On Scheduling Policies for Streams of Structured Jobs*

Aldric Degorre, Oded Maler

Verimag-UJF-CNRS, 2 av. de Vignate, 38610 Gières, France
{Aldric.Degorre, Oded.Maler}@imag.fr

Abstract. We study a class of scheduling problems which combines the structural aspects associated with task dependencies, with the dynamic aspects associated with ongoing streams of requests that arrive during execution. For this class of problems we develop a scheduling policy which can guarantee bounded accumulation of backlog for all admissible request streams. We show, nevertheless, that no such policy can guarantee bounded latency for all admissible request patterns, unless they admit some laxity.

1 Introduction

The problem of efficient allocation of reusable resources over time, also known as *scheduling*, is a universal problem, appearing almost everywhere, ranging from the allocation of machines in a factory [22,6,19], allocation of processor time slots in a real-time system [20,9], allocating communication channels in a network [16], or allocation of vehicles for transportation tasks [7]. Unfortunately, the study of scheduling problems is distributed among many academic communities and application domains, each focusing on certain aspects of the problem.

In the vast scheduling literature, one can, very roughly, identify two generic types of problems. In the first type, the work to be scheduled admits a *structure* which includes precedence constraints between tasks, but the problems are, more often than not, *static*: the work to be executed is known in advance and is typically finite. Examples of this type of problems are the job-shop problem motivated by manufacturing (linear precedence constraints, heterogeneous resources) [19,18] or the task-graph scheduling problem, motivated parallel execution of programs (partially-ordered tasks, homogeneous resources) [14] (some recurrent aspects of scheduling are exhibited in program loop parallelization [13] but the nature of uncertainty there is different and rather limited).

On the other hand, in problems related to real-time systems [10] or in queuing theory [17], one is concerned with *infinite streams* of tasks which arrive either periodically or sporadically (or in a combination of both), satisfying some constraints on task arrival patterns. In many of these "dynamical" problems, the structural dimension of the problem is rather weak, and each request consists of

* This research was partially supported by the French Minlogic project Athole.

a monolithic amount of work. A notable exception is the domain of adversarial queuing theory [8] where some structure and uncertainty are combined.

In this paper we propose a model which combines the *dynamic* aspect associated with *request streams* whose exact content is not known in advance, with the *structural* aspects expressed by task dependencies. We define a scheduling problem where the demand for work is expressed as a stream of requests, each being a structured *job* taken from a finite set of types, hence such a stream can be viewed as a timed word over the the alphabet of job types. Each job type defines a finite partially-ordered set of tasks, each associated with a resource type and a duration. Such a stream is to be scheduled on an *execution platform* consisting of a finite number of resources (machines). A schedule is valid relative to a request stream if it satisfies *both* the precedence constraints imposed by the structure of the jobs and the resource constraints imposed by the number of resources available in the platform (and, of course, it does not execute jobs before they are requested).

The quality of a specific schedule is evaluated according to two types of measures, one associated with the evolution of the *backlog* over time, that is, the difference between the amount of work requested and the amount of work supplied, and the *latency*, the temporal distance between the arrival of a job instance and the termination of its execution. To model the uncertain external environment we use the concept of a *request generator*, a set of request streams satisfying some inter-arrival timing constraints. Such constraints can be expressed, for example, using timed automata [2], real-time logics [3] or timed regular expressions [4]. We restrict the discussion to *admissible* request streams that do not demand more work over time than the platform can offer. A *scheduling policy* (strategy) should produce a schedule for each admissible request stream, subject to *causality* constraints: the decision of the scheduler at a given moment can only be based on the *prefix* of the request stream it has seen so far.

After defining all these notions we prove two major fundamental results:

- Positive: we develop a scheduling policy which produces a bounded backlog schedule for any admissible request stream. Note that due to the precedence constraints between the tasks in the jobs, request stream admissibility does not, a priori, guarantee the existence of such a schedule. In fact, we show that a naive “oldest first” policy can accumulate an unbounded backlog for certain request streams. Our policy achieves this goal by making decisions that provide for pipelined execution whenever possible.
- Negative: there are admissible request streams for which no bounded-latency schedule (and hence no bounded-latency policy) exists.

The rest of the paper is organized as follows: in Sect. 2 we define our scheduling framework, in Sect. 3 we prove a negative result concerning the impossibility of bounded latency schedules. In Sect. 4 we extend the framework to include scheduling policies and in Sect. 5 we develop a scheduling strategy that guarantees bounded backlog. We conclude with a discussion of past and future work.

2 The Recurrent Scheduling Problem

2.1 General Definitions

We use timed words and timed languages to specify streams of requests for work. Intuitively, a timed word such as $\tilde{u} = 3 \cdot a_1 \cdot 2 \cdot a_2 \cdot a_3 \cdot 6$ consists of a passage of time of duration 3, followed by the event a_1, followed by a time duration 2, followed by the two events a_2 and a_3 and then a time duration of 6. We present some basic definitions and notations (see more formal details in [4]).

- A word over an event alphabet Σ is either ϵ, the empty word, or $u \cdot a$ where u is a word and $a \in \Sigma$. An ω-word is an infinite sequence $(a_i)_{i\in\mathbb{N}} \in \Sigma^\omega$.
- A timed word over Σ is a word over $\Sigma \cup \mathbb{R}_+$. The duration of a timed word u, denoted by $|u|$ is the sum of its elements that are taken from $\mathbb{R}_+$, for example $|\tilde{u}| = 11$. A timed ω-word is an infinite sequence $(a_i)_{i\in\mathbb{N}} \in (\Sigma \cup \mathbb{R}_+)^\omega$ such that its duration diverges.
- The concatenation of a word u and a word (or ω-word) v is denoted by $u \cdot v$.
- A word u is a prefix of v iff there exists w such that $v = u \cdot w$, which we denote $u \sqsubseteq v$. We say that u is a proper prefix of v, denoted by $u \sqsubset v$, if $u \neq v$.
- A word (or an ω-word) u is a suffix of v iff there exists w such that $v = w \cdot u$.

For a timed (ω-)word u over Σ

- By $u(a, i)$ we denote the time of the i-th occurrence of event $a \in \Sigma$ in the timed word u. Formally $u(a, i) = t$ if $u = v \cdot a \cdot w$ such that $|v| = t$ and v contains $i - 1$ occurrences of a. We let $u(a, i) = \infty$ when a occurs less than i times in u.
- The timed word $u_{[0,t]}$ is the longest prefix of u with duration t. Formally $u_{[0,t]} = t_0 \cdot a_0 \cdot t_1 \cdot a_1 \cdot ... \cdot t_i$ such that $\sum_{0\leq k\leq i} t_k = t$ and there exists no discrete event a such that $t_0 \cdot a_0 \cdot t_1 \cdot a_1 \cdot ... \cdot t_i \cdot a$ is a prefix of w. For example, $\tilde{u}_{[0,4]} = 3 \cdot a_1 \cdot 1$ and $\tilde{u}_{[0,5]} = 3 \cdot a_1 \cdot 2 \cdot a_2 \cdot a_3 \cdot 0$.

Sets of timed (ω-)words over Σ are called timed (ω-)language. We denote the sets of such languages by $\mathcal{T}(\Sigma)$ and $\mathcal{T}_\omega(\Sigma)$, respectively.

2.2 Execution Platform, Jobs and Tasks

The execution platform determines our capacity to process work.

Definition 1 (Execution Platform) *An execution platform over a finite set $M = \{m_1, \ldots, m_n\}$ of resource (machine) types is a function $R : M \to \mathbb{N}$.*

Example: $\{m_1 \mapsto 2, m_2 \mapsto 4, m_3 \mapsto 1\}$ is an execution platform with three resource types m_1, m_2, m_3 having 2 instances of m_1, 4 instances of m_2, and 1 instance of m_3.[1]

[1] We will use the notation R_m for $R(m)$ and R when we want to treat the whole platform capacity as vector and make component-wise arithmetical operations. The same will hold for sets of functions indexed by the elements of M.

The *task* is the atomic unit of work, specified by the resource type it consumes and by its duration. The job is a unit of a larger granularity, consisting of tasks related by precedence constraints. Each job is an instantiation of a job type.

Definition 2 (Job Type) *A job type over a set M of resources is a tuple $J = \langle T, \prec, \mu, d\rangle$ such that $\prec \subseteq T \times T$ and $\langle T, \prec\rangle$ is a finite directed acyclic graph whose nodes are labelled by 2 functions: $\mu : T \to M$, which associates a task to the resource type it consumes, and $d : T \to \mathbb{R}_+ - \{0\}$ specifying task duration.*

As an example consider a job type where $T = \{a_1, a_2, a_3\}$, $\prec = \{(a_1 \prec a_3), (a_2 \prec a_3)\}$ $\mu = \{a_1 \mapsto m_1, a_2 \mapsto m_2, a_3 \mapsto m_3\}$, $d = \{a_1 \mapsto 3, a_2 \mapsto 2, a_3 \mapsto 1\}\rangle$, where a_1 needs resource m_1 for 3 time units, a_2 uses resource m_2 for 2 time units while a_3 consumes m_3 for 1 time unit. Task a_3 cannot start before both a_1 and a_2 terminate.

For a set $\mathcal{J} = \{\langle T_1, \prec_1, \mu_1, d_1\rangle, ..., \langle T_n, \prec_n, \mu_n, d_n\rangle\}$ of job types, we let $T_\mathcal{J}, \prec_\mathcal{J}, \mu_\mathcal{J}$ and $d_\mathcal{J}$ denote, respectively, the (disjoint) union of T_i, $\prec_i$, μ_i and d_i, for $i = 1..n$. We call elements of $T_\mathcal{J}$ *task types*. When $\mathcal{J}$ is clear from the context we use notations T, $\prec$, μ and d.

Definition 3 (Initial Tasks, Rank) *An initial task a is an element of T such that there exists no $a' \in T$ with $a' \prec a$. The rank of task a is the number of edges of the longest path $a_0 \prec a_1 \prec \ldots \prec a$ such that a_0 is initial. Initial tasks have rank* 0.

2.3 The Demand

The sequence of jobs and tasks that should be executed on the platform is determined by a request stream.

Definition 4 (Request Streams and Generators) *A request stream over a set $\mathcal{J}$ of job types is a timed ω-word over $\mathcal{J}$. A request generator is a timed ω-language over $\mathcal{J}$.*

Each request stream presents a demand for work over time which should not exceed the platform capacity, otherwise the latter will be saturated.

Definition 5 (Work Requested by Jobs and Streams) *With each resource type m we define a function $W_m : \mathcal{J} \to \mathbb{R}_+$ so that $W_m(J)$ indicates the total amount of work on m demanded by job J, $W_m(J) = \sum_{\{a \in T_J : \mu(a) = m\}} d(a)$. We lift this function to request stream prefixes by letting $W(\epsilon) = 0$, $W(u \cdot t) = W(u)$ for $t \in \mathbb{R}_+$ and $W(u \cdot J) = W(u) + W(J)$ for $J \in \mathcal{J}$.*

We restrict our attention to request streams that do not ask for more work per time unit than the platform can provide, and, furthermore, do not present an unbounded number of requests in a bounded time interval.

Definition 6 (Admissible, Critical and Subcritical Request Streams) *A request stream σ is α-lax ($\alpha \in \mathbb{R}_+$) with respect to an execution platform R if for every $t < t'$, $W(\sigma_{[0,t']}) - W(\sigma_{[0,t]}) \leq \alpha(t'-t)R + b$ for some constant $b \in \mathbb{R}^n$. A stream is admissible if it is α-lax for some $\alpha \leq 1$, subcritical if it is α-lax for $\alpha < 1$ and critical if it is admissible but not subcritical. A request generator G is α-lax if every $\sigma \in G$ is α-lax.*

2.4 Schedules

Definition 7 (Schedule) *A schedule is a function $s : T \times \mathbb{N} \to \mathbb{R}_+^\infty$ (where $\mathbb{R}_+^\infty = \mathbb{R}_+ \cup \{\infty\}$ with the usual extension of the order and operations).*

The intended meaning of $s(a, i) = t$ is that the i-th instance of task a (which is part of the i-th instance of the job type to which it belongs) starts executing at time t. If we restrict ourselves to "non-overtaking" schedules[2] such that $s(a, i) \leq s(a, i')$ whenever $i < i'$, we can view a schedule as a timed ω-word in $\mathcal{T}_\omega(T)$. Likewise we can speak of finite prefixes $s_{[0,t]}$ which are timed words in $\mathcal{T}(T)$.

Since tasks have fixed durations and cannot be preempted, a schedule determines uniquely which tasks are executed at any point in time and, hence, how many resources of each type are utilized, a notion formalized below.

Definition 8 (Utilization Function, Work Supplied) *The resource utilization function associated with every resource m is $U_m : \mathcal{T}_\omega(T) \times \mathbb{R}_+ \to \mathbb{N}$ defined as $U_m(s, t) = |\{(a, i) \in T \times \mathbb{N} : \mu(a) = m \wedge s(a, i) \leq t < s(a, i) + d(a)\}|$. The work supplied by a prefix of s is the accumulated utilization: $W(s_{[0,t]}) = \int_0^t U(s, \tau) d\tau$.*

Definition 9 (Valid Schedule) *A schedule s is valid for a request stream σ on an execution platform R if for any task instance (a, i)*

- *if J is the job type a belongs to, then $s(a, i) \geq \sigma(J, i)$ (no proactivity: jobs are executed after they are requested);*
- *$\forall a', a' \prec a, s(a, i) \geq s(a', i) + d(a')$ (job precedences are met);*
- *$\forall t \in \mathbb{R}_+, U(s, t) \leq R$ (no overload: no more resource instances of a type are used than their total amount in the execution platform).*

The quality of a schedule can be evaluated in two principal and related (but not equivalent) ways, the first of which does not look at individual job instances but is based on the amount of work. During every prefix of the schedule there is a non-negative difference between the amount of work that has been requested and the amount of work that has been supplied. This difference can be defined in a "continuous" fashion like $\Delta_{\sigma,s}(t) = W(\sigma_{[0,t]}) - W(s_{[0,t]})$. An alternative that we will use, is based on the concept of *residue* or backlog, which is simply the set of requested tasks that have not yet started executing. It is not hard to see that a bounded residue is equivalent to a bounded difference between requested and supplied work.

[2] Note that non-overtaking applies only to tasks of the *same type*.

Definition 10 (Residue, Bounded Residue Schedules) *The residue associated with a request stream σ and a valid schedule s at time t is $\rho_{\sigma,s}(t) = \{(a,i) \in T \times \mathbb{N} : \sigma(a,i) \leq t < s(a,i)\}$. A valid schedule s is of bounded residue if there is a number c such that $|\rho_{\sigma,s}(t)| \leq c$ for every t.*

The second performance measure associated with a schedule is related to latency, the time an individual job has to wait between being requested and the completion time of its last task.

Definition 11 (Latency) *Given a request stream σ and a valid schedule s, the latency of a job instance (J,i) is $L_{J,i}(\sigma,s) = \max_{a \in T_J}\{(s(a,i)+d(a))\} - \sigma(J,i)$. The latency of s with respect to σ is $L(\sigma,s) = \sup_{J \in \mathcal{J}, i \in \mathbb{N}} L_{J,i}(\sigma,s)$.*

Note that it is possible that every job instance is served in finite time but the latency of the schedule is, however, infinite, that is, the sequence $\{L_{J,i}\}_{i \in \mathbb{N}}$ may diverge. Bounded residue does not imply bounded latency: we can keep one job waiting forever, while still serving all the others without accumulating backlog. But the implication holds in the other direction.

Lemma 1 *A valid schedule with bounded latency has a bounded residue.*

Proof. Let s be a valid schedule with latency $\lambda \in \mathbb{R}_+$. Let $V(t)$ be the total amount of work of the tasks that are in the residue at time t. Since all these tasks are supposed to be completed by $t + \lambda$ we have $V(t) \leq \lambda R$ which implies a bound on the residue. □

2.5 The Running Example

We will use the following recurrent scheduling problem to construct the negative result and to illustrate our scheduling policy. Consider a platform over $M = \{m_1, m_2\}$ with $R(m_1) = R(m_2) = 1$. The set of job types is $\mathcal{J} = \{A, B\}$ whose respective sets of tasks $\{a_1 \prec a_2\}$ and $\{b_1 \prec b_2\}$ have all a unit duration. The difference between these job types is that A uses m_1 before m_2 while B uses m_2 before m_1 (see Fig. 1). As a request generator we consider $G = ((A \cdot 1) + (B \cdot 1))^{\omega}$, that is, every unit of time, an instance of either one of these jobs is requested (to simplify notations we will use henceforth A and B as a shorthand for $A \cdot 1$ and $B \cdot 1$, respectively). Since each job type requires exactly the amount of work offered by the platform, G is admissible and, in fact, critical. A bounded-residue schedule for such critical request streams should keep the machines busy *all the time* except for some intervals (that we call *utilization gaps*) whose sum of durations is bounded.

The reversed order of resource utilization in A and B renders these two job types *incompatible* in the sense that it is not easy to "pipeline" them on our platform. Intuitively at the moment a request stream switches from A to B, we may have tasks a_2 and b_1 ready for execution but only one instance of their common resource m_2 is free. Our scheduling policy will, nevertheless, manage to pipeline them but, as we show in the next section, bounded latency schedules are impossible.

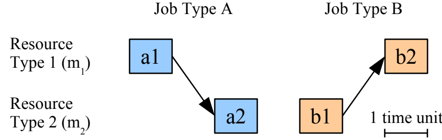

Fig. 1. The example.

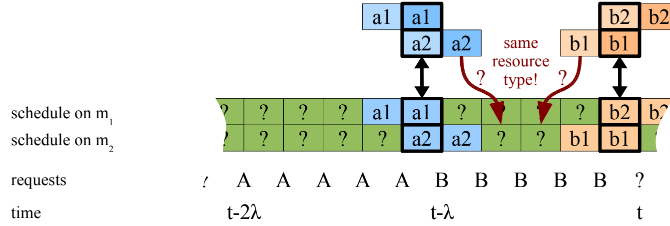

Fig. 2. An illustration of the fact that a request segment $A^\lambda \cdot B^\lambda$ implies a utilization gap in any schedule of latency λ or less. Before $t - \lambda$: job type A has been requested for a long time, so the residue contains only tasks from A. At $t - \lambda$: from now on, requests are of type B. After $t - 1$: if the latency is λ, there should be no more tasks from A in the residue. Now between $t - \lambda$ and $t - 1$, only suffixes of A and prefixes of B can be scheduled, and among those, at least one proper suffix.

3 Negative Result

Theorem 1 *Some admissible request streams admit no bounded-latency schedule.*

We prove the theorem using the following lemma, which shows that for any latency λ, the occurrence of a certain request pattern implies a unit increase in the residue and hence infinitely many consecutive repetitions of this pattern will imply an unbounded residue. The statement of the lemma and its proof are illustrated in Fig. 2.

Lemma 2 *Let σ be a request stream with a prefix of the form $\sigma_{[0,t]} = u \cdot A^\lambda \cdot B^\lambda$ and let s be a valid schedule for σ with latency λ. Then there is a utilization gap (an idle resource) of duration 1 or more in the interval $[t - \lambda - 1, t]$.*

Proof. Since the latency of s is λ, no task instance of B belongs to the residue $\rho_{\sigma,s}(t - \lambda - 1)$, so the only way to avoid a gap at time $t - \lambda - 1$ is to schedule an instance of a_1 and an instance of a_2. For the same reason, $\rho_{\sigma,s}(t - 1)$ contains no task instance of A, so that at time $t - 1$, s schedules b_1 and b_2. Moreover any task instance of B in the residue after $t - \lambda - 1$ is an instance that was requested since $t - \lambda$.

Now what happens in $[t-\lambda, t-1]$? In that interval, the residue has task instances from requests for A made before $t-\lambda$ and from requests for B made since that time. Due to bounded latency all the instances from A are due for $t-1$. We also know that, because $a_1 \prec a_2$, the residue has always more a_2 than a_1, and that their amount is the same only when all started job instances of A are finished, which is not possible at $t-1$ because an a_1 is scheduled for $t-\lambda-1$ (and thus task a_2 of the same job instance cannot start before $t-\lambda$). In that interval we also schedule task instances from B the earliest of which can have started execution at $t-\lambda$. Thus, since $b_1 \prec b_2$, we cannot schedule more b_2 than b_1.

Summing up the quantity of work scheduled by s between $t-\lambda$ and t, we find that on m_1 we schedule n_{a_1} instances of a_1 and n_{b_2} instances of b_2 and on m_2 we schedule n_{a_2} instances of a_2 and n_{b_1} instances of b_1, satisfying $n_{a_2} > n_{a_1}$ and $n_{b_1} \geq n_{b_2}$. Thus m_2 performs at least one unit of work more than m_1 in the same interval, which is only possible if m_1 admits a utilization gap of duration 1. $\square$

Consider now a request stream that has infinitely many occurrences of the pattern $u \cdot A^\lambda \cdot B^\lambda$. A schedule with latency λ for this stream will have infinitely many gaps, and hence an unbounded residue, a fact which contradicts Lemma 1. Hence such a stream admits no schedule whose latency is λ or less.

Proof (of Theorem 1). Consider now any request stream σ in the language $L_\infty = \mathcal{J}^* \cdot A \cdot B \cdot B \cdot \mathcal{J}^* \cdot A \cdot A \cdot A \cdot B \cdot B \cdot B \cdot B \cdot \mathcal{J}^* \cdot A \cdot A \cdot A \cdot B \cdot B \cdot B \cdots$, where $\mathcal{J}$ stands for $(A+B)$. For every λ, σ has infinitely many prefixes of the form $u \cdot A^\lambda \cdot B^\lambda$ and cannot have a schedule of latency λ. Consequently it admits no bounded latency schedule. $\square$

Note that this impossibility result is not related to the dynamic aspect of the scheduling problem. Even a clairvoyant scheduler who knows the whole request stream in advance cannot find a bounded latency solution.

Note also that the language L_∞ is not pathological. If fact, in any reasonable way to induce probabilities on $(A+B)^\omega$, this language will have probability of 1. Hence we can say that critical systems having two incompatible jobs will almost surely admit only unbounded-latency schedules.

4 Scheduling Policies

Now we want to consider the act of scheduling as a dynamic process where a scheduler has to adapt its decisions to the evolution of the environment, here the incoming request stream. We want the scheduler to construct a schedule *incrementally* as requests arrive. The mathematical object that models the procedure of mapping request stream prefixes into scheduling decisions is called a scheduling *policy* or a *strategy*.

Formally speaking, a policy can be viewed as a timed transducer, a causal function $p : \mathcal{T}_\omega(\mathcal{J}) \to \mathcal{T}_\omega(T)$ which produces for each request stream σ a valid schedule $s = p(\sigma)$. Causality here means that the value of $s_{[0,t]}$ depends only

on $\sigma_{[0,t]}$. We will represent the policy as a procedure p which, at each time instant t, looks at $\sigma_{[0,t]}$ and selects a (possibly empty) set of task instances to be scheduled for execution at time t, that is, $s(a,i) = t$ if $(a,i) \in p(\sigma_{[0,t]})$. We will use $s_{[0,t]} = p(\sigma_{[0,t]})$ to denote the schedule prefix constructed by successive applications of p during the interval $[0,t]$. We assume that each policy is designed to work with admissible request streams taken from a generator $G \subseteq \mathcal{T}_\omega(\mathcal{J})$.

Definition 12 (Scheduling policy) *A scheduling policy is a function $p : \mathcal{T}(\mathcal{J}) \to 2^{T \times \mathbb{N}}$ such that for every task instance (a,i) and a request stream prefix σ, $(a,i) \in p(\sigma)$ implies that $(a,i) \notin p(\sigma')$ for any $\sigma' \sqsubset \sigma$. A scheduling policy is valid for σ if for every t, the obtained schedule $s_{[0,t]} = p(\sigma_{[0,t]})$ satisfies the conditions of Definition 9, namely, no proactivity and adherence to precedence and resource constraints.*

We evaluate the overall performance of a policy based on the worst schedule it produces over the streams in the generator. Since we have just shown a negative result concerning latencies, we focus on the residue.

Definition 13 (Bounded Residue Policies) *A scheduling policy has a bounded residue relative to a generator G if it produces a bounded-residue schedule for every $\sigma \in G$.*

In the following, we use notation $\rho_{\sigma,p}$ instead of $\rho_{\sigma,p(\sigma)}$ to denote the residue resulting from the application of a policy p to a request stream (or prefix) σ.

5 Positive Result

In this section we show that any recurrent scheduling problem with an admissible request generator admits a policy in the sense of Sect. 4 which maintains the residue bounded. We emphasize again that the policy makes decisions at run time without knowing future requests.

5.1 Oldest-First Policy Does Not Work

To appreciate the difficulty, let us consider first a naive Oldest-First policy: whenever the number of tasks that are ready to use a resource is larger than the number of free instances of the resource, the available instances are granted to the older tasks among them. We show that this policy fails to guarantee bounded residues.

Theorem 2 *The Oldest-First policy cannot guarantee a bounded residue.*

In fact, this policy will lead to an unbounded residue schedule for request streams in the language L_∞ of the previous section as illustrated in Fig. 3 and proved below. The reason is, again, the incompatibility between the job types, which leads to infinitely many utilization gaps where a resource is free while none of the corresponding tasks in the residue is ready to utilize it. The result is a direct corollary of the following lemma:

Lemma 3 *A bounded residue schedule which conforms to the Oldest-First policy has a bounded latency.*

Note that we already proved the converse for arbitrary schedules and policies.

Proof. First we show that any task instance (a, i) that becomes eligible for execution at time t, is scheduled for execution within a bounded amount of time after t. This holds because, following the policy, the only tasks that can be executed between t and $s(a, i)$ are those that are already in the (bounded) residue at time t. Next we show, by induction on the rank of the tasks, that this fact implies that any task is executed within a bounded amount of time after its job is issued. This holds trivially for the initial tasks which become eligible for execution immediately when the job arrives and then holds for tasks of rank $n + 1$ by virtue of the bounded latency of tasks of rank n. Thus the latency of a bounded-residue schedule produced by the Oldest-First has to be bounded. □

Since we have already shown that request streams in L_∞ do not admit bounded-latency schedules, a bounded residue strategy will lead to a contradiction and this proves Theorem 2. Like the case for Theorem 1, under reasonable probability assignments to jobs, one can show that the Oldest-First policy will almost surely lead to unbounded-residue schedules when applied to critical streams of incompatible jobs.

schedule on m_1	a1			b2	b2	a1	a1	a1			b2	b2	b2
schedule on m_2		a2	b1	b1			a2	a2	a2	b1	b1	b1	b1
residue on m_1	0	1	2	2	2	2	2	2	3	4	4	4	4
residue on m_2	1	1	1	1	2	3	3	3	3	3	3	3	3
requests	A	B	B	A	A	A	B	B	B	B	A	A	A

Fig. 3. Schedule generated by the "oldest first" policy on a stream in the language L_∞, described in 3. Here we see that a gap of length 2 is created on one of the resource types at every change of job type in the request stream, which makes the residue grow indefinitely.

5.2 A Bounded Residue Policy

Theorem 3 *Any admissible generator admits a bounded-residue scheduling policy.*

In order to circumvent the shortcomings of the "Oldest First" policy, we describe in the sequel a policy that eventually reaches the following situation: whenever a resource becomes free and the residue contains tasks that need it, at least one of those tasks will be ready for execution.

The policy is described in detail in Algorithm 1 and and proved correct in App. A. We explain the underlying intuition below. The policy separates the act of choosing which tasks to execute in the future from the act of actually starting them. The first decision is made upon job arrival while the second is made whenever a resource is free and a corresponding task has been selected. To this end we partition the residue into two parts. The first part P (the "pool") consists of requested task instances that have not yet been selected for execution. Among those, only task instances whose $\prec$-predecessors have already terminated are eligible for being selected and moved to the other part, which consists of n FIFO queues $\{Q_m\}_{m\in M}$, one for each resource type. The passage between the two is controlled by two types of events:

- Task termination: when a task (a, i) terminates, eligibility status of its successors in P is updated;
- Job arrival: when a job instance (J, i) arrives we pick the oldest[3] eligible instance $(a, j_a) \in P$ (if such exists) for every task type $a \in T_J$ such that $\mu(a) = m$, and move it to Q_m. Note that only initial tasks of (J, i) are eligible for being selected when (J, i) arrives, while for other task types only earlier instances can be chosen.

Whenever a resource of type m is free and Q_m is not empty, the first element is removed from Q_m and starts executing. This is sufficient to ensure a bounded residue. However, to improve the performance of the algorithm when the streams are subcritical, we also choose to start the oldest eligible task which requires m if an instance of m is released when Q_m is empty.

The intuition why this policy works is easier to understand when we look at critical request streams. For such streams, any job type which is requested often enough will eventually have instances of each of its tasks in Q and hence, whenever a resource is freed, there will always be some task ready to use it. This guarantees smooth pipelining and bounded residue for all admissible request streams. In Fig. 4 we can see how our policy schedules the request stream $\sigma_\infty = A \cdot B \cdot B \cdot A \cdot A \cdot A \cdots$.

Scheduling policies of the FIFO type have also been studied in the context of adversarial queuing and it has been shown under various hypothesis [12] that those were not stable, sometimes even for arbitrarily small loads. What makes our policy work is the fact that the act of queuing is triggered by a global event (arrival of a new job request) on which the actual choice of tasks to be queued depends. So the decision is somehow "conscious" of the global state of the system, as opposed to what happens in a classical FIFO network.

5.3 Bounded Latency for Subcritical Streams

We just showed that a policy could ensure bounded residues in the case of critical streams for which one needs full utilization. But criticality is just a limit case and for that reason it is interesting to know whether such a policy can adapt and

[3] Or one of the oldest if there are several of the same age.

Algorithm 1 The Bounded-Residue Policy

```
declarations
    req: jobType inputEvent                              // events from σ
    free: resourceType inputEvent        // triggered when a resource is freed
    start: taskInstance outputEvent               // scheduling decisions
    P: taskInstance set                          // pool, unselected tasks
    Q: resourceType → (taskInstance fifo)        // queues, selected tasks

procedure INIT
    P = ∅;
    for all m ∈ M do Q_m = ∅

procedure STARTWORK(m: resourceType)
    if Q_m is not empty then α = pop(Q_m); emit start(α)
    else                 // for bounded latency against subcritical streams
        if P has eligible task instances requiring m then
            α = the oldest eligible task instance using m in P;
            P = P − {α}; emit start(α)

on free(m) do startWork(m)

on req(J) do
    for all a ∈ T_J do
        P = P ∪ {newInstance(a)};
        if P has eligible task instances of type a then
            α = the oldest eligible task instance of a in P;
            P = P − {α}; push(α, Q_m)                 // select for execution
    for all m ∈ M do
        for all free instances of m do startWork(m)
```

schedule on m_1	a1		b2	a1	a1	a1	b2	b2	b2	b2	a1	a1	a1
schedule on m_2		b1	b1	a2	a2	a2	b1	b1	b1	b1	a2	a2	a2
residue on m_1	0	1	1	1	1	1	1	1	1	1	1	1	1
residue on m_2	1	1	1	1	1	1	1	1	1	1	1	1	1
requests	A	B	B	A	A	A	B	B	B	B	A	A	A

Fig. 4. The schedule generated by the bounded-residue policy for σ_∞. We can see that after the arrival of the second B, every resource is always occupied, and that the residue does not grow after that.

behave better when the request stream is subcritical. Fortunately the answer is positive: the previously exhibited policy, by starting tasks which are not queued when a resource would be otherwise idle, ensures bounded latencies for request streams that admit some laxity.

Theorem 4 *The policy described by Algorithm 1 has a bounded latency when applied to any α-lax stream with $\alpha < 1$.*

Lemma 4 *There exists a time bound $T_{\alpha,m}$ such that any interval $[t, t + T_{\alpha,m}]$ admits a time instant where Q_m is empty, an instance of m is free and no new request arrives.*

Sketch of proof. Consider an interval of the form $[t, t + d]$ in which no machine of type m is idle. The quantity of work dequeued from Q_m is $R_m d$ and, due to laxity, the amount of work enqueued into Q_m is at most $(1 - \alpha)R_m d$. Hence the total contribution to the amount of work in Q_m is $(\alpha - 1)R_m d$ and for some sufficiently large d it will empty Q_m. □

Proof (of Theorem 4). We know that, when a task in the pool becomes the oldest task of the residue which is not queued, it becomes eligible in a bounded amount of time (all its predecessors must be in the queue). Thus we know that at most $T_{\alpha,m}$ units of time after that, this task is started (either queued or started to fill a gap). Since furthermore the residue (and hence the pool) is bounded (Thm. 3), there is a bound on the time it takes a task to become the oldest in the pool and hence to be executed. Thus we conclude that the latency of the policy is bounded. □

6 Discussion

We have proved some fundamental results on a model that captures, we believe, many real-world phenomena. Let us mention some related attempts to treat similar problems. The idea that verification-inspired techniques can be used to model and then solve scheduling problems that are not easy to express in traditional real-time scheduling models has been studied within the timed controller synthesis framework and applied to scheduling problems [23,21,5,1]. What is common to all these approaches (including [15] which analyzes *given* policies that admit task preemption) is that the scheduler is computed using a verification/synthesis algorithm for timed automata, which despite several improvements [11] are intrinsically not scalable. The policy presented in this paper does not suffer from this problem, it only needs the request generator to be admissible. Explicit synthesis may still be needed in more complex settings.

In the future it would be interesting to investigate various extensions of the model and variations on the rules of the game, in particular, moving from worst-case reasoning to average case by using probabilistic request generators and evaluating policies according to expected backlog or latency. Finally, we intend to look closer at the question of “pipelinability”, that is, the mutual compatibility

of a set of job types. Results in this direction may lead to new design principles for request servers.

Acknowlegments: We thank anonymous referees for their comments. We are indebted to Viktor Schuppan for his great help in writing the paper.

References

1. K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium*, pages 154–163, 1999.
2. R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
3. R. Alur and T. Henzinger. Logics and models of real time: A survey. In *REX Workshop*, pages 74–106, 1991.
4. E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.
5. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, 1994.
6. J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidth, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer, 2nd edition, 2001.
7. L. Bodin, B. Golden, A. Assad, and M. Ball. Routing and scheduling of vehicles and crews : The state of the art. *Computers & OR*, 10(2):63–211, 1983.
8. A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson. Adversarial queuing theory. *J. ACM*, 48(1):13–38, 2001.
9. G. Bottazzo. *Hard Real-Time Computing Systems: Predictable Schedulding Algorithms and Applications*. Real-Time Systems Series. Springer, 2nd edition, 2005.
10. M. Caccamo, T. Baker, A. Burns, G. Buttazzo, and L. Sha. Real-time scheduling for embedded systems. In D. Hristu-Varsakelis and W. Levine, editors, *Handbook of Networked and Embedded Control Systems*, pages 173–196. Birkhäuser, 2005.
11. F. Cassez, A. David, E. Fleury, K. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR*, pages 66–80, 2005.
12. V. Cholvi and J. Echagüe. Stability of fifo networks under adversarial models: State of the art. *Computer Networks*, 51(15):4460–4474, 2007.
13. A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.
14. H. El-Rewini. Partitioning and scheduling. In A. Zomaya, editor, *Parallel & Distributed Computed Handbook*, chapter 9, pages 239–273. McGraw-Hill, 1996.
15. E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.
16. C.-H. Gan, P. Lin, N.-C. Perng, T.-W. Kuo, and C.-C. Hsu. Scheduling for time-division based shared channel allocation for UMTS. *Wirel. Netw.*, 13(2):189–202, 2007.
17. G.-H. Hsu. A survey of queueing theory. *Ann. Oper. Res.*, 24(1-4):29–43, 1990.
18. A. Jain and S. Meeran. A state-of-the-art review of job-shop scheduling techniques, 1998.
19. E.G. Coffman Jr., editor. *Computer and Job-Shop Scheduling Theory*. J. Wiley, New York, 1976.

20. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
21. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
22. M. Pinedo. *Planning and Scheduling in Manufacturing and Services.* Springer Series in Operations Research and Financial Engineering. Springer, 2007.
23. H. Wong-Toi and D. Dill. Synthesizing processes and schedulers from temporal specifications. In *CAV*, pages 272–281, 1990.

A Proof of Theorem 3

Theorem 3 *Any admissible generator admits a bounded-residue scheduling policy.*

Below we prove that Algorithm 1 yields a valid schedule with (uniformly) bounded residue for any (uniformly) admissible generator, which establishes Thm. 3.

Validity Let σ be a request stream and let s be the schedule for σ produced by p. Then, for any task instance $\alpha = (a, i)$ s.t. $s(\alpha) = t$:

- α had been in the pool at some previous $t' < t$. We therefore know that s is not proactive, because the pool contains only requested job instances.
- Either α was in a queue at t. Hence, as a task gets moved from the pool to a queue only if all of its predecessors have finished, s respects dependencies. Or α has been started when the queue was empty, and in that case the algorithm states that α was ready (its predecessors were all finished).
- Execution of α is started only in procedure $startWork(m)$. That procedure starts executing only a single task instance and is called at most once for each free resource. Hence, s does not exceed resource capacities.

These 3 conditions prove that s is a valid schedule.

Bounded Residue Below we first establish boundedness of the queues (Lemma 5). After that we show boundedness of the pool by proving the following invariant (Lemma 6): we eventually enqueue, at every request for a job type, one instance of every task type that belongs to that job type. Finally, having both lemmas, it will be easy to deduce our result.

Lemma 5 *If $\sigma \in G$ is b-admissible, then $\forall m \in M, \forall t \in \mathbb{R}_+, |Q_m(t)| \leq b_m + D \cdot R_m$, where D is the maximal length of a task type, and where $Q_m(t)$ is the state of the variable Q_m at time t.*

Proof. In fact, the size of a queue at a time t is the difference between the enqueued work and the dequeued work, hence we know how to relate those quantities to the dimensions of the platform.

It is easy to see that when the policy is applied, an instance of a resource m can never stay free for longer than 0 units of time unless Q_m is empty at the time it is freed.

Therefore we just have to prove that the Q_m cannot grow arbitrarily during intervals where $U_m(s,t) = R_m$ (intervals of length 0 do not matter as the utilization function U is constant on semi-open intervals).

So let $[t_0, t_1)$ be an interval during which the utilization is full.

The amount of work dequeued during $[t_0, t_1)$ is at least the capacity available in the interval $[t_0, t_1)$ minus the maximum amount of work that may already have

been executing at the beginning of the interval. I.e., if D denotes the length of the longest task type, then we have: $W_m(s^{-1}([t_0, t_1))) \geq (t_1 - t_0)R_m - DR_m = (t_1 - t_0 - D)R_m$.

The quantity of work that is enqueued during that interval is at most the quantity of work of the request word of the interval, so it is at most $W(\sigma_{[0,t_1]}) - W(\sigma_{[0,t_0]})$.

Since σ is b-admissible, the following holds: $W_m(\sigma_{[0,t_1]}) - W_m(\sigma_{[0,t_0]}) \leq (t_1 - t_0) \cdot R_m + b_m = (t_1 - t_0 - D) \cdot R_m + b_m + D \cdot R_m \leq W_m(s^{-1}([t_0, t_1))) + b_m + D \cdot R_m$.

Therefore, in any interval where the queue is not empty, the quantity of work in the queue can only increase by at most $b_m + D \cdot R_m$, which is independent of the length of the interval. Thus the quantity of work in the queue for m is never more than $b_m + D \cdot R_m$.

Since there exists $d \in \mathbb{R}_+ - \{0\}$ such that all tasks last more than d, there are at most $(b_m + D \cdot R_m)/d$ tasks in Q_m. □

Lemma 6 *For any $\sigma \in G$, there is a time T after which at every request for a job J, the union of the set of tasks we put in Q at that time and of the set of task instances from task types of J that have been directly started as a "fill up" since previous request for J (we say the instances of that union are selected for that latest request), contains exactly one instance of each task type of J.*

Proof. Base case: for the initial tasks this is trivially true, since they are queued as soon as the job is requested.

Inductive case: now suppose there exists a time t_{l-1} after which when a job of type J is requested, a task instance of each type of rank $l - 1$ is selected for that request. Since the policy chooses the oldest instance first, the difference of indices between the job instance from which the task comes, and the job request at which it is selected is bounded. And after it is queued (if it is), it will be served in bounded time, during which the number of requests for J is also bounded. Thus any requested task of rank $l - 1$ is served after a bounded number of requests for J. Let N_{l-1} be that number.

Let a be a task type of rank l from job type J. Then an instance of a has to become eligible before N_{l-1} requests for J after its own. This we can write $n_r(t) - N_{l-1} \leq n_e(t)$, where $n_r(t)$ is the number of requested J at to time t and $n_e(t)$ is the number of tasks instances form a made eligible since the beginning.

Let also $n_s(t)$ be the number of selected instances of a until time $t \in \mathbb{R}_+$. Then there are two possibilities:

- Either the number of eligible but not selected instances of a stays smaller than N_{l-1} ($n_e(t) - n_s(t) \leq N_{l-1}$), so that any of them has to be selected before N_{l-1} requests for J. Thus $n_s(t) + N_{l-1} \geq n_e(t) \geq n_r(t) - N_{l-1}$, which gives $n_r(t) - n_s(t) \leq 2N_{l-1}$. Hence $n_r(t) - n_s(t)$ never decreases (we select no more than requested), therefore it is eventually constant, and thus eventually we queue an instance of a every time a J is requested.
- Or $n_e(t) - n_s(t)$ can go above $N_{l-1} + 1$. Then, suppose it is true at a time t_0.

Then we prove that $n_e(t) - n_s(t)$ can never go to zero after t_0: indeed, $n_s(t)$ increases only when $n_r(t)$ does, so that means that $\forall t \geq t_0, n_s(t) - n_s(t_0) \leq n_r(t) - n_r(t_0)$, so $n_s(t) \leq n_r(t) - n_r(t_0) + n_s(t_0) \leq n_r(t) - n_r(t_0) + n_e(t_0) - N_{l-1} - 1 \leq n_e(t) - n_r(t_0) + n_e(t_0) - 1$, and thus $n_e(t) - n_s(t) \geq n_r(t_0) - n_e(t_0) + 1 > 1$.
So after t_0 there are always eligible instances of a that are not yet selected, so after t_0, when a job is requested, an instance of each of its task of rank l is selected.

In both cases, the fact that eventually at every request we queue an instance of every task type of rank $l-1$ implies that we eventually also select one instance for each type of rank l. That means, as there are only finitely many ranks, that eventually it is true for any rank. □

Conclusion (the residue is thus bounded): Lemma 6 states that eventually we select, at each request for a job type J, one task instance per task type from J, so that the total amount of task instances of each type in the pool cannot increase anymore. Hence the residue is the union of the pool and of the queues, which are also bounded (Lemma 5), thus the residue is bounded. That proves the first part of Thm. 3.

If we look at the proof, we notice that the bound on the queues and the maximal time at which the property of Lemma 6 comes true depend on the request stream σ only through the constant b that characterizes its admissibility. Thus if G is uniformly admissible, then the residue is uniformly bounded, which proves the second part of the theorem. □