transition in a constraint automaton carries also a *data constraint*. Data constraints are logical assertions that specify which particular data items may be observed on the ports that participate in a transition. Because data constraints do not matter in what follows—they address an orthogonal concern—we omit them from the definition for simplicity (technically, thus, we consider *port automata* [16]); the work presented in this paper straightforwardly extends to constraint automata with data constraints.

The constraint automaton of an entire application can be obtained by parallel composition of the constraint automata of its processes. For $i \in \{0, 1\}$, let $\mathcal{A}_i = (Q_i, \mathcal{P}_i, \to_i)$ be a constraint automaton. The composition $\mathcal{A}_0 \bowtie \mathcal{A}_1$ is defined by $(Q_0 \times Q_1, \mathcal{P}_0 \cup \mathcal{P}_1, \to)$, where $\to$ is the smallest relation that satisfies the following rule: if $i \in \{0, 1\}$, $\tau_i = (q_i, N_i, q_i') \in \to_i$, $\tau_{1-i} = (q_{1-i}, N_{1-i}, q_{1-i}') \in \to_{1-i} \cup \{(q, \emptyset, q) \mid q \in Q_{1-i}\}$ and $N_0 \cap \mathcal{P}_1 = N_1 \cap \mathcal{P}_0$, then $\tau_0 \mid \tau_1 = ((q_0, q_1), N_0 \cup N_1, (q_0', q_1')) \in \to$ (cf., Definition 3.2 in [3]). In other words, a transition $\tau = ((q_0, q_1), N, (q_0', q_1')) \in \to$ of the composition is possible if either (1) both restrictions $\tau|_{\mathcal{P}_0} = (q_0, N \cap \mathcal{P}_0, q_0')$ and $\tau|_{\mathcal{P}_1} = (q_1, N \cap \mathcal{P}_1, q_1')$ are transitions in $\mathcal{A}_0$ and $\mathcal{A}_1$, or (2) for some $i \in \{0, 1\}$, the restriction $\tau|_{\mathcal{P}_i}$ is a transition in $\mathcal{A}_i$ that is independent of $\mathcal{A}_{1-i}$, i.e., $N \cap \mathcal{P}_{1-i} = \emptyset$.

2.2 Job constraints

A system of constraint automata describes only interaction, while the workload of each process remains unspecified. Therefore, we extend transition labels in constraint automata with a *work expression* that models the amount of work that needs to be done before a transitions fires.

In the simplest of cases, a transition in a constraint automaton models an atomic piece of work, belonging to a single process. In that case, we can straightforwardly model this amount of work as a natural number $n \in \mathbb{N}_0$. However,

through (parallel) composition, a transition in a constraint automaton may also model the synchronous firing of *multiple* transitions (originating from different constraint automata for different processes). In that case, a single natural number fails to express that the work involved by each of these multiple transitions may actually be done in parallel. For instance, for $i \in \{0, 1\}$, let $\mathcal{A}_i = (Q_i, \mathcal{P}_i, \rightarrow_i)$ be a work automaton and $\tau_i = (q_i, N_i, q_i') \in \rightarrow_i$ a transition that requires $n_i \in \mathbb{N}_0$ units of work. Suppose that τ_0 and τ_1 synchronize, i.e., $N_0 \cap \mathcal{P}_1 = N_1 \cap \mathcal{P}_0$. Intuitively, $\tau_0 \mid \tau_1$ then requires $n_0 + n_1$ units of work, which may seem to define the composition of work. However, this composition loses the information that $\mathcal{A}_0$ and $\mathcal{A}_1$ may run in parallel and that the n_0 and n_1 units of work are independent of each other. To avoid this loss, we keep the values n_0 and n_1 separate by associating τ_i with a *job* x_i that requires n_i units of work. We represent the work of τ_i as the *job constraint* $x_i = n_i$, and the work of $\tau_0 \mid \tau_1$ as $x_0 = n_0 \wedge x_1 = n_1$.

Although job constraints with equalities (as introduced above) enable us to express parallelism of work between *synchronizing* transitions, they do not enable us to express parallelism of work between *independent* transitions (i.e., transitions that do not share any ports). The issue here is that if a transition τ_0 in automaton $\mathcal{A}_0$ fires *before* an independent transition τ_1 in automaton $\mathcal{A}_1$ fires, $\mathcal{A}_1$ is free to already perform (some) work *while* τ_0 fires, *in anticipation of* later firing τ_1. To model this, we should associate τ_0 with a job constraint that specifies that the work associated with τ_1 can be performed *partially*. We do this by allowing inequalities in job constraints. For instance, if the job constraint of τ_0 is $x_0 = n_0$, while the job constraint of τ_1 is $x_1 = n_1$, we define the job constraint of $\tau_0 \mid \epsilon$ (i.e., the incarnation of τ_0 in the composition of $\mathcal{A}_0$ and $\mathcal{A}_1$, where ϵ denotes an internal action of $\mathcal{A}_1$) as $x_0 = n_0 \wedge x_1 \leq n_1$.

We define the set of job constraints w over a set of jobs $\mathcal{J}$ by the grammar

$$w ::= \top \mid x = n \mid x \leq n \mid w_0 \wedge w_1, \tag{1}$$

with $x \in \mathcal{J}$ and $n \in \mathbb{N}_0$. The need for inequalities in w, precludes using weights on transitions in weighted automata [11] to represent work.

For notational convenience, we introduce the following terminology regarding a job constraint w over a set of jobs $\mathcal{J}$. Let $F, G \subseteq \mathcal{J}$ and $n_x, m_y \in \mathbb{N}_0$, for all $x \in F$ and $y \in G$, such that w is equivalent to $\bigwedge_{x \in F} x = n_x \wedge \bigwedge_{y \in G} y \leq m_y$. We call w *saturated*, whenever $F \cup G = \mathcal{J}$. We call w *satisfiable*, whenever $n_x \leq m_x$, for all $x \in F \cap G$. If w is satisfiable and $x \in \mathcal{J}$, then we define the *available work* $w_x \in \mathbb{N}_0 \cup \{\infty\}$ for job x by $w_x = n_x$, if $x \in F$, $w_x = m_x$, if $x \in G \setminus F$, and $w_x = \infty$ otherwise. Finally, we define the set of *required jobs* $\rho_w \subseteq \mathcal{J}$ by $\rho_w = F$.

2.3 Work automata

We now extend the transition labels of constraint automata from Section 2.1 with the job constraints from Section 2.2.

Definition 1. *A* work automaton *is a tuple* $(Q, \mathcal{P}, \mathcal{J}, \rightarrow)$ *that consists of a set of states* Q*, a set of ports* $\mathcal{P}$*, a set of jobs* $\mathcal{J}$*, and a transition relation* $\rightarrow \subseteq Q \times 2^{\mathcal{P}} \times \Omega_{\mathcal{J}} \times Q$*, where* $\Omega_{\mathcal{J}}$ *is the set of all satisfiable job constraints over* $\mathcal{J}$*.*

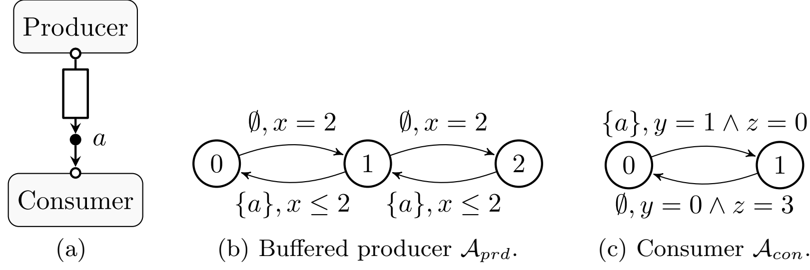


Fig. 1. Producer-consumer application (a), and its corresponding system of work automata (b) and (c) with $\{a\}$ as their interface $\{x\}$ and $\{y, z\}$ as their respective job sets.

Example 1. One of the simplest non-trivial examples of concurrent systems is the producer-consumer system, shown in Figure 1(a). The producer generates data and puts them into its buffer. The consumer takes these data from the buffer and processes them. We assume that the buffer has capacity 2. We split the system into a buffered producer and a consumer. Figures 1(b) and 1(c) show their respective work automata. States 0, 1 and 2 in Figure 1(b) indicate the amount of data in the buffer. In state 0 or 1, the producer can produce a new datum by finishing 2 units of work of job x. In state 1 or 2, the consumer can take a datum from the buffer by synchronizing on port a, which requires no work on job x. In state 0 in Figure 1(c), the consumer waits for a datum d at port a. When d arrives, the consumer takes it from the buffer, requiring 1 unit of work on job y. In state 1, the consumer processes datum d, requiring 3 units on job z.

2.4 Job execution

Let $\mathcal{A} = (Q, \mathcal{P}, \mathcal{J}, \rightarrow)$ be some fixed work automaton. In this section, we informally introduce the semantics of $\mathcal{A}$. The jobs in a work automaton are executed by a *parallel machine* $\mathcal{M}$, which consists of a heterogeneous set of processors and a map that represents the execution speed of jobs on processors.

Definition 2. *A* parallel machine *is a tuple* $(M, \mathcal{J}, v)$*, where* M *is a set of processors,* $\mathcal{J}$ *is a set of jobs and* $v : \mathcal{J} \times M \rightarrow \mathbb{N}_0$ *is a map that models the speeds of jobs on processors.*

It is the task of a scheduler to assign jobs from a set $\mathcal{J}$ to processors in a parallel machine $(M, \mathcal{J}, v)$ over $\mathcal{J}$. We model this assignment of jobs to processors by an injective partial map $s : M \rightharpoonup \mathcal{J}$ that represents the *scheduled jobs*, i.e., $s(i) = s(j)$ implies $i = j$, for all $i, j \in M$. We write $S(M, \mathcal{J})$ for the set of all injective partial maps $s : M \rightharpoonup \mathcal{J}$.

We represent the *speeds of jobs* in $\mathcal{J}$ subject to the scheduled jobs $s \in S(M, \mathcal{J})$ by the map $v_s : \mathcal{J} \rightarrow \mathbb{N}_0$, given by $v_s(x) = v(x, s^{-1}(x))$ if $x \in \mathrm{im}(s)$ and $v_s(x) = 0$ otherwise. Here, $\mathrm{im}(s) = \{s(m) \in \mathcal{J} \mid m \in M\}$ is the image of s.

We represent the current *progress* of jobs by a map $p : \mathcal{J} \to \mathbb{Q}_{\geq 0}$, where $\mathbb{Q}_{\geq 0}$ is the set of non-negative rational numbers. After executing the scheduled jobs $s \in S(M, \mathcal{J})$ for $t \in \mathbb{Q}_{\geq 0}$ time, the progress of jobs in $\mathcal{J}$ equals $p' = p + v_s \cdot t$, where $+$ is pointwise addition and $\cdot$ is multiplication by a scalar, i.e., $p'(x) = p(x) + v(x, s^{-1}(x)) \cdot t$ if $x \in \text{im}(s)$ and $p'(x) = p(x)$ otherwise.

Example 2. Let $k > 0$ be a positive integer and $\mathcal{J}$ a set of jobs. Then, $\mathcal{M}_k = (\{1, \ldots, k\}, \mathcal{J}, v)$, with $v(x, i) = 1$ for all $x \in \mathcal{J}$ and $1 \leq i \leq k$, models a parallel machine that consists of k identical processors. Any two processors are identical and interchangeable. Therefore, the scheduled jobs $s \in S(M, \mathcal{J})$ depend solely on the image $\text{im}(s)$. If $s, s' \in S(\{1, \ldots, k\}, \mathcal{J})$ and $\text{im}(s) = \text{im}(s')$, then $v_s = v_{s'}$. Hence, we represent scheduled jobs as a subset $J \subseteq \mathcal{J}$.

Let $\tau = (q, N, w, q')$ be a transition in $\mathcal{A}$ and $p : \mathcal{J} \to \mathbb{Q}_{\geq 0}$ be the current progress of jobs. Recall the notation regarding job constraints from Section 2.2. We call a job x *finished* whenever its progress $p(x)$ equals $w_x \in \mathbb{N}_0 \cup \{\infty\}$. We demand that all required jobs $x \in \rho_w$ finish their available work w_x. The automaton $\mathcal{A}$ may take a transition τ if the progress of jobs p satisfies the job constraint w (notation: $p \models w$), i.e., $p(x) \leq w_x$ for all jobs $x \in \mathcal{J}$ and $p(x) = w_x$ for required jobs $x \in \rho_w$. Note that for $\rho_w = \emptyset$, transition τ requires no work, and τ then represents for example the arrival of input data.

Suppose that $p \models w$ and $\mathcal{A}$ takes transition τ. Then, the current state of $\mathcal{A}$ becomes q' and the progress of required jobs resets to zero. Formally, the progress becomes $p' = \overline{\rho_w}(p)$, where $\overline{\rho_w} : \mathbb{N}_0^{\mathcal{J}} \to \mathbb{N}_0^{\mathcal{J}}$ is the *reset operation* associated with ρ_w defined as $\overline{\rho_w}(p)(x) = p(x)$ if $x \notin \rho_w$ and $\overline{\rho_w}(p)(x) = 0$ otherwise.

2.5 Composition

In Section 2.3, we extended constraint automata to work automata. We now extend the composition of constraint automata from Section 2.1 to work automata.

Let $\mathcal{A}_0$ and $\mathcal{A}_1$ be two work automata. We want our composition of work automata to conservatively extend the composition of constraint automata. This means that the state space, interface and transition relation (up to job constraints) of the composition are already determined. Since a job x in $\mathcal{A}_i$, for $i \in \{0, 1\}$, is merely a name for a piece of work inside $\mathcal{A}_i$, we may rename x to (x, i). This allows us to define the set of jobs of the composition as the disjoint union $\mathcal{J}_0 + \mathcal{J}_1 = \mathcal{J}_0 \times \{0\} \cup \mathcal{J}_1 \times \{1\}$. For $i \in \{0, 1\}$, let $\tau_i = (q_i, N_i, w_i, q_i')$ be a transition in $\mathcal{A}_i$. If $N_0 \cap \mathcal{P}_1 = N_1 \cap \mathcal{P}_0$, then τ_0 and τ_1 synchronize and give rise to a transition $\tau_0 \mid \tau_1 = ((q_0, q_1), N_0 \cup N_1, w_0 \wedge w_1, (q_0', q_1'))$. If τ_0 and τ_1 are independent, i.e., $N_0 \cap \mathcal{P}_1 = N_1 \cap \mathcal{P}_0 = \emptyset$, then τ_0 and the relaxation $(q_1, \emptyset, w_1^{\leq}, q_1)$ of τ_1, give rise to a transition $\tau_0 \mid \tau_1^{\leq} = ((q_0, q_1), N_0, w_0 \wedge w_1^{\leq}, (q_0', q_1))$ in the composition, where $w_1^{\leq}$ is the job constraint derived from w_1 by substituting every $=$ with $\leq$. This substitution is well-defined, because, according to grammar (1), jobs exclusively appear on the left hand side of an equality. Transition $\tau_0 \mid \tau_1^{\leq}$ represents that τ_0 is taken, while jobs in τ_1 makes arbitrary progress bounded by w_1. We define the a lift $\tau_0^{\leq} \mid \tau_1$ of τ_1 analogously. Finally, if τ_0

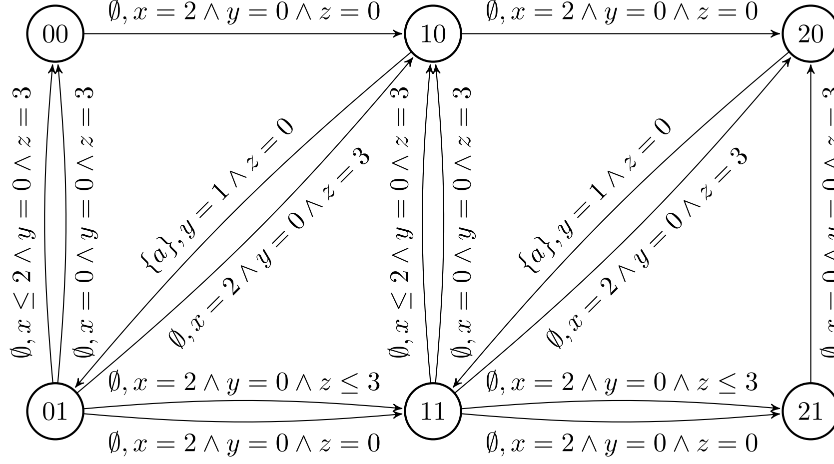


Fig. 2. Composition $\mathcal{A}_{prd} \bowtie \mathcal{A}_{con}$ of the work automata in Figures 1(b) and 1(c).

is independent of $\mathcal{A}_1$ (i.e., $N_0 \cap \mathcal{P}_1 = \emptyset$), then τ_0 gives rise to a transition $((q_0, q_1), N_0, w_0 \wedge \bigwedge_{x \in \mathcal{J}_1} x = 0, (q_0', q_1))$ in the composition, where τ_0 executes independently of $\mathcal{A}_1$ and all jobs in $\mathcal{A}_1$ block. This blocking means that $\mathcal{A}_1$ needs to wait, unless a transition τ_1 in $\mathcal{A}_1$ induces a synchronization $\tau_0 \mid \tau_1$ or $\tau_0 \mid \tau_1^{\leq}$.

Definition 3. *For $i \in \{0, 1\}$, let $\mathcal{A}_i = (Q_i, \mathcal{P}_i, \mathcal{J}_i, \rightarrow_i)$ be a work automaton. We define the* composition $\mathcal{A}_0 \bowtie \mathcal{A}_1$ *of $\mathcal{A}_0$ and $\mathcal{A}_1$ as the work automaton $(Q_0 \times Q_1, \mathcal{P}_0 \cup \mathcal{P}_1, \mathcal{J}_0 + \mathcal{J}_1, \rightarrow)$, where $\rightarrow$ is the smallest relation satisfying the following rule: if $i \in \{0, 1\}$, $\tau_i = (q_i, N_i, w_i, q_i') \in \rightarrow_i$, $\tau_{1-i} = (q_{1-i}, N_{1-i}, w_{1-i}, q_{1-i}') \in \rightarrow_{1-i} \cup \{(q, \emptyset, \bigwedge_{x \in \mathcal{J}_{1-i}} x = 0, q) \mid q \in Q_{1-i}\}$ and $I := N_0 \cap \mathcal{P}_1 = N_1 \cap \mathcal{P}_0$, then*

1. *$\tau_0 \mid \tau_1 = ((q_0, q_1), N_0 \cup N_1, w_0 \wedge w_1, (q_0', q_1')) \in \rightarrow$; and*
2. *$I = \emptyset$ implies $\tau_0 \mid \tau_1^{\leq} \in \rightarrow$ and $\tau_0^{\leq} \mid \tau_1 \in \rightarrow$, where $\tau_i^{\leq} = (q_i, \emptyset, w_i^{\leq}, q_i)$.*

Example 3. Figure 2 shows the composition of the work automata from Figures 1(b) and 1(c). A state ij indicates that the buffered producer is in state i and the consumer is in state j. In state 00, the consumer cannot retrieve a datum from the buffer. Hence, the consumer is not allowed to work on job y. The transition from 01 to 11 with job constraint $x = 2 \wedge y = 0 \wedge z = 3$ is redundant, because the other transition from 01 to 11 has a weaker job constraint $x = 2 \wedge y = 0 \wedge z \leq 3$.

3 Scheduling games

A work automaton can make non-deterministic internal choices, beyond the control of the scheduler. Therefore, we can view the scheduler synthesis problem over a work automaton and a parallel machine as a game that is played between the scheduler and the application modelled by a work automaton. The scheduler

assigns jobs to processors and the application executes the running jobs and, whenever possible, makes a perhaps non-deterministically selected transition. We represent this game as a token game played on a graph that we derive from a work automaton and a parallel machine. Every play of this game (i.e., a path in this graph) corresponds to a run of the work automaton. Hence, a strategy in this game corresponds to a schedule of the corresponding concurrent application. In Section 3.1, we recall some basic terminology and known results for games played on graphs. In Section 3.2, we interpret the execution of jobs in a work automaton as a game played on a graph. In Section 3.3, we assign an execution time to every move in a scheduling game. In Section 3.4, we introduce a class of scheduling objectives for both terminating and non-terminating applications. In Section 3.5, we find schedules that minimize the number of context-switches.

3.1 Preliminaries on games on graphs

We view scheduler synthesis as a problem of finding optimal strategies in a game played on a graph. Therefore, we recall the basic definitions about these games.

A *game arena* is a finite directed bipartite leafless graph A. More formally, A is a triple (V, E, φ) that consists of a finite set of vertices V, a set of edges $\to \subseteq V \times V$ such that for all $a \in V$ there exists a $b \in V$ with $(a, b) \in E$, and a 2-colouring $\varphi : V \to \{0, 1\}$, i.e., $(a, b) \in E$ implies $\varphi(a) \neq \varphi(b)$, for all $a, b \in V$. Vertices and edges in this graph are called *positions* and *moves*. For every $a_0 \in V$, consider the following token game on A between Player 0 and Player 1. Let a_0 be the initial position of the token. Construct an infinite sequence $\pi = (a_i)_{i=0}^{\infty}$ as follows: for all $i \geq 0$, Player $\varphi(a_i)$ selects a successor position $a_{i+1} \in V$, with $(a_i, a_{i+1}) \in E$, and moves the token from a_i to a_{i+1}. The sequence π is called a *play* of this game, and $\text{plays}(A) \subseteq V^\omega$ is the set of all such plays in A. A *game* G is a triple (A, a_0, f), where $A = (V, E, \varphi)$ is a game arena, $a_0 \in V$ is the initial position, and $f : \text{plays}(A) \to D$ is a *payoff* function, where D is some partially ordered set. The goal of Player 0 is to maximize the value $f(\pi)$, while Player 1 tries to minimize $f(\pi)$. A *strategy* σ_k for Player $k \in \{0, 1\}$ in a game G is a map $\sigma_k : V^* \times V_k \to V_{1-k}$, such that $(v, \sigma_k(u, a)) \in E$ for all $u \in V^*$ and $a \in V_k$. Intuitively, a strategy σ_k determines the successor position $\sigma_k(u, a) \in V_{1-k}$ of Player k, based on the history u and the current position a. A strategy σ is called *memoryless* if and only if $\sigma(u, a) = \sigma(u', a)$, for all $u, u' \in V^*$ and $a \in V$. A play $\pi = a_0 a_1 \cdots$ is *consistent* with a strategy σ_k for Player k if and only if for all $i \geq 0$ we have that $\varphi(a_i) = k$ implies $a_{i+1} = \sigma_k(a_0 \cdots a_{i-1}, a_i)$.

Example 4 (Mean payoff games [12]). Let $A = (V, E, \varphi)$ be an arena, and let $c : E \to \mathbb{Z}$ be a *weight function*. In Section 3.3, we use these weights to represent the execution time of moves. A *mean payoff game* over A and c is a triple $G = (A, a_0, M_c)$, where $a_0 \in V$ is the starting position, and

$$M_c(a_0 a_1 a_2 \cdots) = \liminf_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n-1} c(a_i, a_{i+1}).$$

Intuitively, M_c computes the 'smallest' average value of the play $a_0 a_1 a_2 \cdots$.

3.2 Scheduling arena

We now formulate the problem of scheduling a concurrent application, represented as a work automaton $\mathcal{A} = (Q, \mathcal{P}, \mathcal{J}, \rightarrow)$, onto a set of heterogeneous processors, represented as a parallel machine $\mathcal{M} = (M, \mathcal{J}, v)$. The scheduling problem consists of finding an optimal strategy in a game on a graph played by the scheduler (Player 0) and the application (Player 1). Intuitively, the game is played by alternating moves by the scheduler and the application. A scheduler move selects a schedule $s \in S(M, \mathcal{J})$. Recall the notation for job constraints from Section 2.2. An application move selects a transition $\tau = (q, N, w, q')$ that allows scheduled jobs to progress, and then updates the progress $p : \mathcal{J} \rightarrow \mathbb{Q}_{\geq 0}$ of the jobs by executing the scheduled jobs s until one of the jobs $x \in \mathcal{J}$ finishes w_x units of work. If after the execution the job constraint w is satisfied, the application makes transition τ. Otherwise, the application makes the 'fictitious' idling transition $\epsilon_q := (q, \emptyset, \top, q)$, where $q \in Q$ is the current state of the automaton.

We now explain the construction of the game arena in more detail. We want every play of this game to correspond to an run of the associated work automaton. Therefore, we record, in every position of the game, the progress of the jobs and the state of the automaton. We define the positions of the scheduler as pairs (p, τ), where $p : \mathcal{J} \rightarrow \mathbb{Q}_{\geq 0}$ is the progress of jobs and $\tau = (q, N, w, q') \in \rightarrow \cup \{\epsilon_q \mid q \in Q\}$ is the transition that is previously taken by the application (i.e., q' is the current state of the work automaton). We define the positions of the application as triples $[p, q, s]$, where $p : \mathcal{J} \rightarrow \mathbb{Q}_{\geq 0}$ is the progress of jobs, $q \in Q$ is the current state of the work automaton and $s \in S(M, \mathcal{J})$ is the set of the scheduled jobs that are selected by the scheduler.

In a position (p, τ), the scheduler may select any assignment $s \in S(M, \mathcal{J})$ of jobs to processors, which corresponds to selecting a successor position $[p, q', s]$. For the definition of application moves, we first define *enabled* transitions. Intuitively, a transition is *enabled* in position $[p_b, q_b, s]$ if its source state is q_b, its job constraint is *potentially* satisfiable (i.e., $p_b(x) \leq w_x$, for all $x \in \mathcal{J}$) and all scheduled jobs s can execute (i.e., $v_s(x) > 0$ implies $p_b(x) < w_x$, for all $x \in \mathcal{J}$).

Definition 4. *We call a transition $\tau = (q, N, w, q')$* enabled *at a position $b = [p_b, q_b, s]$ of the application if and only if for all $x \in \mathcal{J}$, we have that $q = q_b$, $p_b(x) \leq w_x$, and $v_s(x) > 0$ implies $p_b(x) < w_x$. We write $\mathcal{E}_b \subseteq \rightarrow$ for the set of all transitions that are enabled at b.*

If there is no enabled transition, then the application selects the successor position (p, ϵ_q). Otherwise, the application selects any enabled transition $\lambda = (q, N, w, q') \in \mathcal{E}_b$ and executes its scheduled jobs, until one of them finishes.

Definition 5. *The* time to first completion $t_b(\lambda)$ *of an enabled transition $\lambda \in \mathcal{E}_b$ at a position $b = [p, q, s]$ is*

$$t_b(\lambda) = \begin{cases} \min \; T_b(\lambda) & \text{if } T_b(\lambda) \neq \emptyset \\ 0 & \text{otherwise} \end{cases},$$

where $T_b(\lambda) = \{t \in \mathbb{Q}_{\geq 0} \mid \exists x \in \mathcal{J} : v_s(x) > 0$ and $p(x) + v_s(x) \cdot t = w_x\}$.

After executing the jobs for $t_b(\lambda)$ units of time, the progress of the jobs becomes $p+v_s \cdot t_b(\lambda)$, which is defined as $(p+v_s \cdot t_b(\lambda))(x) = p(x)+v_s(x) \cdot t_b(\lambda)$, for all $x \in \mathcal{J}$. If the job constraint of λ is satisfied ($p + v_s \cdot t_b(\lambda) \models w$), the application makes transition λ by selecting position $(\overline{\rho_w}(p+v_s \cdot t_b(\lambda)), \lambda)$, where $\overline{\rho_w}$ resets the progress of all finished jobs ρ_w to zero. If the job constraint of λ is not satisfied ($p+v_s \cdot t_b(\lambda) \not\models w$), the application selects position $(p+v_s \cdot t_b(\lambda), \epsilon_q)$.

Definition 6. *A* scheduling arena *A over a work automaton $(Q, \mathcal{P}, \mathcal{J}, \rightarrow)$ and a parallel machine $(M, \mathcal{J}, v)$ is a tuple $A = (V, E, \varphi)$, where $V = V_0 \cup V_1$,*

$$V_0 = \{(p, \tau) \mid p : \mathcal{J} \to \mathbb{Q}_{\geq 0} \text{ and } \tau \in \rightarrow \cup \{\epsilon_q \mid q \in Q\}\},$$
$$V_1 = \{[p, q, s] \mid p : \mathcal{J} \to \mathbb{Q}_{\geq 0},\ q \in Q \text{ and } s \in S(M, \mathcal{J})\}$$

are the sets of positions of the scheduler and the application, $\varphi(a) = 0$ if and only if $a \in V_0$, and $E \subseteq V \times V$ is the largest relation that satisfies the following rule: for all $a = (p, \tau) \in V_0$ and $b = [p, q, s] \in V_1$ we have

1. *if $\tau = (-, -, -, q'_\tau)$, then $(a, [p, q'_\tau, s]) \in E$; and*
2. *if $\mathcal{E}_b = \emptyset$, then $(b, (p, \epsilon_q)) \in E$; and*
3. *if $\lambda = (q, N, w, q') \in \mathcal{E}_b$, then*
   (a) $p + v_s \cdot t_a(\lambda) \models w$ implies $(b, (\overline{\rho_w}(p + v_s \cdot t_a(\lambda)), \lambda)) \in E$; and
   (b) $p + v_s \cdot t_a(\lambda) \not\models w$ implies $(b, (p + v_s \cdot t_a(\lambda), \epsilon_q)) \in E$.

As a scheduling arena A is infinite, it is not an arena as in Section 3.1. The following lemma provides a sufficient condition ensuring that A is *locally finite*, i.e., only finitely many positions in the A are reachable from any given position.

Lemma 1. *Let A be a scheduling arena over a work automaton $\mathcal{A}$ and a parallel machine $(M, \mathcal{J}, v)$. If $\mathcal{A}$ has finitely many transitions, all job constraints in $\mathcal{A}$ are saturated and all speeds $v(x, i)$ are either zero or u, for some $u \in \mathbb{N}_0$, then only finitely many positions in A are reachable from any given position $a \in A$.*

Proof. For every job $x \in \mathcal{J}$, define $m_x = \max\{w_x \mid w \text{ is a job constraint in } \mathcal{A}\}$. Since all job constraints are saturated, we have that $|\mathcal{J}| < \infty$ and $m_x < \infty$, for all $x \in \mathcal{J}$. Hence, we find $\alpha \in \mathbb{N}_0$ such that for every job $x \in \mathcal{J}$ the progress $p_a(x)$ of x at a satisfies $\alpha p_a(x) \in \mathbb{N}_0$. Using Definitions 4, 5 and 6, it follows that for every job $x \in \mathcal{J}$ the progress $p_b(x)$ of x at a position $b \in A$ reachable from a satisfies $p_b(x) \in \{p_a(x), 0, \frac{1}{\alpha}, \ldots, \frac{\alpha m_x - 1}{\alpha}, m_x\}$, for all $x \in \mathcal{J}$. We conclude that only finitely many positions in A are reachable from any given position $a \in A$.

Example 5. Consider the work automaton $\mathcal{A} = (\{0\}, \emptyset, \{x, y\}, \{\lambda, \tau, \mu\})$ and parallel machine $\mathcal{M}_2$, where $\lambda = (0, \emptyset, x = 1 \wedge y \leq 1, 0)$, $\tau = (0, \emptyset, x = 5 \wedge y = 0, 0)$ and $\mu = (0, \emptyset, x = 0 \wedge y = 1, 0)$. Figure 3 shows the scheduling arena over $\mathcal{A}$ and $\mathcal{M}_2$ from Example 2 according to Definition 6. A circular position labelled by $k\alpha$, with $k \in \{0, 1\}$ and $\alpha \in \{\epsilon_0, \lambda, \tau, \mu\}$, corresponds to $(p, \alpha) \in V_0$, with $p(x) = 0$ and $p(y) = k$. For $k \in \{0, 1\}$, a square position labeled by k, kx, ky or kxy corresponds to a position $(p, 0, s) \in V_1$ with $p(x) = 0$, $p(y) = k$ and $\text{im}(s) = \emptyset$, $\text{im}(s) = \{x\}$, $\text{im}(s) = \{y\}$ or $\text{im}(s) = \{x, y\}$, respectively.

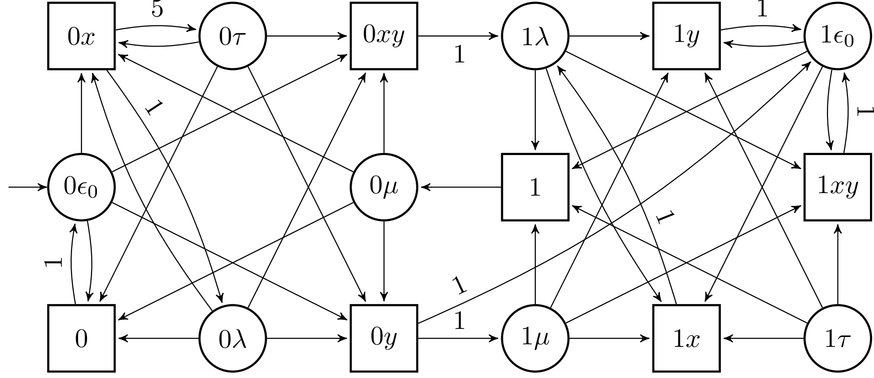


Fig. 3. Scheduling arena over a work automaton $\mathcal{A}$ from Example 5. Circular positions belong to the scheduler position and square positions belong to the application.

3.3 Strategies and classical schedules

From a given work automaton $\mathcal{A}$ and parallel machine $\mathcal{M}$, we constructed in Section 3.2 a scheduling arena A. Suppose that the non-deterministic behaviour of $\mathcal{A}$ is fully controlled by the scheduler, i.e., there is only one move possible at every position $a \in V_1$ of the application. We now argue that strategies in A naturally correspond to classical schedules of concurrent applications.

Since the application has a unique strategy, every strategy σ_0 of the scheduler induces a play π in A. The following definition assigns an execution time to every move in π, which allows us to represent π as a Gantt chart [13].

Definition 7. *The* execution time $t(a,b)$ *of a move* $(a,b) \in E$ *in a scheduling arena* $A = (V, E, \varphi)$ *is*

$$t(a,b) = \begin{cases} t_a(\lambda) & \text{if } (a,b) \in V_1 \times V_0 \text{ comes from } \lambda \in \mathcal{E}_a \\ 1 & \text{if } a = [p,q,s] \in V_1 \text{ and } \mathcal{E}_{[p,q,s']} \neq \emptyset = \mathcal{E}_a \text{ for some } s' \\ 0 & \text{otherwise} \end{cases} .$$

The case for $t(a,b) = 1$ can be seen as a time penalty for selecting $s \in S(M, \mathcal{J})$ that unnecessarily blocks the execution ($\mathcal{E}_{[p,q,s]} = \emptyset$).

Example 6. In the scheduling arena in Figure 3, consider the play π that is given by $0\epsilon_0, 0x, 0\tau, 0xy, 1\lambda, 1, 0\mu, 0y, 1\mu, 1, 0\mu, 0y, 1\mu, 1y, 1\epsilon_0, 1y, \ldots$. All zeros on the move labels are omitted in this arena. Figure 4(a) shows a Gantt chart representation of π. Note that, since x and y are executed on identical processors $\mathcal{M}_2$, it is not important on which processor x and y are scheduled.

We conclude that every strategy in A naturally induces a classical schedule of the concurrent application. Conversely, not every classical schedule comes from a strategy in such an arena A. According to Definition 6, scheduling strategies

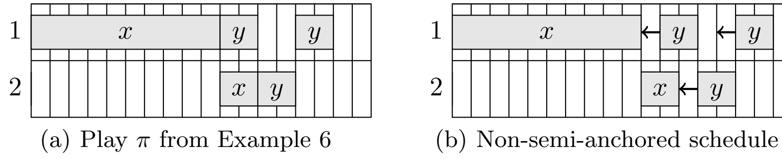


Fig. 4. Play π from Example 6 (a), and a schedule that is not semi-anchored (b).

induce only *semi-anchored* schedules, i.e., a job can start at time $t + n$, with $t \in \mathbb{Q}_{\geq 0}$ and $1 \leq n \in \mathbb{N}$, only if $t = 0$ or t is a time that some job finishes, and all processors are idle between t and $t + n$. Figure 4(b) shows a schedule that cannot be represented by a strategy in A. However, shifting the executions of all jobs y to the left transforms Figure 4(b) into an anchored schedule.

We now show that this shifting always produces a valid schedule for $\mathcal{A}$. Let $\mathcal{S}$ be a (non-semi-anchored) classical schedule, and $T \subseteq \mathbb{Q}_{\geq 0}$ be the set of all finish times of jobs in $\mathcal{S}$ including zero. Let t_s be the start time of a job x with $t_s \notin T$, and $t_f = \max\,\{t \in T \mid t \leq t_s\}$ the last time a job in $\mathcal{S}$ finishes before t_s. Every transition taken by $\mathcal{A}$ after t_f was already enabled at time t_f. Thus, shifting the execution of job x from t_s to t_f produces a valid schedule.

We call a scheduling objective *regular*, whenever this shifting operation produces a schedule that is at least as good as the initial schedule. For example, minimizing total execution time is a regular scheduling objective, while scheduling objectives that penalize for jobs that finish 'too early' are not regular.

3.4 Scheduling games

In this section, we define payoff functions for games played on scheduling graphs that naturally correspond to regular scheduling objectives.

Let $\pi = a_0 a_1 \cdots$ be a play in a scheduling arena $A = (V, E, \varphi)$. Using Definition 7, we associate with every initial prefix $\pi_n = a_0 \cdots a_n$, $n \geq 0$, the total execution time $t_n = \sum_{i=0}^{n-1} t(a_i, a_{i+1})$. If our application terminates, then for every play $\pi = a_0 a_1 \cdots$, the sequence $t_0, t_1, \ldots$ eventually stabilizes, i.e., $t_n = t_m$, for some n and all $m \geq n$. Then, t_n represents the total execution time of π. If our application does not always terminate, then we cannot associate with every play π its total execution time. An example of such an application is a streaming application (cf., Section 4). A natural scheduling objective in a streaming application is latency minimization at some output port $o \in \mathcal{P}$. We define the latency at a port o as the average time between two successive I/O operations on o. To keep track of these I/O operations, we use a map $\theta_o : E \to \{0, 1\}$, such that $\theta_o(a, b) = 1$ if and only if $b = (p, \tau) \in V_0$, where $\tau = (q, N, w, q')$ and $o \in N$. For a prefix $\pi_n = a_0 \cdots a_n$, $n \geq 0$, of π, we define the *latency* as the ratio between the total execution time t_n and the number of I/O operations $1 + \sum_{i=0}^{n-1} \theta_o(a_i, a_{i+1})$.

By varying $\theta : E \to \{0, 1\}$, we define the following class of scheduling games, called *latency games*, wherein Player k maximizes the 'smallest limiting ratio'. Recall the definition of locally finite scheduling arena's from Section 3.2.

Definition 8. *Let $A = (V, E, \varphi)$ be a locally finite scheduling arena, $\theta : E \to \{0, 1\}$ a map, and $k \in \{0, 1\}$. A* latency game *G for Player k over A and θ is a tuple $G = (A, a_0, T^k_\theta)$, where a_0 is an initial position and*

$$T^k_\theta(a_0a_1\cdots) = \liminf_{n\to\infty} \frac{(-1)^k}{1 + \sum_{i=0}^{n-1}\theta(a_i, a_{i+1})} \sum_{i=0}^{n-1} t(a_i, a_{i+1}), \qquad (2)$$

where $t : E \to \mathbb{N}_0$ is the execution time from Definition 7.

Example 7 (Makespan games: T^1_0). Let θ be the map $0 : E \to \{0, 1\}$, given by $0(a, b) = 0$, for all $(a, b) \in E$, and $k = 1$. The scheduling objective in the latency game over θ and k is given by $T^1_0 = \liminf_{n\to\infty} -\sum_{i=0}^{n-1} t(a_i, a_{i+1})$. Recall from Section 3.1 that Player 0 wants to maximize T^1_0, which corresponds to minimizing the total execution time $-T^1_0 = \limsup_{n\to\infty} \sum_{i=0}^{n-1} t(a_i, a_{i+1})$.

Example 8 (Context-switches: T^0_1). Due to changes in the assignment of jobs to processors, context-switches may occur. Typically, context-switches inflict substantial overhead and their occurrences should be avoided. This scheduling objective can be seen as a latency game, where $k = 0$ and θ is the map $1 : E \to \{0, 1\}$, given by $1(a, b) = 1$, for all $(a, b) \in E$. Then, the scheduling objective becomes $T^0_1 = \liminf_{n\to\infty} \frac{1}{n+1} \sum_{i=0}^{n-1} t(a_i, a_{i+1})$, which can be interpreted as maximizing the average time between two consecutive context-switches. Indeed, every move by the application executes all scheduled jobs until at least one of them finishes. The job that finishes should subsequently be descheduled (context-switch), to avoid suboptimal use of compute resources (i.e., idling).

Note that $\lim_{n\to\infty} \frac{n+1}{n} = 1$ implies that the scheduling objective T^0_1 coincides with the payoff function of the mean payoff games in Example 4.

Example 9 (Latency at o: $T^1_{\theta_o}$). Let $\mathcal{A}$ be a work automaton with a port $o \in \mathcal{P}$, and let $A = (V, E, \varphi)$ be a scheduling arena over $\mathcal{A}$ and some parallel machine. Using Definition 6, we can identify the moves in the scheduling arena that come from a transition that requires an I/O operation on port o. Thus, let $\theta_o : E \to \{0, 1\}$ be given by $\theta_o(a, b) = 1$ if and only if $b = (p, \tau) \in V_0$, where $\tau = (q, N, w, q')$ and $o \in N$. The scheduling objective $T^1_{\theta_o}$ corresponds to maximizing the production rate at port o.

3.5 Optimal strategies

In Section 3.4, we viewed the scheduling problem as a game played on a graph. We now take advantage of the fact that these games have been extensively studied in the literature. To do this, we need some terminology about games on graphs. Let G be a game over an arena A, with initial position $a_0 \in A$, and payoff function $f : \text{plays}(A) \to D$, for some partially ordered set D of values. A strategy σ_k for Player $k \in \{0, 1\}$ *secures* a value $v \in D$ whenever $(-1)^k f(\pi) \geq (-1)^k v$, for every play $\pi \in \text{plays}(A)$ consistent with σ_k. Intuitively, this means that if Player k uses strategy σ_k then the value $f(\pi)$ of any resulting

play is not worse than v. Now, there exists an optimal strategy for Player k, whenever the maximum value $v_k(G) = \max\{(-1)^k v \mid \text{some } \sigma_k \text{ secures } v\}$ exists. The game G is *determined*, whenever $v_0(G)$ and $v_1(G)$ exist and are equal.

Theorem 1. *The latency game for $\theta = 1$ is memorylessly determined, and a memoryless optimal strategy can be found for it in $\mathcal{O}(|V|^2 \cdot |E| \cdot \log(|V| \cdot T) \cdot T)$ time, with $T = \max_{(a,b)\in E} u \cdot t(a,b)$ and u the speed of the processors.*

Proof. For $\theta = 1$, a latency game coincides with a mean payoff game (cf., Example 8). Ehrenfeucht and Mycielski show that mean payoff games are memorylessly determined [12]. Brim et al. provide a pseudopolynomial time algorithm for finding an optimal memoryless strategy [7].

In view of Example 8, the result of Theorem 1 shows that there exists an optimal strategy (determinacy) of good quality (memoryless) for maximizing the average time between two consecutive context-switches. For optimal play, the scheduler need not remember earlier scheduling decisions. Moreover, such an optimal strategy can be efficiently computed from the scheduling arena.

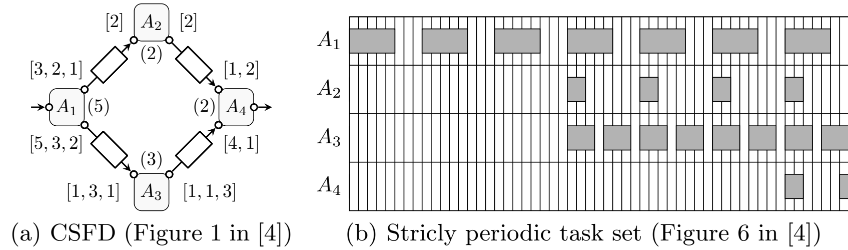
4 Cyclo-static dataflow

In a streaming application, a network of filters transforms an input stream of data into an output stream. Examples of such applications range from video decoding to sorting algorithms. A streaming application can be formally represented by a *cyclo-static dataflow* (CSDF) graph. Bamakhrama and Stefanov proposed a framework for scheduling CSDF graphs that are annotated with worst-case execution times [4]. In this section, we argue that our proposed scheduling framework of Section 3 subsumes this scheduling framework for CSDF graphs.

Consider the CSDF graph in Figure 5(a), wherein four filter processes A_1, A_2, A_3 and A_4, called *actors*, are connected by FIFO buffers. The behaviour of an actor consists of a periodic sequence of execution steps, whose worst-case execution time is represented the value (μ_i) at A_i, for $i \in \{1,2,3,4\}$. In each step, an actor atomically consumes tokens from its input buffers and produces tokens for its output buffers. The production and consumption rates of an actor A_i with period n are determined by vectors $[x_1, \ldots, x_n]$ (Section 3.1 in [4]).

Bamakhrama and Stefanov generate from a CSDF graph with worst-case execution times a strictly periodic task set (Example 3 in [4]), by determining for every filter A_i, a starting time $S_i \in \mathbb{N}_0$ and a period $T_i \in \mathbb{N}_0$ such that all required input tokens are available for all execution steps of all actors and all buffers need only a finite capacity throughout the execution. Figure 5(b) shows the strictly periodic task-set of the CSDF graph in Figure 5(a). Bamakhrama and Stefanov then use standard scheduling algorithms for strictly periodic task set to compute a schedule $\mathcal{S}$ for this CSDF graph (cf., Section 3.2.2 in [4]).

Every actor A_i, with $i \in \{1,2,3,4\}$, can be represented as a work automaton over a single job x_i, and every buffer can be represented as a work automaton without jobs. Using the composition operator from Section 2.5, we find a work



(a) CSFD (Figure 1 in [4]) (b) Stricly periodic task set (Figure 6 in [4])

Fig. 5. Cyclo-static dataflow model (a) and strictly periodic task set (b).

automaton $\mathcal{A}$ that describes the behaviour of the CSFD graph in Figure 5(a). The behaviour of $\mathcal{A}$ is fully under the control of the scheduler. Hence, Section 3.3 shows that, for regular scheduling objectives, the schedule $\mathcal{S}$ obtained in [4] induces a strategy in the scheduling arena A over $\mathcal{A}$ and $\mathcal{M}_4$.

We conclude that, for regular scheduling objectives, a schedule induced by an optimal scheduling strategy in a scheduling game is not worse than any schedule found by the scheduling framework proposed by Bamakhrama and Stefanov.

5 Discussion

We extended constraint automata with job constraints to model the work of processes in a concurrent application. We recognize that scheduling decisions do not completely determine the execution of a concurrent application, and therefore view scheduler synthesis as playing a game on a graph between a scheduler and the application. We introduced a class of natural scheduling objectives, and applied game-theoretic results for mean payoff games to find a scheduling strategy that maximizes the time between subsequent context-switches.

Work automata are similar to timed automata [1]. Clock constraints and clock valuations correspond to job constraints and progress of jobs. Still, there are two main differences between them. First, we reset only required jobs in work automata, while in timed automata clocks can reset at any time. Second, we allow jobs to make progress at different speeds, while clocks in timed automata increment uniformly. Using this clock-speed relaxation, the scheduler controls the execution rate of each job by selecting which jobs to schedule. Using our notion of jobs, it seems possible to represent the execution of a concurrent application on a set of processors by means of hybrid automata [14] or hybrid constraint automata [9]. However, since such a representation convolutes the specification of the application with the specification of the parallel machine, hybrid (constraint) automata are unsuitable for our purpose.

Scheduler synthesis for concurrent applications is similar to controller synthesis for real-time systems [5, 6, 8, 17], because the non-deterministic behaviour of a real-time system, modeled as a timed automaton [1], is not fully determined by its controller. Therefore, the controller synthesis problem is formulated as

a game on the automaton that is played between the controller and an adversary. However, our problem differs from controller synthesis in that scheduler synthesis requires a strong relation between processes and processors.

The size of a composed work automaton for a whole application very quickly becomes too large. Moreover, the size of a scheduling arena is again much larger than that of the work automaton. Nevertheless, an optimal strategy in such an immense game may indeed have a very simple form (like balancing production and consumption rates in buffers). One direction for our future work is to investigate under what conditions it is possible to bypass these exponential blow-ups. The existence of efficient solutions for more restricted scheduling problems (e.g., CSDF applications [4]) let us believe that it is possible to find such conditions.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126, 183–235 (1994)
2. Arbab, F.: Puff, the magic protocol. In: Formal Modeling: Actors, Open Systems, Biological Systems, LNCS, vol. 7000, pp. 169–206. Springer (2011)
3. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Programming 61(2), 75–113 (2006)
4. Bamakhrama, M.A., Stefanov, T.P.: On the hard-real-time scheduling of embedded streaming applications. Des. Autom. Embed. Syst. 17(2), 221–249 (2013)
5. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal strategies in priced timed game automata. In: Proc. of FSTTCS '04. LNCS, vol. 3328, pp. 148–160
6. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Synthesis of optimal strategies using hytech. Electron. Notes Theor. Comput. Sci. 119(1), 11 – 31 (2005)
7. Brim, L., Chaloupka, J., Doyen, L., Gentilini, R., Raskin, J.F.: Faster algorithms for mean-payoff games. Form. Method. Syst. Des. 38(2), 97–118 (2011)
8. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Proc. of CONCUR. pp. 66–80. Springer (2005)
9. Chen, X., Sun, J., Sun, M.: A hybrid model of connectors in cyber-physical systems. In: Proc. of ICFEM. pp. 59–74 (2014)
10. Crovella, M., Frangioso, R., Harchol-Balter, M.: Connection scheduling in web servers. In: Proc. of USITS. vol. 10, pp. 243–254 (1999)
11. Droste, M., Kuich, W., Vogler, H.: Handbook of weighted automata. Springer Science & Business Media (2009)
12. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. Int. J. of Game Theory 8(2), 109–113 (1979)
13. Gantt, H.L.: Work, wages, and profits. Engineering Magazine Co. (1913)
14. Henzinger, T.A.: The theory of hybrid automata. Springer (2000)
15. Jha, N.K.: Low power system scheduling and synthesis. In: Proc. of ICCAD. pp. 259–263. IEEE (2001)
16. Koehler, C., Clarke, D.: Decomposing port automata. In: Proc. of SAC. pp. 1369–1373. ACM (2009)
17. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems. In: Proc. of STACS. pp. 229–242. Springer (1995)
18. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: A language for streaming applications. In: Proc. of ETAPS. pp. 179–196. Springer (2002)