



HAL
open science

An MDP-Based Solution for the Energy Minimization of Non-Clairvoyant Hard Real-Time Systems

Bruno Gaujal, Alain Girault, Stéphan Plassart

► **To cite this version:**

Bruno Gaujal, Alain Girault, Stéphan Plassart. An MDP-Based Solution for the Energy Minimization of Non-Clairvoyant Hard Real-Time Systems. 2024. hal-02371742v3

HAL Id: hal-02371742

<https://inria.hal.science/hal-02371742v3>

Preprint submitted on 4 Nov 2024

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.

An MDP-Based Solution for the Energy Minimization of Non-Clairvoyant Hard Real-Time Systems

Bruno Gaujal¹, Alain Girault¹, Stéphan Plassart²

¹Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000,
Grenoble, France.

²Université Savoie Mont Blanc, LISTIC, 74044, Annecy, France.

Contributing authors: bruno.gaujal@inria.fr; alain.girault@inria.fr;
stephan.plassart@univ-smb.fr;

Abstract

We address the problem of scheduling a possibly infinite sequence of hard real-time jobs on a single-core processor, with the dual goal that (1) all the jobs must finish before their deadline, and (2) the energy consumption must be minimized. The decision variable is the speed of the processor at each time instant, to be chosen from a finite set of available speeds. Our goal is to design a speed policy in charge of deciding, at each time instant, the speed of the processor in function of the job characteristics. We focus more specifically on the non-clairvoyant case, meaning that the actual size of the jobs (the amount of work to be done to complete the job) is unknown when they are released, but its probability distribution is known, and of course, the maximal size is known too. In this context, we propose two new speed policies, the optimal solution of a Markov Decision Process (called **MOSP**), and a heuristic speed policy called Expected Load (**EL**), obtained by adapting the classical policy Optimal Available (**OA**) to jobs with random sizes. Our **MOSP** algorithm is split in two phases: the first phase is *offline* — it computes the optimal processor speed for each possible system state — while the second phase is *online* — it retrieves the speed to apply to the processor thanks to a table lookup. Compared with the existing speed policies from the literature, **MOSP** achieves the optimal energy consumption but at the cost of a significant state space size. In contrast, **EL** achieves an energy consumption that is, on average, close to the optimal one obtained with **MOSP**, but at almost no cost in terms of state space.

Keywords: Hard Real-Time Systems, Energy Optimization, Markov Decision Process, Dynamic Voltage and Frequency Scaling.

1 Introduction

The **single-core hard real-time energy minimization problem** involves choosing, at each decision time, the frequency of the processor to execute the current jobs, such that all jobs meet their deadline and such that the total energy consumed by the processor is minimized. Hard real-time constraints and energy minimization are difficult to combine because the former require to be very conservative by only considering the worst cases, while the latter would benefit greatly from relaxing strict deadlines for job completions. Nevertheless, several approaches have been proposed to tackle the hard real-time energy minimization problem under several assumptions on the processor and on the jobs to be executed. For instance, Chen and Kuo have assembled a state-of-the-art review of energy-efficient scheduling methods for real-time systems on Dynamic Voltage Scaling (DVS) platforms for uni- and multi-processor [1].

The most classical case is the **offline case**, where the number of jobs is finite, and all their characteristics are known before choosing the processor speeds. The finite set of jobs is thus $\{J_i\}_{i \in \{1, \dots, N\}}$, each job J_i being defined by a triple (r_i, w_i, d_i) with a *release time* r_i , a *size* or *workload* w_i (scaled so that a job of size 1 completes in 1 time unit on a processor working at speed 1), and a *relative deadline* d_i . Based on this complete knowledge, and denoting $H = \max_{i=1, \dots, N}(r_i + d_i)$ the time horizon, the processor must select, at each time t in $[0, H]$: (1) which job to execute among all the jobs present in the system at time t , and (2) its operating speed $s(t)$ among a given set of admissible speeds \mathcal{S} , such that all jobs are completed before their deadline and the total energy consumption of the processor over the interval $[0, H]$ is minimized. The policy used to decide which job to execute is the *scheduling policy*, while the policy used to decide the processor speed is the *speed policy*. The set \mathcal{S} of admissible speeds is the set of available speeds on the target processor, and the energy consumption is only a function of the processor speed chosen in \mathcal{S} . The offline case has been first solved by Yao et al. in [2].

The **online case** differs in that, at any time t , only the jobs released before t or at t are known. We further distinguish the **clairvoyant** online case where the characteristics of each job J_i (its size w_i and its deadline d_i) are revealed at the release time r_i of J_i , and the **non clairvoyant** online case where only d_i is revealed when J_i is released, but the actual size w_i is only known when J_i finishes.

The **clairvoyant online case** has been first investigated by Yao et al. who proposed Optimal Available (OA, a greedy speed policy), and Average Rate (AVR, a proportional fair speed policy) in [2]. These speed policies have been compared with the optimal offline solution by Bansal et al. in [3], who also computed their competitive ratio. Their feasibility has been studied for the first time in [4]. Further improvements have been proposed by Li and Yao in [5, 6], and by Bansal et al. in [3]. Other clairvoyant online solutions exist in the literature: for example, Zhong et al. [7] have proposed a time variant voltage scaling algorithm to solve the energy minimization problem for

general tasks with DVS. Moreover, several solutions can be seen as intermediate in the sense that they lie between the online and offline cases. This is the case, for instance, of [8] and [9], which consider only periodic tasks where the execution times are known at release time (*i.e.*, the clairvoyant case).

In the **non-clairvoyant online case**, only the maximal size of the jobs is known. Estimating the maximal size of a real-time job is a problem related to the Worst-Case Execution Time (WCET) estimation problem. This difficult problem has attracted the attention of many researchers in the hard real-time community. Schematically, two types of approaches exist, static analysis and measurement methods.

Static analysis methods over-approximate the size (or WCET) of a piece of code by analyzing it offline, of course taking into account the hardware it will run on [10–14]. This is extremely complex because modern processors exhibit many features that make them more efficient but at the price of more non-determinism, such as caches, pipe-lines, branch prediction, out-of-order execution, and so on. This explains why over-approximations have to be made.

Measurement methods under-approximate the size (or WCET) of a piece of code by repeatedly running it onto its hardware under varied initial conditions, and then measuring the job size for each run. The maximal job size is then obtained by taking the maximum over all the runs, or by using more complex statistical methods [15], for instance Extreme Value Theory. By construction, it is an under-approximation because all possible cases might not be encountered during the necessarily limited number of runs. Conversely, the minimal job size is the minimum over all runs.

Besides the maximal and minimal job size, both static analysis and measurement-based methods also retrieve an (approximate) *distribution* of the job size. We will provide an example of such a distribution in Fig. 8.

The **non-clairvoyant online case** has been first investigated by Lorch and Smith in [16, 17] for a *single job*: they have proposed the Processor Acceleration to Conserve Energy method (PACE), which provides a closed form solution. That is, the processor speed is computed as a function of the job size distribution. The drawback of PACE is that this closed form is only valid for a single job, when the energy consumption rate is proportional to some power of the speed (*e.g.*, the cube of the speed), and for continuous processor speeds. Barnett [18] has extended this study by considering the bi-criteria problem of minimizing the expected execution time under a hard energy budget in addition to minimizing the expected energy consumption under a hard execution deadline, still for a single job and continuous processor speeds.

To the best of our knowledge, PACE has been extended in three directions. First, Xu et al. have studied the practical case with discrete speeds, no assumption on the power function, non-null processor idle power, and non-null speed change overhead [19]. They have proposed Practical PACE (PPACE), a fully polynomial time ε -optimal approximation algorithm in the single job case, which performs a time discretization of the speed selection.

Second, Bini and Scordino have proposed an optimal solution in the particular case where the processor uses only two speeds to execute the job, taking into account the speed change overhead [20].

Third, the single job assumption of PACE has been relaxed in another series of papers. Considering several jobs gives rise to the distinction between *sporadic* and *periodic* jobs. In the periodic case, several papers have focused on the constrained case known as the *frame based multi-task model* [21, 22], where all the jobs are periodic, with their deadline equal to their period, and share the *same period*.

In the frame based context, Zhang et al. have proposed the Global Optimal Procrastinating Dynamic Voltage Scaling algorithm (Global OP-DVS) [23], while Xu et al. have proposed a Hybrid Dynamic Voltage Scaling algorithm (HDVS), hybrid in the sense that it addresses both intra-task DVS and inter-task DVS [24, 25]. The HDVS algorithm is a fully polynomial time ε -optimal approximation. Bandari et al. [26] have investigated the specific case of a frame composed of two specific alternating periodic tasks where they use both DVS and Dynamic Modulation Scaling (DMS) techniques in a wireless embedded system using probabilistic computation and communication workload models with discrete speeds. The drawback is that the frame based model can be restrictive: modern real-time systems exhibit a combination of sporadic and periodic jobs with *significantly different periods*, typically ranging between 1 ms and $1,000\text{ ms}$. The former cannot be captured at all in a frame based model, while for the latter, a decomposition of the hyper-period schedule into frames would result in too many frames to be practical, and, more importantly, would be sub-optimal in terms of energy consumption.

In the general case, Lorch and Smith have proposed a multi-job extension of PACE in [17] by considering each job *independently* and adding the speeds obtained for each job in isolation. However, handling each job independently and summing the resulting processor speeds results in a sub-optimal speed selection. Still in the general case, Yuan and Nahrstedt have proposed to relax the hard constraint by considering soft real-time task and propose heuristics in the multimedia context to build the speed schedule [27].

Finally, several *heuristics* have been proposed in the literature to deal with particular cases. For example, Pillai et al. [28] have proposed two algorithms, the cycle-conserving RT-DVS and the look-ahead RT-DVS, both of them considering only periodic tasks with variable workloads. The first one is dominated by OA and the second one is dominated by PACE. Aydin et al. [29] have improved the solution of [28] by proposing the GDRA and AGR algorithms, which use speed adjustment algorithms based on the expected workload. Again, both of them consider only periodic tasks and do not offer any optimality guarantees.

To summarize, many online heuristics have been proposed for the non-clairvoyant online problem (*e.g.*, [29]), some of them with competitive ratios (*e.g.*, [2, 3, 5]). But no optimal solution has been proposed so far in the general hard real-time case.

In this paper, we propose a solution that is optimal in expectation for the **non-clairvoyant online problem in the general hard real-time case**. To solve this problem, we build a Markov Decision Process (MDP) that computes the *optimal speed* of the processor at each time instant, to minimize the *expected total energy consumption* while guaranteeing the completion of all jobs before their deadline. To achieve this, we design a finite state space for the evolution of the system and compute the transition probabilities from one state to another, based on the distributions of the

job characteristics. The combinatorial cost of this construction is significant, but since the computations of the transition probabilities and of the optimal speed policy are done offline and only once, we claim that it does not hamper the online usability of the resulting speed policy for embedded systems.

Our algorithm consists of two phases, one offline and one online. During the offline phase, we compute the processor speed for each possible state of the system that we could encounter during the online phase. The state space is bounded by three parameters — the maximal inter-arrival time, the maximal deadline, and the maximal size — and is, therefore, *finite*. To compute offline these speeds, three steps are necessary:

1. Construction of the Markov Decision Process (MDP) under constraints that represents the system evolution. A key contribution is that we simplify the resolution of this problem by transforming an MDP under constraints problem into an MDP without constraints one, where the constraints are embedded inside the MDP parameters.
2. Construction of the transition probability matrix for going from one state of the MDP to another one. This matrix can be obtained either by simulation (Sec. 7.4), or can be built explicitly when we know the distribution of the parameters for each task of the system (Appendix A).
3. Resolution of the MDP problem thanks to the *Value Iteration* algorithm (VI), known to be the most efficient in practice. Its output is a table that associates, to each state of the MDP, the optimal speed to be applied to the processor in this state.

During the online phase, we apply the processor speed determined in the offline phase for the current state, with a table lookup (Sec. 6.2).

The paper is organized as follows. We first formalize the problem in Sec. 2. We present in details the two state of the art speed policies OA and PACE in Sec. 3 and we provide two counter examples showing that they are not optimal. Then we present our MDP-based solution in Sec. 4 followed by a heuristic speed policy called Expected Load (EL) in Sec. 5. EL is inspired by both our optimal solution of a Markov Decision Process (called MOSP, for “MDP Optimal Speed Policy”) and OA, with the goal to be as efficient as possible as MOSP to the extent possible but as lightweight as OA. We provide in Sec. 6 the details of the implementation of our speed policies, before performing numerical experimentation on synthetic and real-life benchmarks in Sec. 7. In particular we compare the two state of the art speed policies (OA and PACE), our optimal speed policy (MOSP), and our heuristic one (EL). In Sec. 8, we discuss how two assumptions of our model can be relaxed, related to the cost of context switch between real-time jobs and the cost of speed change for the processor. Finally, we give concluding remarks in Sec. 9.

Our original contributions are: the counter-examples for OA and PACE proving their sub-optimality, the MDP formulation and the MOSP optimal speed policy for the non-clairvoyant online problem in the general hard real-time case, the EL heuristics, the implementation of MOSP by adapting the VI algorithm, the experimental evaluation, and the extensions to handle the costs of context switches and of speed changes.

2 Formalization

2.1 System Model

Our real-time system consists of a processor and an infinite sequence of real-time jobs. We define these notions in the following subsections.

2.1.1 Processor model

We consider a single-core processor equipped with Dynamic Voltage and Frequency Scaling (DVFS) capabilities, characterized by a finite set \mathcal{S} of available speeds ($\mathcal{S} \subset \mathbb{N}$), with a minimum speed denoted s_{\min} and a maximum speed denoted s_{\max} . By convention, $s_{\min} = 0$. The set of available speeds is denoted \mathcal{S} , which is a subset of $\{0, \dots, s_{\max}\}$ because not all values between 0 and s_{\max} may be available.

For simplicity, we assume that the cost of speed change is null. Our approach can be generalized with a non-null speed change cost by modifying the power function to include the speed change costs as in [30]. This will be discussed in Sec. 8.2.

The power consumed at any time t by the processor running at speed $s(t)$ is denoted $F(s(t))$. We make no assumption on the function F ; in particular, we do not assume that it is convex. Indeed, the classical power model is the following [19]: If $s \neq 0$, then $F(s) = P_{leak} + P_{sc} + P_{dyn}(s)$ while if $s = 0$, then $F(s) = P_{idle}$. Here, P_{leak} is the leakage power, P_{sc} is the short circuit power, and $P_{dyn}(s)$ is the dynamic power, proportional to s^3 , and P_{idle} is the power dissipated when the processor is idle. In that case, $F(s)$ is a non convex function of s .

In the rest of the paper, we will use several models for the power function depending on the cases. The counterexamples in Section 3 use $F(s) = s^2$ because it allows the worst case to be written in simple closed form. The experimental part (Section 7) uses $F(s) = s^3$, which is more classical in the literature. The strong point is that the general algorithm presented in Section 4 is optimal for *any* power function.

2.1.2 Jobs

We adopt the most general model of preemptive real-time jobs, defined by Def. 1. Jobs can be produced by one or more real-time tasks (periodic tasks for instance) or can be independent.

Definition 1 (Job). *A real-time job J_i is characterized by the triplet (τ_i, w_i, d_i) , where:*

- $\tau_i \in \mathbb{N}$ is the random **inter-arrival time** between J_i and J_{i-1} , the distribution of which has a finite memory L .
- $w_i \in \mathbb{N}$ is the random **size** of job i , i.e., the amount of work to be done to complete the job, upper-bounded by the maximal size W ;
- $d_i \in \mathbb{N}^*$ is the random **relative deadline**, upper-bounded by Δ .

Here are some important remarks about the job features:

- The inter-arrival time distribution has a finite memory L . This means that, as soon as the previous job arrived more than L units of time ago, this previous job arrival time is “forgotten” and does not affect the next arrival. In mathematical

terms,

$$\forall J_i = (\tau_i, w_i, d_i), t \geq L, t' \geq 0, \quad \mathbb{P}(\tau_i = t+t' | \tau_i \geq t) = \mathbb{P}(\tau_i = L+t' | \tau_i \geq L). \quad (1)$$

An example with a finite memory is the geometric distribution used in Sec. 2.1.5 the memory size of which is $L = 0$. Another example is when the inter-arrival time has a finite support $[0, L]$: In this case, the memory size is the bound on the support (hence L here).

- The size w_i of the job is not to be confused with its execution time (ET, in s), nor should the maximal size be confused with the worst-case execution time (WCET, in s). The size w_i is the number of elementary operations that must be done to complete the job. The relation with the execution time is $ET = w/s$, where s is the speed of the processor. When the speed of the processor is constant, it is common to arbitrarily consider that it is equal to 1. In such a case, the size and the ET are equal and one can be taken for the other (even though their physical dimensions differ). Here, however, the speed of the processor is changing over time, so the confusion must be avoided.
- The release time of the jobs. Actually, the release time r_i of any job $J_i = (\tau_i, w_i, d_i)$ can be computed the following equation:

$$r_0 = 0 \quad \text{and} \quad \forall i > 0, r_i = r_{i-1} + \tau_i, \quad (2)$$

which implies that $r_0 = 0$, $r_1 = \tau_1$, $r_2 = \tau_1 + \tau_2$, and so on.

Finally, we assume that all the time quantities (arrival times, execution times, deadlines) belong to \mathbb{N} . The *characteristic time* of the stream of jobs, denoted ξ , is the greatest duration such that the jobs' features (deadlines and arrival times) are multiple of ξ . This discretization is classical in real-time systems and reasonable. It will play an essential role in our design of a discrete-time scheduler. The time unit ξ only depends on the jobs' features, and is unrelated to the cycle time of the processor, which can be several orders of magnitude smaller.

2.1.3 Discrete time non-clairvoyant scheduler

The role of the scheduler is to choose which job to execute among all currently pending jobs and the current speed for the processor. The scheduler acts at discrete times compatible with the time unit ξ used to quantify the job features, *i.e.*, it acts at every time $t\xi$, with $t \in \mathbb{N}$. In the sequel, we will forget about ξ and simply take as the time scale the set of integers \mathbb{N} .

We further assume that jobs are *preemptive* with a null switching context time. Our scheduler constructions given in Sec. 6 can be modified to relax this assumption, as we will see in Sec. 8.

The crucial point in the design of our scheduler is the fact that it is *non-clairvoyant*: Upon the release time of job J_i , its size w_i is not known to the scheduler and remains unknown until J_i actually terminates. In other words, J_i is characterized by the triplet (τ_i, w_i, d_i) but, when it is released, the scheduler is only aware of its maximal size W

and its relative deadline d_i upper-bounded by Δ (Δ being identical for all the jobs). The release time of the next job J_{i+1} is also unknown at this point.

2.1.4 Dynamic evolution of the system

The dynamic evolution of the system proceeds as follows. At time 0, no job is present and the state is empty. The first job $J_1 = (\tau_1, w_1, d_1)$ arrives at time $\tau_1 > 0$. The processor chooses one of its available speeds $s \in \mathcal{S}$ to start executing J_1 . In this situation, two cases are possible:

1. Either the chosen speed is sufficient to finish executing J_1 during the current time interval $[\tau_1, \tau_1 + 1]$: in this case the system state at $\tau_1 + 1$ is once more empty, and the next job arrival will be treated like J_1 .
2. Or the chosen speed is insufficient to finish executing J_1 during the current time interval $[\tau_1, \tau_1 + 1]$: in this case several things can occur at $\tau_1 + 1$:
 - (a) If no job arrives at $\tau_1 + 1$, then the processor again chooses one speed $s \in \mathcal{S}$ to continue executing J_1 .
 - (b) If a new job $J_2 = (1, w_2, d_2)$ arrives at $\tau_1 + 1$, then the two jobs J_1 and J_2 are present in the system, and some *priority rule* must be enforced to decide whether the processor must execute J_1 or J_2 first. This issue is classically referred to as the *scheduling policy*, and we will deal with it later, in Sec. 2.2. In any case, the processor again uses one of its available speeds $s \in \mathcal{S}$ to execute either J_1 or J_2 .

In all cases, the speed chosen by the processor at time $\tau_1 + 1$ can be the same speed as before or a different one. This process repeats at the next instant $\tau_1 + 2$, and so on and so forth.

At any instant t where at least one job is present in the system, several things can occur depending on the size w_i of the job J_i having the higher priority (whatever the priority rule enforced by the processor) and on the speed $s(t)$ chosen by the processor at time t :

- If $s(t) < w_i$ and J_i 's deadline is not reached, then J_i is not finished at time $t + 1$ and there still remain $w_i - s(t)$ units of work to be performed on J_i at the next time instant $t + 1$. If its deadline is reached, then J_i is discarded.
- If $s(t) = w_i$, then J_i finishes *exactly* at $t + 1$, so the processor will start executing another job (if there is one) at the next time instant $t + 1$.
- If $s(t) > w_i$, then J_i finishes *within* the current time instant, so the processor either becomes idle (if no other job is present in the system), or starts executing the next available job J_k according to the priority rule. In the latter case, the same reasoning takes place, but this time depending on J_k 's size w_k and on the remaining available amount of computing $s(t) - w_i$.

A final remark is that our system model only refers to *jobs*, while real-time systems classically consider *tasks* that generate jobs. Our system model being extremely general, our jobs can be generated by *sporadic* tasks, by *periodic* tasks, or by periodic tasks enriched with an *arrival probability*. All the examples provided in the paper are oblivious of tasks.

2.1.5 Probability distributions on arrival times

Our first example is an infinite sequence of sporadic jobs $\{J_i\}_{i=1,\dots}$ where the sequence of inter-arrival times is $0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, \dots$. It is depicted in Fig. 1 up to $t = 16$. We do not consider tasks in our model (see Sec. 2.1.2), but the reader can notice that the sequence of jobs of Fig. 1 could be generated by two periodic tasks with respective periods 1 and 3.

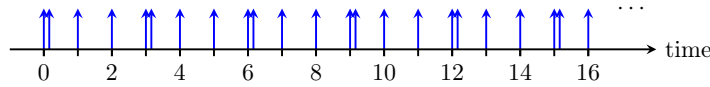


Fig. 1: An infinite sequence of sporadic jobs. Each upward arrow represents the arrival of a new job.

Our second example is an infinite sequence of sporadic jobs $\{J_i\}_{i=1,\dots}$ where the sequence of inter-arrival times is $0, 0, 1, 2, 5, 1, 3, 1, 1, 1, 0, \dots$. It is depicted in Fig. 2 up to $t = 16$. Again, we do not consider tasks in our model, but the reader can notice that the sequence of jobs of Fig. 2 could be generated by two periodic tasks with respective periods 1 and 3 and with arrival probabilities both equal to 0.5.

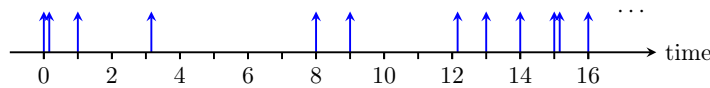


Fig. 2: A second infinite sequence of sporadic jobs. Each upward arrow represents the arrival of a new job.

An alternative to arrival probabilities would be to specify a *minimum inter-arrival time*, an approach sometimes adopted in the real-time literature. A first remark is that these two models, with arrival probabilities and with minimum inter-arrival times, are not comparable, as exemplified by Fig. 2. A second remark is that the second task model prevents the inter-arrival times from being independent and identically distributed random variables (i.i.d.), a useful property when one wants to take into account statistical properties of the jobs, which is our case in this paper.

2.1.6 Probability distribution on sizes

Our model can also take into account probability distributions on sizes. But recall that we are in the *non-clairvoyant* case, *i.e.*, the size of a job is only known when this job actually finishes. We consider in this paper that *statistical information* is known on the size of the jobs, and we adopt the same model as the PACE algorithm [16] where we know the probability distribution on the size.

Our third example is an infinite sequence of sporadic jobs $\{J_i\}_{i=1,\dots}$ the size of which follows a uniform probability distribution law in the interval $[1, W]$. This is

illustrated in Fig. 3 up to $t = 16$. Each vertical arrow represents the arrival of a job, with a vertical span proportional to its size (unknown by the processor until the job actually terminates). As in the two previous examples, this sequence of jobs could have been generated by periodic tasks with arrival probabilities and with uniform size probabilities in the range $[1, 4]$. Of course, the sizes can follow more complex probability distribution laws than the uniform law.

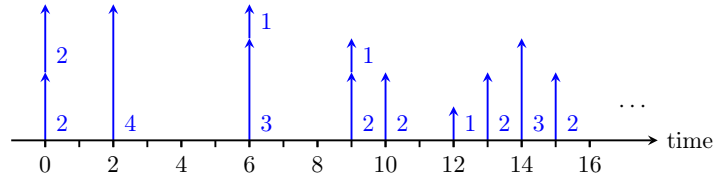


Fig. 3: Instantaneous amount of work resulting from an infinite sequence of sporadic jobs. Each upward vertical arrow represents the arrival of a job, with a vertical span proportional to its size.

2.1.7 Probability distribution on deadlines

Our model can also take into account probability distributions on deadlines. Our fourth example is an infinite sequence of sporadic jobs $\{J_1 = (0, 1, 1), J_2 = (2, 4, 4), J_3 = (1, 2, 2), J_4 = (3, 2, 2), J_5 = (1, 3, 3), J_6 = (2, 2, 2), J_7 = (5, 2, 2), J_8 = (1, 3, 3), J_9 = (1, 1, 1), \dots\}$. This is illustrated in Fig. 4.

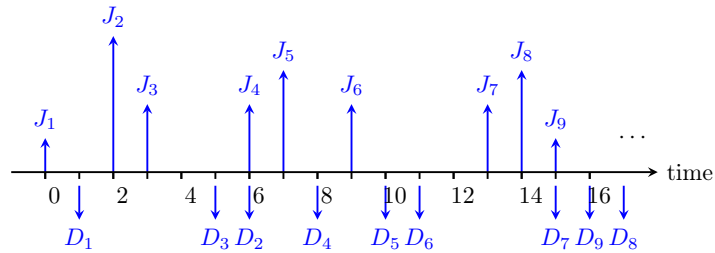


Fig. 4: An infinite sequence of sporadic jobs. Each upward vertical arrow represents the arrival of a job J_i , with a vertical span proportional to its size. Each downward vertical arrow represents the absolute deadline D_i of job J_i .

2.1.8 Bounds on the distribution supports

As mentioned in Sec. 2.1.2, the size of the jobs, and the relative deadlines all have bounded support (resp. W, Δ) and the inter-arrival distribution has a finite memory L (see Sec. 2.1.2.)

We further impose additional bounds to guarantee that the *backlog* (*i.e.*, the amount of work present in the processor) remains bounded. This is needed to prove the feasibility of the speed policies presented later. To achieve this, one possibility is to impose a strictly positive lower-bound on the inter-arrival times. Let T_{\min} denote the lower-bound on the inter-arrival times: $T_{\min} = \min_i \tau_i$. The constraint is thus $T_{\min} > 0$. In that case, at most W work can arrive every T_{\min} instants, and the total amount of work present at any time cannot exceed $W \Delta / T_{\min}$.

However, assuming $T_{\min} > 0$ is too strong a constraint in the sense that the cases that comply with it are very restricted. For instance, all the examples in Figs. 1, 2, and 3 have simultaneous job arrivals, meaning $T_{\min} = 0$. Actually, as soon as the individual jobs are released from several periodic tasks whose periods do not have a common divisor, then $T_{\min} = 0$. For this reason, we will instead impose the constraint that the number of jobs that can arrive simultaneously is upper-bounded by Υ . When the jobs come from a finite number of tasks, then this upper-bound Υ is obviously the number of tasks. This implies that the number of jobs present in the system is upper-bounded by $\Upsilon \Delta$, while the work backlog is upper-bounded by $\Upsilon W \Delta$. We further introduce the *buffer size* of the processor, denoted B . The number of jobs present at any time in the processor is therefore upper-bounded by $\min\{B, \Upsilon \Delta\}$.

2.2 Problem Statement

We consider the following problem: Given an infinite sequence of real-time jobs (Definition 1) arriving on a single-core processor equipped with DVFS, such that:

- the available set of speeds \mathcal{S} is finite ($\mathcal{S} \subset \mathbb{N}$),
- the jobs' inter-arrival times distribution has a finite memory L ,
- the maximum of the jobs' sizes is W ,
- the maximum of the jobs' deadlines is Δ , and
- the maximal number of jobs that can arrive simultaneously is Υ ,

we want to design a discrete time scheduler that chooses (1) which job to execute, and (2) a processor speed such that all the jobs meet their deadline and the power consumption is minimal.

Two questions arise to solve this problem: The first one is the choice by the scheduler of the *scheduling policy* (as stated in Sec. 2.1.4), which determines in what *order* the jobs shall be executed. This question will be addressed in Sec. 2.2.1. The second question is the choice by the controller of the *speed policy*, which determines the *processor speed* at each instant to execute the jobs present in the system. This question is the main topic of this article.

2.2.1 Feasibility

Definition 2 (Feasibility). *A set of jobs is **feasible** if there exist a **scheduling policy** and a **speed policy** such that no job deadline is missed.*

In the rest of the paper, we use the EDF (Earliest Deadline First) scheduling policy [31]. A key advantage of EDF is that it is optimal for *feasibility* in our real-time systems model, *i.e.*, with sporadic and preemptive jobs and with a null context switching time. The optimality of EDF implies that, if a set of jobs is feasible for

a given scheduling policy and a given speed policy, then it is also feasible when the scheduling policy is replaced by EDF while the speed policy remains the same. This has been proved in the PhD of S. Plassart [32]–Appendix A.1 for the same system model as the one used in this paper.

Our job model does not assume that the inter-arrival time is *lower-bounded*. Under this job model, an arbitrary number of jobs may arrive at any given instant, resulting in an infeasible system (unless $s_{\max} = +\infty$). This is unrealistic, of course, and we will state in Sec. 4.3 a proposition relating s_{\max} and W to guarantee that the system being studied is feasible.

2.2.2 Objective function

Since most real-time embedded systems are life-long, we compute the *average power consumption* c over an *infinite horizon* as:

$$c = \lim_{T \rightarrow +\infty} \frac{1}{T} \int_{t=0}^T F(s(t)) dt. \quad (3)$$

Our goal is to execute all the jobs before their deadline while minimizing the average power consumption c . This goal is achieved jointly by the scheduling policy and the speed policy. In this paper, we solve this constrained optimization problem with an *online speed policy*, which is computed *offline*. At any time t , the processor does not know the future job releases, nor the exact duration of jobs currently present in the system (*non-clairvoyant* case). Instead of investigating an adversarial model (worst possible future arrivals and worst job durations), we focus on a *statistical* approach. The variables τ_i, w_i, d_i are viewed as *random variables* following a known joint *probability distribution*. Such a distribution can be either given by the system designer or statistically estimated from numerous job samples.

In Sec. 3, we study the two state of the art speed policies OA [2] and PACE [16]. PACE is a *local* policy, in the sense that the speed is computed *individually* for each job, and then *summed* over all the jobs active at any given time. Since our model can allow multiple jobs to be simultaneously present in the system, we use a processor sharing scheduling policy. OA is a *global* policy, in the sense that the speed is computed from the *cumulated work* of all the jobs present at a given time in the system (like the optimal one MOSP introduced in Sec. 4 and the heuristic one EL introduced in Sec. 5). For this reason, OA, MOSP, and EL also rely on EDF when several jobs are active at the same time.

PACE and OA are, in fact, both optimal, but in respective contexts that are restrictive w.r.t. the general problem studied in this paper: PACE is optimal when there is at most one job present in the system at any time, while OA is optimal when the jobs sizes are all maximal and the inter-arrival times are either equal to 0 or larger than Δ . Moreover, the optimality of both PACE and OA relies on the convexity of the power function $F(s(t))$, while in practice the power is not convex due to leakage currents (see Sec. 2.1.1). In contrast, our optimal solution does not rely on this convexity assumption.

2.2.3 State Space

The information available to the processor to choose its speed can be split into two parts. The static part consists of the joint distribution of the execution times, release times, and deadlines of the jobs. The dynamic part changes over time; it will be called the *system state* of the system in the following.

Definition 3 (System state). *The **system state** at time t is composed of two elements:*

- \mathbf{x}_t : the **list of current jobs**, sorted by their deadlines (jobs that have the same deadline are sorted by increasing release time). In this list, each job J_i is represented by a pair (e_i, d_i) where e_i is the work quantity already executed by the processor on job J_i and d_i is the relative deadline of job J_i .
- ℓ_t : the **time elapsed** since the latest job arrival.

The state space is $\mathcal{X} \times \{1, \dots, L\}$, where \mathcal{X} is the set of all possible lists of current jobs and L is the maximal inter-arrival time between any two jobs. The state space is *finite* because the job characteristics are upper-bounded by L , W , and Δ . The state evolution from (\mathbf{x}_t, ℓ_t) to $(\mathbf{x}_{t+1}, \ell_{t+1})$ is driven by two processes:

1. One or several jobs may be *completed* in $(t, t + 1]$ by the processor running at its current speed $s(t)$. These jobs must be removed from \mathbf{x}_{t+1} . For example, in Fig. 5, part of job J_1 and job J_2 are executed during the interval $[2, 3]$, they should be removed from the state at time step 3.
2. Some new jobs may *arrive* at time $t + 1$. These jobs must be inserted in \mathbf{x}_{t+1} , sorted according to the EDF policy, that is, by increasing order of their relative deadline d_i . In case of tie, the jobs are ordered by decreasing order of their executed work e_i .

Both scheduling decisions and speed change decisions are made at the same instant, at integer time step t . At this time step t , we have an ordered list of job ready to be executed (the list of jobs \mathbf{x}_t) and a speed to apply to the processor $s(t)$.

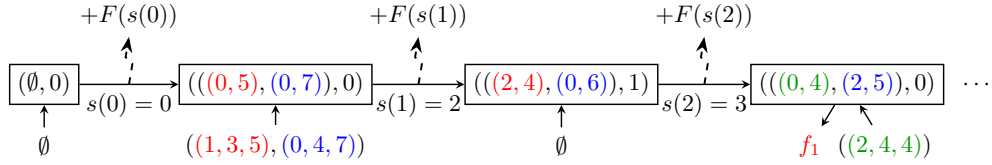


Fig. 5: Illustration of a state evolution. The system is initially empty at $t = 0$. Job $J_1 = (1, 3, 5)$ in red arrives at $t = 1$ and is completely executed at $t = 3$ (finish time f_1). Job $J_2 = (0, 4, 7)$ in blue arrives at $t = 1$ and is partially executed at $t = 3$. Job $J_3 = (2, 4, 4)$ in green arrives at $t = 3$ and because its relative deadline is earlier than that of J_2 , it is inserted at the head of the sorted list of jobs (EDF policy). For each job, only the work quantity already executed on it and its relative deadline are stored in the system state (non-clairvoyant case).

The state evolution is essential in the construction of the MDP. Let us introduce a simple example to illustrate how we go from one state to another (Fig. 5). The system

starts at $t = 0$, and then at $t = 1$ two jobs arrive, J_1 and J_2 , with the following characteristics: $J_1 = (\tau_1 = 1, w_1 \stackrel{?}{=} 3 \leq 6, d_1 = 5)$ and $J_2 = (\tau_2 = 0, w_2 \stackrel{?}{=} 4 \leq 6, d_2 = 7)$. We write “ $w_i \stackrel{?}{=} p \leq q$ ” to specify that, at the release time of the job J_i , the processor knows that J_i ’s worst case size is equal to q , but regarding its size w_i , only its distribution is known. The exact value of J_i ’s size will be “discovered” by the processor when J_i finishes. Accordingly, the size w_i is not part of the system state. With these two jobs, the system state at $t = 1$ is therefore: $(\mathbf{x}_1 = ((e_1 = 0, d_1 = 5), (e_2 = 0, d_2 = 7)), \ell_1 = 0)$, where the jobs are sorted according to the EDF policy. In our example, J_1 is inserted first since its deadline is earlier than J_2 ’s deadline.

Suppose that, at time 1, we decide to set the processor speed to $s(1) = 2$ and that no new job arrives at time 2. The system state at $t = 2$ is therefore: $(\mathbf{x}_2 = ((e_1 = 2, d_1 = 4), (e_2 = 0, d_2 = 6)), \ell_2 = 1)$, where the two units of work executed by the processor have been devoted to J_1 ($e_1 = 2$) because it is more urgent than J_2 due to EDF, the two relative deadlines d_1 and d_2 have both been decreased by one time unit, and ℓ has been increased by one time unit.

Suppose now that, at time 2, we decide to set the processor speed to $s(2) = 3$, and that a third job J_3 arrives at time 3 with the following characteristics: $J_3 = (\tau_3 = 2, w_3 \stackrel{?}{=} 4 \leq 5, d_3 = 4)$. The system state at $t = 3$ is therefore: $(\mathbf{x}_3 = ((e_3 = 0, d_3 = 4), (e_2 = 2, d_2 = 5)), \ell_3 = 0)$, where the three units of work executed by the processor have been devoted to J_1 (last unit) and to J_2 (two units, hence $e_2 = 2$) because, after one unit, J_1 is finished (recall that $w_1 = 3$). Since J_1 is finished, it is not part anymore of the system state at time 3. Finally, the newly arrived job J_3 has an earlier relative deadline ($d_3 = 4$) than the new relative deadline of J_2 ($d_2 = 5$), hence it appears first in the sorted list \mathbf{x}_3 due to EDF. Fig. 5 shows the state evolution for this example, where J_1 is in red, J_2 in blue, and J_3 in green. Fig. 6 shows the corresponding schedule with the evolution of the remaining work.

Thanks to the construction introduced in Definition 3, the system state (\mathbf{x}, ℓ) is guaranteed to be Markovian, *i.e.*, the future state depends only on the current state and on the inputs received during the current instant.

A mathematical formalization of the state evolution is provided in Appendix A.1. It will be used to build the probability transition matrix in Appendix A.2. Beforehand, we analyze in details in Sec. 3 two state of the art policies from the literature, OA [2] and PACE [3].

3 Alternative Speed Policies

In this section, we focus on several solutions proposed in the literature, namely OA and PACE, defined in full details hereinafter. Up to our knowledge, these are the only ones that can deal with universal job characteristics, *i.e.*, with *random release times, deadlines, and sizes*. As explained in the introduction, there exist many other alternatives, but most of them are specific to particular (and more restrictive) real-time models. For both OA and PACE we first explain how the processor speed is computed. Then, thanks to well-chosen illustrative examples, we show that these two policies are sub-optimal and we explain what are their respective drawbacks.

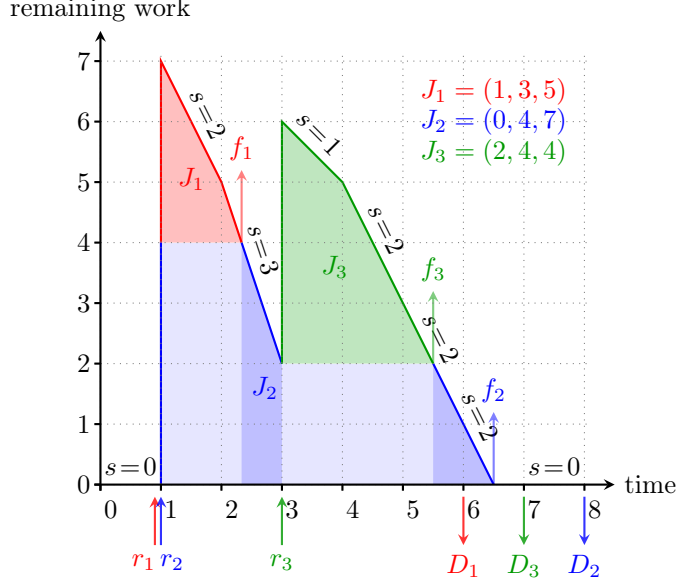


Fig. 6: Remaining work corresponding to the jobs of Fig. 5 under EDF and speeds $(0, 2, 3, 1, 2, 2, 2)$. For each job J_i , we have shown its release time r_i , its finish time f_i and its absolute deadline D_i (contrary to the system model where we use only *relative* dates to keep the state space finite; here we show their *absolute* counterpart so as to show how these dates occur on the absolute timeline). From $t = 1$ to $t = 7/3$, J_2 is inactive because J_1 has an earlier deadline. Then J_2 is active from $t = 7/3$ to $t = 3$, because at that time J_3 is released with an earlier deadline. Thus J_2 is preempted by J_3 and will only resume at $t = 11/2$ when J_3 completes.

3.1 The Optimal Available (OA) Speed Policy

Proposed by Yao et al. [2], Optimal Available (OA) is an online speed policy, which we denote σ^{OA} .

3.1.1 OA Speed Selection

Under the two assumptions that the size of each job is always equal to the maximal size W , and that no further job arrives in the system, $\sigma^{\text{OA}}(\mathbf{x}_t)$ is the optimal speed that can execute the current remaining work at time t , where (\mathbf{x}_t, ℓ) is the state at time t . Yao et al. show that, by considering that all jobs present at time t have a remaining worst case possible work equal to $W - e_i$ (with our notation $\mathbf{x}_t = ((e_i, d_i)_{i=1\dots n})$), then:

$$\sigma^{\text{OA}}(\mathbf{x}_t) = \left\lceil \max_{i=1\dots n} \frac{\sum_{j \text{ s.t. } d_j \leq d_i (W - e_j)} d_i}{d_i} \right\rceil. \quad (4)$$

where n is the total number of jobs present in the system at time t [2].

3.1.2 Sub-optimality of OA

OA has many good properties, notably it offers a constant competitive ratio with the optimal offline solution [3]. However, it is sub-optimal in terms of expected energy consumption because it is *too conservative* (it considers that all job sizes are equal to their worst case) and *myopic* (it does not anticipate future arrivals). As an example, consider an infinite sequence of jobs released periodically with period 4 (starting at $t = 1$), deadline 4, maximal size 100, and actual size equal to 10, 25, 50, 100 with respective probabilities $\frac{6}{8}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}$. At each instant, at most one such job is present in the system, because the period is equal to the deadline. Since OA only uses the maximal size to compute its speed (see Eq. (4)), at each instant t it selects exactly the same speed: $\forall t, \sigma^{\text{OA}}(\mathbf{x}_t) = \frac{100}{4} = 25$.

Assume in this section that the power function is $F(s) = s^2$. Any time interval $[4k + 1, 4k + 5]$, with $k \in \mathbb{N}$, contains a single job released at $t = 4k + 1$ with relative deadline 4, *i.e.*, with absolute deadline $4k + 5$. If the current job is of size 10 or 25 (this happens with probability $\frac{7}{8}$), then OA completes it in 1 time unit with the speeds $(25, 0, 0, 0)$, so its energy consumption is 25^2 . If the job is of size 50 (this happens with probability $\frac{1}{16}$), then OA completes it in 2 time units with the speeds $(25, 25, 0, 0)$, so its energy consumption is $2 \cdot 25^2$. Finally, if the job is of size 100 (this happens with probability $\frac{1}{16}$), then OA completes it in 4 time units with the speeds $(25, 25, 25, 25)$, so its energy consumption is $4 \cdot 25^2$. It follows that the expected average power spent to complete this sequence of jobs under OA is $\mathbb{E}(c^{\text{OA}})$:

$$\mathbb{E}(c^{\text{OA}}) = \lim_{K \rightarrow +\infty} \frac{1}{K} \sum_{k=1}^K \left(\frac{7}{8} + \frac{2}{16} + \frac{4}{16} \right) \cdot 25^2 = 781.25 J.$$

In this simple case, the optimal speed at time $4k + i$ ($i = 1, 2, 3, 4$) is 0 if the k^{th} job is finished before time $4k + i$, or else σ_{4k+i} , where $\sigma_{4k+1}, \sigma_{4k+2}, \sigma_{4k+3}$, and σ_{4k+4} are the solution of the following constrained optimization problem:

$$\begin{aligned} & \text{minimize } \sigma_{4k+1}^2 + \mathbb{P}(w > \sigma_{4k+1})\sigma_{4k+2}^2 + \mathbb{P}(w > \sigma_{4k+1} + \sigma_{4k+2})\sigma_{4k+3}^2 \\ & \quad + \mathbb{P}(w > \sigma_{4k+1} + \sigma_{4k+2} + \sigma_{4k+3})\sigma_{4k+4}^2 \\ & \text{under the two constraints } \forall 1 \leq i \leq 4, \sigma_{4k+i} \geq 0 \text{ and } \sum_{i=1}^4 \sigma_{4k+i} = 100. \end{aligned}$$

The optimal solution of this problem is, for any $k \in \mathbb{N}$, $\sigma_{4k+1}^* = 10$, $\sigma_{4k+2}^* = 15$, $\sigma_{4k+3}^* = 25$, and $\sigma_{4k+4}^* = 50$. Under this speed policy σ^* , a job of size 10, 25, 50, or 100 is terminated respectively in 1, 2, 3, or 4 time units, yielding an expected average energy equal to $\mathbb{E}(c^*) = 390.625 J$. The over-consumption of OA vs. this optimal policy is computed as $(\mathbb{E}(c^{\text{OA}}) - \mathbb{E}(c^*)) / \mathbb{E}(c^*)$. For our periodic sequence of jobs, it is exactly 100%.

An additional efficiency loss of OA is to be expected when several jobs are present simultaneously in the system with random arrival times and deadlines, or when the distribution of the size is even more biased towards 0. These intuitions will be confirmed by the numerical experiments presented in Sec. 7.

3.2 The Processor Acceleration to Conserve Energy (PACE) Speed Policy

Proposed by Lorch and Smith [17], the Processor Acceleration to Conserve Energy (PACE) algorithm computes a *closed form* of the optimal speed policy to execute a *single job* while minimizing the expected energy consumption when the power function is of the form $F(s) \propto s^3$.

3.2.1 PACE Speed Selection

We denote the PACE speed policy as $\sigma^{PACE,1}$ for one job, and its generalization to multiple jobs as σ^{PACE} .

Consider first a single job $J_1 = (w, d)$ such that, at time t , an amount e of work has already been executed. In that case, the closed formula for $\sigma^{PACE,1}$ is given in Eq. (5):

$$\sigma^{PACE,1}((e, d)) = \Omega(1 - G(e))^{-1/3}, \quad (5)$$

where G is the cumulative distribution function of the size w of job J_1 , given in Eq. (6):

$$G(e) = \mathbb{P}(w \leq e), \quad (6)$$

and the normalizing constant Ω is obtained by solving Eq. (7) that makes sure that the job J_1 with maximal size W is completed before its deadline d :

$$\int_0^W \frac{1}{\sigma^{PACE,1}((w, d))} dw = d. \quad (7)$$

PACE considers that the processor speed choices are continuous, but in practice only a finite number of speed values are available on actual processors. Moreover, decision times are also discrete. In the following, we will use a discrete version of PACE, which uses at each time instant the closest integer value to the speed computed with PACE.

Since the job size distribution is discrete in our system model, the cumulative distribution function G for the job size is approximated as a *piece-wise affine function* with integer changing points as follows with $U(i) = \sum_{j=0}^i \mathbb{P}(w=j)$:

$$\forall i \leq w \leq i+1, G(w) = (U(i+1) - U(i)) U(i) (w - i) + U(i), \quad (8)$$

Eq. (8) can be written under the affine form $G(w) = a_i w + b_i$ with $a_i = (U(i+1) - U(i)) U(i)$ on the one hand and $b_i = U(i) - (U(i+1) - U(i)) U(i) i$ on the other hand.

Recall that the normalizing constant Ω is obtained by solving Eq. (7) for Ω . Replacing in Eq. (7) G by its piece-wise affine approximation from Eq. (8) and solving the resulting equation for Ω yields:

$$\Omega = \frac{3}{4d} \sum_{i=0}^{W-1} \frac{1}{a_i} \left[(1 - ia_i - b_i)^{4/3} - (1 - (i+1)a_{i+1} - b_{i+1})^{4/3} \right]. \quad (9)$$

Hence, the PACE speed for a single job J_1 with a deadline d and an amount of already executed work $e \in [i, i+1)$ is:

$$\sigma^{PACE,1}((e, d)) = \Omega [1 - (a_i e + b_i)]^{-1/3}. \quad (10)$$

In the case where several jobs J_i are present at a given time t , PACE executes each individual job J_i at the speed computed for each job in isolation, as presented above. Then, the speed of the processor is the sum of the speeds allocated to each currently present job, these jobs being executed using processor sharing:

$$\sigma^{PACE,n}(\mathbf{x}_t) = \sum_{J_i \in \mathbf{x}_t} \sigma^{PACE,1}(J_i). \quad (11)$$

In general, the speed computed by Eq. (11) does not belong to the set of available speeds \mathcal{S} . In the final equation, we compute σ^{PACE} by selecting the smallest available speed from \mathcal{S} larger than the result of Eq. (11):

$$\sigma^{PACE}(\mathbf{x}_t) = \min \left\{ s \in \mathcal{S} \mid s \geq \sum_{J_i \in \mathbf{x}_t} \sigma^{PACE,1}(J_i) \right\}. \quad (12)$$

A final remark is that the speed aggregation computed in Eq. (11) and the discretization computed in Eq.(12) (discretization due to \mathcal{S}) contribute to the sub-optimality of PACE, as will be shown in the next section.

3.2.2 Sub-optimality of PACE

PACE has many good properties, notably it is optimal in terms of energy consumption for *single jobs*, hence also for jobs released by periodic tasks with a deadline smaller than their period. However, it is sub-optimal when several jobs are released simultaneously.

Consider an infinite sequence of jobs released by n periodic tasks $\{T_i\}_{1 \leq i \leq n}$ with the characteristics $T_i = (P_i = n, w_i = W = 1, \Delta_i = i)$. Because all the tasks are periodic with the same period n , at any time $t = kn$ (with $k \in \mathbb{N}$) we therefore have n jobs J_i released simultaneously, all with their size equal to 1, and with respective relative deadlines equal to $1, 2, \dots, n$. Since the distribution of the sizes is deterministic (equal to 1), the speed selected by PACE to execute the job J_i with relative deadline i is constant over the time interval from its release time to its deadline (*i.e.*, $[kn, kn+i]$), and equal to $\sigma^{PACE,1}(J_i) = 1/i$.

It follows that the cumulative speed selected at time $t = kn$ by PACE, when the state \mathbf{x}_{kn} consists of the n jobs J_i , is the n^{th} harmonic number H_n :

$$\sigma^{PACE,n}(\mathbf{x}_{kn}) = \sum_{i=1}^n \sigma^{PACE,1}(J_i) = \sum_{i=1}^n \frac{1}{i} = H_n, \quad (13)$$

which is upper and lower-bounded as $\log(n) < H_n \leq \log(n) + 1$.

At time $kn + 1$, the job J_1 is finished, so the cumulative speed is now:

$$\sigma^{PACE,n}(\mathbf{x}_{kn+1}) = \sum_{i=2}^n \sigma^{PACE,1}(J_i) = \sum_{i=2}^n \frac{1}{i} = H_n - H_1, \quad (14)$$

and so on and so forth up to time $kn + n - 1$ where the cumulative speed is:

$$\sigma^{PACE,n}(\mathbf{x}_{kn+n-1}) = H_n - H_{n-1}. \quad (15)$$

Since the jobs are deterministic, the expected average power spent to complete this sequence of jobs under PACE is simply the average energy per time unit, computed again with Eq. (3). If $F(s) = s^2$, then it is equal to:

$$\begin{aligned} \mathbb{E}(c^{\text{PACE}}) &= c^{\text{PACE}} = \lim_{T \rightarrow +\infty} \frac{1}{T} \sum_{t=0}^T \sigma^{PACE,n}(\mathbf{x}_t)^2 \\ &= \lim_{k \rightarrow +\infty} \frac{k(H_n^2 + (H_n - H_1)^2 + \dots + (H_n - H_{n-1})^2)}{kn} \\ &= \frac{H_n^2 + (H_n - H_1)^2 + \dots + (H_n - H_{n-1})^2}{n} \end{aligned} \quad (16)$$

Developing the numerator of Eq. (16) and regrouping squared terms first and then the cross products involving $1/j$ as the smallest factor yields:

$$\begin{aligned} \mathbb{E}(c^{\text{PACE}}) &= \frac{1}{n} \left(\sum_{i=1}^n \frac{i}{i^2} + 2 \sum_{j=2}^n \frac{1}{j} \sum_{i=1}^{j-1} \frac{i}{i} \right) = \frac{1}{n} \left(H_n + 2 \sum_{j=2}^n \frac{j-1}{j} \right) \\ &= \frac{1}{n} \left(H_n + 2 \sum_{j=2}^n \left(1 - \frac{1}{j} \right) \right) = \frac{1}{n} \left(H_n + 2(n-1 - H_n + 1) \right). \end{aligned}$$

As a conclusion, the expected average power spent under PACE is:

$$\mathbb{E}(c^{\text{PACE}}) = \frac{2n - H_n}{n} = 2 - \frac{H_n}{n}. \quad (17)$$

Meanwhile, under the same set of jobs, the optimal speed policy σ^* (as well as OA for that matter) will use speed 1 at each time instant: $\forall t, \sigma^*(\mathbf{x}_t) = 1$. Therefore, the

average power used by the optimal speed policy to execute the infinite sequence of jobs (J_i) is simply $\mathbb{E}(c^*) = c^* = 1$.

The over-consumption of PACE versus the optimal policy is computed as $(\mathbb{E}(c^{\text{PACE}}) - \mathbb{E}(c^*)) / \mathbb{E}(c^*)$. For our infinite sequence of jobs, it is lower-bounded and upper-bounded by:

$$2 - \frac{\log(n) + 1}{n} < \frac{\mathbb{E}(c^{\text{PACE}}) - \mathbb{E}(c^*)}{\mathbb{E}(c^*)} < 2 - \frac{\log(n)}{n}. \quad (18)$$

As a consequence, when n grows to $+\infty$, the over-consumption of PACE converges to 100%.

4 Markov Decision Process Solution

We present in this section our algorithm that computes the optimal speed policy, as a Solution of a Markov Decision Process (called MOSP for ‘‘MDP Optimal Speed Policy’’). The numerical experiments reported in Sec. 7 confirm that optimal speeds computed by our solution provide an energetic gain in all cases over the two speed policies presented in Sec. 3 (OA and PACE). For certain sets of jobs, OA (resp. PACE) can be very close to our optimal solution, while in other cases they can be very far.

4.1 Expected Energy Consumption

We first generalize Eq. (3) in order to compute the *expected* average power consumption per time unit. Recall that the power consumption of the processor when working at speed $\sigma(\mathbf{x}_t)$ is $F(\sigma(\mathbf{x}_t))$. The ergodic theorem can be used here because the empty state ($\mathbf{x} = \emptyset, \ell = L - 1$) can be reached from any other state. This implies that the optimal average power consumption c^* can be defined as

$$c^* = \min_{\sigma(\cdot)} \left(\lim_{T \rightarrow +\infty} \frac{1}{T} \int_{t=0}^T F(\sigma(\mathbf{x}_t)) \right). \quad (19)$$

The minimum is taken over all speed policies $\sigma(\cdot)$, which select one speed for each state and each time instant. Since c^* is a long-run performance criterion, by ergodicity it does not depend on the initial state. Our goal is to compute the optimal speed policy $\sigma^*(\cdot)$ that minimizes Eq. (19).

4.2 Bellman Equations

Since speed decisions are only taken at integer times, the integral in (19) can be replaced by a sum over a discrete version of the power function (see the definition of \tilde{F}). In this discrete context, the optimal average power consumption c^* can be computed by solving the Bellman Equation (see Eq. (20) below). The solutions of this equation can be computed explicitly because the state space \mathcal{X} is finite (see Sec. 2.2.3), and the transition probability matrix \mathbf{P} is known (see Appendix A.2 for its construction).

According to Eq. (8.4.2) in [33], the optimal average power consumption c^* is the solution of the Bellman equation (20), together with h^* , a bias function of an optimal policy:

$$c^* + h^*(\mathbf{x}, \ell) = \min_{s \in \mathcal{A}(\mathbf{x})} \left(\tilde{F}(s, \mathbf{x}) + \sum_{(\mathbf{x}', \ell') \in \mathcal{X} \times L} \mathbf{P}((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) h^*(\mathbf{x}', \ell') \right). \quad (20)$$

Here, $\tilde{F}(s, \mathbf{x})$ is the expected energy consumption during the current interval. If the server does not become idle in the interval, then its energy consumption is just the power $F(s)$ times the interval duration (equal to 1). If the server becomes idle in the interval, then its energy consumption is $\beta_s F(s) + (1 - \beta_s) P_{idle}$, where β_s is the expected busy time (*i.e.*, the duration when the processor is not idle) and P_{idle} is the power dissipation when the processor is idle (could be non zero for some devices).

In expectation, we get

$$\tilde{F}(s, \mathbf{x}) = \beta_s(\mathbf{x}) F(s) + (1 - \beta_s(\mathbf{x})) P_{idle}, \quad (21)$$

where

$$\beta_s(\mathbf{x}) = \mathbb{E} \left(\min \left(1, \frac{\sum_i w_i - e_i}{s} \right) \middle| \mathbf{x} \right) \quad (22)$$

which can be computed since the distribution of all w_i is known. As for $\mathcal{A}(\mathbf{x})$, this is the set of admissible speeds in state \mathbf{x} , built to ensure that no job will miss a deadline. It is obtained by summing the worst-case remaining amount of work $W - e_i$ one must spend on job (e_i, d_i) , over all jobs in \mathbf{x} having a relative deadline d_i equal to 1:

$$\mathcal{A}(\mathbf{x}) = \left\{ s \in \mathcal{S}, s \geq \sum_{(e_i, d_i) \in \mathbf{x} \text{ s.t. } d_i=1} (W - e_i) \right\}. \quad (23)$$

Let us explain in detail the construction of $\mathcal{A}(\mathbf{x})$, the set of speeds that can be selected by the processor if the current state at time t is (\mathbf{x}, ℓ) . At time t , the pending jobs whose absolute deadline is equal to $t + 1$ are called *critical jobs*, because they must all be completed in the interval $[t, t + 1]$: they are all the jobs J_i whose current relative deadline is $d_i = 1$. In the worst case, the remaining size of these jobs is $W - e_i$, *i.e.*, the greatest possible size minus the work already executed on the job J_i . Therefore, if on the one hand the speed selected by the processor is greater than $\sum_{(e_i, d_i) \in \mathbf{x} \text{ s.t. } d_i=1} (W - e_i)$, then all the critical jobs will be completed before time $t + 1$. This is precisely what Eq. (23) states. By repeating this reasoning at times $t + 1$, $t + 2$, and so on, one can make sure that no job will ever miss its deadline. On the other hand, if a speed smaller than $\sum_{(e_i, d_i) \in \mathbf{x} \text{ s.t. } d_i=1} (W - e_i)$ is chosen, there is a chance that all critical jobs are of size $W - e_i$ so that at least one of them will miss its deadline, which we do not want.

The optimal speed policy σ^* is the speed policy that achieves, for each state (\mathbf{x}, ℓ) , the minimum over all admissible speeds in Eq. (20):

$$\sigma^*((\mathbf{x}, \ell)) = \arg \min_{s \in \mathcal{A}(\mathbf{x})} \left(\tilde{F}(s, \mathbf{x}) + \sum_{(\mathbf{x}', \ell') \in \mathcal{X} \times L} \mathbf{P}((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) h^*((\mathbf{x}', \ell')) \right). \quad (24)$$

The proof that the optimal average power consumption c^* defined in Eq. (19) satisfies Eq. (20) is classical provided that the MDP is ergodic (see Th. (8.4.5) in [33]). In our case, for any choice of speeds and any initial state, the empty state $(\mathbf{x} = \emptyset, \ell = L - 1)$ can be reached with positive probability. This occurs when there are null job arrivals for Δ time steps (*i.e.*, jobs J_i such that $w_i = 0$), and the current inter-arrival time is $L - 1$. This sequence of events has a positive probability when $\mathbb{P}(w_i = 0) > 0$. In such a case, the empty state $(\mathbf{x} = \emptyset, \ell = L - 1)$ is a positive recurrent and aperiodic state. This fact ensures that the MDP is ergodic and concludes the proof.

The optimal speed policy can be computed using the classical algorithms Value Iteration (VI) or Policy Iteration (PI). The convergence of VI and PI is guaranteed because the MDP is recurrent (see Sec. 8.3 in [33]). Our numerical experiments in Sec. 7 rely on VI, which is presented in details in Sec. 6.

Finally, starting from the joint probability distribution of the inter-arrival times, sizes, and deadlines, it is possible to construct the transition matrix $\mathbf{P}((\mathbf{x}, \ell_1), s, (\mathbf{y}, \ell_2))$ that gives the probability to go from state (\mathbf{x}, ℓ_1) to state (\mathbf{y}, ℓ_2) over time step $(t, t + 1]$, when the processor uses speed s . This construction is not straightforward and is detailed in Appendix A. Of course, the global formula for \mathbf{P} is quite complex, but one should keep in mind that those computations are done offline and only once. At run time, when the processor is actually executing jobs, it uses the precomputed optimal policy σ^* .

4.3 Deadline Constraints

Eq. (20) shows how to compute the speed that minimizes the energy consumption but does not explicitly use the deadline constraints. Actually, the constraint that all jobs must be completed before their deadline, while their size and arrival times are unknown, is contained in the definition of the set $\mathcal{A}(\mathbf{x})$ in Eq. (23).

From Sec. 4.2, we have seen that if the state at time t is (\mathbf{x}, ℓ) , then if the speed selected by the processor is greater than $\sum_{(e_i, d_i) \in \mathbf{x} \text{ s.t. } d_i = 1} (W - e_i)$, then all the critical jobs will be completed before time $t + 1$. By doing that at each time step, one can make sure that no job will ever miss its deadline.

The final issue is to check that, for all possible states (\mathbf{x}, ℓ) reached at time t under our choice of speeds from time 0 to time t , the set $\mathcal{A}(\mathbf{x})$ is never empty. The following proposition gives sufficient (and necessary in a particular case) conditions on s_{\max} , the maximal speed available to the processor, to guarantee that $\mathcal{A}(\mathbf{x})$ is never empty.

Let us recall that Υ is the maximal number of jobs that can arrive simultaneously and that T_{\min} is the minimal inter-arrival time (both introduced in Sec 2.1.8). Let us denote by Δ_{\min} the minimal deadline of any job: $\Delta_{\min} = \min_i d_i$. Since for all i ,

$d_i \in \mathbb{N}^*$, we thus have $\Delta_{\min} \geq 1$. Finally, let us denote by L_{\min} the minimal inter-arrival time between two *non simultaneous* jobs. When $T_{\min} > 0$, we have $T_{\min} = L_{\min}$ and $\Upsilon = 1$.

As said in Sec. 2.2, a sequence of jobs is *feasible* by the processor if there exists a way to choose speeds such that no job misses its deadline. This is regardless of energy considerations.

Proposition 1. *The two following propositions hold:*

(P1) *The system is feasible if and only if $s_{\max} \geq \max(\Upsilon W/\Delta_{\min}, \Upsilon W/L_{\min})$.*

(P2) *The optimal energy policy σ^* completes all jobs before their deadlines if and only if the job sequence is feasible.*

Proof. The proof is by contradiction. We first prove the necessary part of (P1):

- Assume that $s_{\max} < \Upsilon W/\Delta_{\min}$ and consider a set of Υ jobs, each one of size W and deadline Δ_{\min} , which all arrive simultaneously at time 1. Then one of these jobs will miss its deadline under all speed policies. When $T_{\min} > 0$, we have $\Upsilon = 1$ so the set contains a single job.
- Assume now that $s_{\max} < \Upsilon W/L_{\min}$ and consider that every L_{\min} instant, a set of Υ jobs arrive, each one of size W . Under any speed policy, at most $s_{\max} L_{\min}$ work can be executed before a new job arrives, so the uncompleted work will grow to infinity (since $s_{\max} L_{\min} < \Upsilon W$ by assumption), implying that some job will eventually miss its deadline. This is independent of the job deadlines because the uncompleted work grows to infinity while the deadlines are finite.

We then prove the sufficient part of (P1). Let $s_{\max} \geq \max(\Upsilon W/\Delta_{\min}, \Upsilon W/L_{\min})$. Then there exists one policy that does not miss any deadline, namely the policy σ^{\max} that uses the speed s_{\max} all the time: $\forall t, \sigma^{\max}(t) = s_{\max}$.

There remains to prove (P2): we need to show in both cases that the optimal speed policy σ^* will also execute all jobs before their deadlines. For this, we define an alternative framework with a new set of available processor speeds \mathcal{S}' and a new power consumption function F' . The processor can now use unbounded speeds: $\mathcal{S}' = \mathcal{S} \cup \{s_{\max} + 1, s_{\max} + 2, \dots\}$, but when it uses speeds larger than s_{\max} , its energy consumption F' becomes infinite:

$$\begin{cases} \text{if } s > s_{\max} \text{ then } & F'(s) = +\infty \\ \text{otherwise} & F'(s) = F(s) \end{cases}$$

Under σ^{\max} , the policy that always uses s_{\max} , the long-run average power consumption is $F'(s_{\max}) = F(s_{\max})$ and no job misses its deadline, by (P1).

By definition of optimality, the optimal speed policy under the new power function F' has the smallest energy consumption. Thus the policy σ^{\max} , which uses speed s_{\max} at every instant, has a larger energy consumption than the optimal one. Since σ^{\max} leads to a finite energy consumption, the optimal speed policy does too. This implies that speeds larger than s_{\max} , which would have given an infinite energy consumption, were never used by the optimal speed policy σ^* . This implies that the optimal policies under both power functions coincide. This means that the set $\mathcal{A}(\mathbf{x})$ is never empty for the optimal policy with the original power function, hence no job misses its deadline. \square

5 A new heuristic speed policy: Expected Load (EL)

The main drawback of MOSP is its offline computational complexity due to its huge state space. On the other hand, the main drawback of OA is its poor performance because it is too conservative: the size of each job is assumed to be equal to W . To alleviate this, we propose a new speed policy, called Expected Load (EL), which takes into account the statistical information on the job arrivals, as in PACE, but does not require any costly offline computations, as required by Value Iteration (Algo. 1) for our MDP-based optimal policy presented in Sec. 4.

Let (\mathbf{x}, ℓ) be the current state with n jobs present in the system, such that $\mathbf{x} = ((e_1, d_1), \dots, (e_n, d_n))$. First, we compute a statistical bound on the *expected remaining size* \hat{w}_i of each job $J_i = (e_i, d_i) \in \mathbf{x}$ as using its expected size and standard deviation. Its current expected size is $\hat{w}_i = \mathbb{E}(w_i - e_i \mid \mathbf{x})$. Its current variance is $\hat{v}_i = \mathbb{E}((w_i - e_i)^2 \mid \mathbf{x}) - \hat{w}_i^2$. When w , τ , and d are i.i.d., the computations simplify to $\hat{w}_i = \mathbb{E}(w \mid w > e_i) - e_i$ and $\hat{v}_i = \mathbb{E}((w - e_i)^2 \mid w > e_i) - \hat{w}_i^2$.

Using a parameter $K \geq 0$, we construct a remaining size bound as

$$b_i = \hat{w}_i + K\sqrt{\hat{v}_i} \quad \text{if } d_i > 1. \quad (25)$$

However, if $d_i = 1$, then to guarantee feasibility we have to consider the worst case scenario, *i.e.* the case where the size of the job is equal to W :

$$b_i = W - e_i \quad \text{if } d_i = 1. \quad (26)$$

Another ingredient used to construct the new heuristic is to anticipate future arrivals. A *virtual* job is added: $J_0 = (\hat{\tau}_0, \hat{w}_0, \hat{d}_0)$, such that:

- its inter-arrival time is $\hat{\tau}_0 = \mathbb{E}(\tau \mid (\mathbf{x}, \ell))$, equal to $\mathbb{E}(\tau \mid \tau > \ell)$ in the i.i.d. case,
- its size is $\hat{w}_0 = \mathbb{E}\left(\sum_{i=1}^N w_i \mid (\mathbf{x}, \ell)\right)$, equal to $\mathbb{E}\left(\sum_{i=1}^N w_i\right)$ in the i.i.d. case, where N is the number of simultaneous arrivals. From Wald's formula, we get $\hat{w}_0 = \mathbb{E}(\sum_{i=1}^N w_i) = \mathbb{E}(N)\mathbb{E}(w) = \frac{\mathbb{E}(w)}{1-\theta(0)}$, where $\theta(0)$ is the probability that the inter-arrival time is zero. As for its variance, we get $\hat{v}_0 = \frac{\text{var}(w)}{1-\theta(0)}$.
- Its relative deadline is $\hat{d}_0 = \mathbb{E}(d) + \hat{\tau}_0$.

If the virtual job does not interfere with the current jobs (that is, if $\hat{\tau}_0 \geq d_n$), then we do not add the virtual job J_0 or, equivalently, we fix its estimated size to $b_0 = 0$.

Finally, here is the EL policy: compute the speed σ^{EL} similarly to the OA speed policy, but based on the virtual job J_0 (if added) and on the upper bounds of the expected sizes for all the current jobs J_i in \mathbf{x} (to be compared with Eq. (4)):

$$\sigma^{\text{EL}}(\mathbf{x}, \ell) = \left[\max_{i=0, \dots, n} \frac{\sum_{j \text{ s.t. } d_j \leq d_i} b_j}{d_i} \right]. \quad (27)$$

Two remarks are in order:

- According to Eq. (27), the computational complexity of EL is $\mathcal{O}(n)$ where n is the number of pending jobs, which is upper-bounded by $\min\{B, \Upsilon \Delta\}$, as explained in Sec 2.1.8.

- As long as the maximum processor speed is high enough, the Eq. (27) guarantees the feasibility of EL. Indeed, the processor speed is such that all jobs with a relative deadline of 1 (that is, all jobs that must complete within the current time interval) are executed by using the worse possible size for those jobs instead of their average size. This implies that all the jobs complete before their deadline. We should note that this condition on the maximal processor speed for feasibility is similar to the feasibility condition computed for OA (see Theo. 1 in [4]). Like OA, the processor speed should be high enough to ensure that EL always executes all the jobs before their deadline.

The experiments reported in Sec. 7 will show that, in many cases, EL outperforms both OA and PACE. In the example showing the sub-optimality of OA in Sec. 3.1.2, EL chooses the speeds 5, 5, 20, and 70, yielding an expected consumption $\mathbb{E}(c^{\text{EL}}) = 543.75J$. This is more than 30% better than OA (543.75J versus 781.25J). In the example showing the sub-optimality of PACE (Sec. 3.2.2), EL coincides with OA and hence is optimal. Both policies coincide because the sizes of all jobs are deterministic, and the virtual job J_0 does not interfere with the current jobs.

6 Implementation of MOSP

The implementation of our optimal solution MOSP requires two phases, unlike sub-optimal solutions OA, PACE, or EL whose offline phases are direct. (1) The *offline* phase computes the optimal speed policy for each possible encountered state; we denote it σ^{MDP} as it may differ from the optimal policy σ^* , because it is only ε -optimal. We use a Value Iteration algorithm (VI) (Sec. 6.1). (2) And the *online* phase that applies at runtime the optimal speed policy σ^{MDP} jointly with the EDF scheduling policy (Sec. 6.2).

6.1 Value Iteration Algorithm (VI)

The VI algorithm is implemented in the `ValueIteration()` function, presented in Algo. 1. The second **foreach** loop (lines 16 to 18) computes the ε -optimal speed $\sigma^{\text{MDP}}((\mathbf{x}, \ell))$ to be selected in each state (\mathbf{x}, ℓ) . This optimal speed policy σ^{MDP} is *stationary* and does not depend on t . The state (\mathbf{x}, ℓ) of the system belongs to the state space $\mathcal{X} \times \{0, \dots, L\}$. The vector u^{n+1} computed in the first **foreach** loop (lines 6 to 8) is used to check the convergence. Until the algorithm has converged, u^{n+1} has no meaning with regard to the physics of the system. It is an intermediate value for solving the problem. Only the value of u at convergence has any physical significance: let's call n^* the number of iterations from which VI has converged, then $u^{n^*}(i)$ corresponds to the energy consumed when starting from state i . The *span* of a vector (used at line 11) is the difference between its maximal value and its minimal value: $\text{Span}(x) = \max_i(x_i) - \min_i(x_i)$. A vector with a span equal to 0 has all its coordinates equal.

The time complexity of `ValueIteration(ε)` grows logarithmically with the stopping criterion ε . More precisely, it is $\mathcal{O}\left(S^2 \log\left(\frac{\text{span}(h^*)}{\varepsilon}\right)\right)$ (see [33], Theorem 8.5.2, p. 368), where S is the size of the state space. The numerical experiments in Sec. 7 show that convergence occurs reasonably fast in practice.

Algorithm 1: Function `ValueIteration` to compute the optimal speeds in each state.

Input: A stopping criterion $\varepsilon > 0$.

Output: An ε -optimal speed policy σ^{MDP} .

```

1 Function ValueIteration( $\varepsilon$ ) is
2    $u^0 \leftarrow (0, 0, \dots, 0)$ ;
3    $n \leftarrow 0$ ;
4    $cont \leftarrow True$ ;
5   while  $cont$  do
6     foreach  $(\mathbf{x}, \ell) \in \mathcal{X} \times \{0, \dots, L\}$  do
7        $u^{n+1}[(\mathbf{x}, \ell)] \leftarrow \min_{s \in \mathcal{A}((\mathbf{x}, \ell))} \left\{ \tilde{F}(s, \mathbf{x}) + \sum_{\substack{(\mathbf{x}', \ell') \in \\ \mathcal{X} \times \{0, \dots, L\}}} P((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) u^n[(\mathbf{x}', \ell')] \right\}$ ;
8     end
9      $n \leftarrow n + 1$ ;
10    if  $n > L$  then
11      if  $\text{Span}(u^{n+L-1} - u^{n-1}) < \varepsilon$  then
12         $cont \leftarrow False$ ;
13      end
14    end
15  end
16  foreach  $(\mathbf{x}, \ell) \in \mathcal{X} \times \{0, \dots, L\}$  do
17     $\sigma^{\text{MDP}}((\mathbf{x}, \ell)) \in \arg \min_{s \in \mathcal{A}((\mathbf{x}, \ell))} \left\{ \tilde{F}(s, \mathbf{x}) + \sum_{\substack{(\mathbf{x}', \ell') \in \\ \mathcal{X} \times \{0, \dots, L\}}} P((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) u^n[(\mathbf{x}', \ell')] \right\}$ ;
18  end
19  return  $\sigma^{\text{MDP}}$ ;
20 end

```

6.2 Runtime Algorithm

The `RunTime()` procedure, presented in Algo. 2, consists of a **foreach** loop that runs forever, where *tick* represents the infinite sequence of time instants coming from the environment.

The body of this loop involves three main steps: the processor must (i) compute, at each time instant, the new state based on the newly arrived jobs; (ii) retrieve the optimal speed; and (iii) spend the corresponding amount of work on the list of current jobs:

1. The function `NewJobs()` returns the list of the new jobs arrived during the current instant (line 4). This list corresponds to the term $a(t+1)$ in Eq. (37). The new state $(\mathbf{x}_{t+1}, \ell_{t+1})$ is computed from the previous state (\mathbf{x}_t, ℓ_t) and the list returned by `NewJobs`. The \oplus operator is defined in Sec. A.1.

2. The optimal speed s_{t+1} is obtained by a table look-up operation on the speed table σ_tab passed in argument from the current state $(\mathbf{x}_{t+1}, \ell_{t+1})$ (line 5). The speed table can be either σ^{MDP} computed offline by VI, or a table computed by any other speed policy, *e.g.*, σ^{OA} for OA, σ^{PACE} for PACE, or σ^{EL} for EL.
3. The optimal speed s_{t+1} is then spent on the current list of jobs \mathbf{x}_{t+1} by the **for** loop of line 7. Recall that \mathbf{x}_{t+1} is sorted by increasing order of the job's relative deadline. Therefore, starting with the first job in the list is consistent with the EDF scheduling policy (line 8). The `Finished()` function materializes the fact that the `RunTime()` procedure does not know the exact execution time of the job being executed (non-clairvoyant case) but "discovers" it when the right amount of work has been spent on this job. If the job is indeed finished, then it is removed from \mathbf{x}_{t+1} (line 11), and we continue until all the amount of work has been spent. The list structure used here is assumed to provide the `Head()` and `Remove()` methods, while the job structure is assumed to consist of two fields, `Work` and `Deadline` (in accordance with Def. 3).

Algorithm 2: Procedure `RunTime` used by the processor to apply the optimal speed online.

Input: A speed policy σ_tab .

```

1 Procedure RunTime( $\sigma\_tab$ ) is
2    $t \leftarrow 0$  ;
3   foreach  $t$  do
4     /* Insert the new jobs in the state */
4      $(\mathbf{x}_{t+1}, \ell_{t+1}) \leftarrow (\mathbf{x}_t, \ell_t) \oplus \text{NewJobs}()$  ;
5     /* Retrieve the optimal processor speed */
5      $s_{t+1} \leftarrow \sigma\_tab[(\mathbf{x}_{t+1}, \ell_{t+1})]$  ;
6     SetProcessorSpeed ( $s_{t+1}$ ) ;
7     for  $i = 1$  to  $s_{t+1}$  do
8       /* Retrieve the first job  $J_i$  in the list */
8        $J_i \leftarrow \mathbf{x}_{t+1}.\text{Head}()$  ;
9       /* Execute one unit of work on  $J_i$  */
9        $J_i.\text{Work} \leftarrow J_i.\text{Work} + 1$  ;
10      /* Non-clairvoyant case */
10      if Finished( $J_i$ ) then
11        /* Remove  $J_i$  from the list of active jobs */
11         $\mathbf{x}_{t+1}.\text{Remove}(J_i)$  ;
12      end
13    end
14     $t \leftarrow t + 1$  ;
15  end
16 end

```

The computational cost of the body of the **foreach** loop of Algo. 2 is $\mathcal{O}(1)$ (lines 3 to 15). Let us investigate Algo. 2 line by line. The number of new jobs is bounded by the size B of the buffer, and similarly for the size of the lists of jobs \mathbf{x}_t and \mathbf{x}_{t+1} , so the cost of line 4 is $\mathcal{O}(1)$. The cost of the table look-up operation at line 5 is also $\mathcal{O}(1)$. Finally, the optimal speed s_{t+1} is bounded by the processor maximal speed s_{\max} , so the entire cost of the **for** loop (lines 7 to 13) is also $\mathcal{O}(1)$.

7 Numerical Experiments

To illustrate the key properties of the optimal policy, the job characteristics (*i.e.*, inter-arrival times, sizes, and deadlines) follow well-chosen distributions for each experiment. As stated in Sec. 2.1.8, we denote by B the maximal number of jobs in the buffer of the processor at any time. In all the simulations the set of available speeds \mathcal{S} is assumed to be $\{0, 1, \dots, s_{\max}\}$, with $s_{\max} = BW$ to comply with Prop. 1 (here, $\Delta_{\min} = 1$ and $B > \Upsilon$ by definition).

$N = 1,000$ independent simulations are run for each experiment. For each experiment, we randomly generate a job sequence over a finite time horizon of $T = 1,000$ time steps, on which we apply the MOSP speed policy. As explained in Sec. 4.2, the algorithm used to solve the MDP is VI, with stopping criterion $\varepsilon = 0.01$.

We store the sequence of generated jobs, and we compute the total amount of work executed by MOSP from $t = 0$ to T : $R_{tot} = \int_0^T \sigma^{\text{MDP}}(t) dt$. Then, for each policy $\text{POL} \in \{\text{OA}, \text{PACE}, \text{EL}\}$, we simulate POL over the *same* sequence of jobs *and* until the amount of work executed by POL is also equal to R_{tot} .

At the end of each experiment, we compute the energy consumption of each speed policy on the job sequence. The power consumption function used in all the simulations is $F(s) = s^3$. Since each simulation is performed over a *finite* horizon (1,000 time steps), we compute the *total* energy C instead of the *expected* energy average c :

$$C^{\text{POL}} = \sum_{t=0}^T F(s(t)) \quad \text{where POL} \in \{\text{OA}, \text{PACE}, \text{EL}, \text{MOSP}\}. \quad (28)$$

MOSP being the optimal speed policy, we chose to compute the over-consumption of the three other speed policies w.r.t. MOSP:

$$\text{over-consumption POLvsMDP} = \frac{\mathbb{E}(C^{\text{POL}}) - \mathbb{E}(C^{\text{MOSP}})}{\mathbb{E}(C^{\text{MOSP}})} \quad \text{where POL} \in \{\text{OA}, \text{PACE}, \text{EL}\} \quad (29)$$

and C^{POL} is the total energy computed with Eq. (28). The comparison is fair since all policies execute the same total amount of work R_{tot} .

The speed policy σ^{MDP} computed with Algo. 1 is optimal only in the infinite horizon case. But, of course, the simulations can only be performed over a finite horizon. However, the finite horizon cost converges at a rate equal to $1/T$ towards the optimal solution when the number of time steps grows.

For reproducibility purposes, all data and programs used in the experimental section are available at <https://github.com/ExploitingJobVariability/Comparison4ProcessorSpeedPolicies>, and the edge detection program of Section 7.4 is available at <https://opensourcelibs.com/lib/tmf>.

7.1 Impact of Inter-arrival times

We first study the impact of the inter-arrival time distribution on the over-consumption of OA, PACE, and EL compared to MOSP. The experiments are done on a system with the following characteristics: job deadlines are uniformly distributed from 1 to $\Delta = 3$, job sizes are uniformly distributed from 1 to $W = 4$, and the buffer size is set to $B = 4$. Finally, for the EL policy, we use $K = 1$ as the default value. This value is kept for EL in Secs. 7.1, 7.2 and 7.3.

We report the over-consumption of OA, PACE, and EL w.r.t. MOSP in Tab. 1, for different inter-arrival time distributions: first when the inter-arrival time τ belongs to $\{0, 1\}$ (Tab. 1a), and then to $\{1, 2, 3, 4\}$, meaning without multiple job arrivals (Tab. 1b). For each simulation, we also compute the 95% confidence intervals of the over-consumption values. For example, confidence intervals for the third line of Tab. 1a are: PACE vs MOSP $\rightarrow [44.2, 44.6]$, OA vs MOSP $\rightarrow [10.9, 11.2]$, and EL vs MOSP $\rightarrow [10.6, 10.9]$. For readability reasons, the other confidence intervals are not shown here but are available via the GitHub repository. All of them are small enough to be negligible w.r.t. the empirical mean.

IATD			over-consumption			
$\tau=0$	$\tau=1$	#jobs	P v. M	O v. M	E v. M	Algo. 1 execution time
0	1	1.3	44.4%	11.0%	10.7%	2.14 min
1/4	3/4	1.7	66.6%	8.5%	13.8%	1.77 min
1/2	1/2	2.4	75.9%	5.6%	4.0%	1.59 min
3/4	1/4	3.0	76.0%	0.6%	7.5%	2.11 min

(a) Parameters: $\Delta = 3$, $W = 4$, $\tau \in \{0, 1\}$, $B = 4$.

IATD					over-consumption			
$\tau=1$	$\tau=2$	$\tau=3$	$\tau=4$	#jobs	P v. M	O v. M	E v. M	Algo. 1 execution time
3/4	1/4	0	0	1.2	33.4%	4.8%	9.6%	8.70 min
1/4	1/2	1/4	0	0.75	13.1%	1.1%	3.2%	42.03 min
0	1/4	1/2	1/4	0.5	0.1%	1.8%	4.4%	65.50 min
0	0	1/4	3/4	0.4	1.9%	3.7%	0%	265.10 min

(b) Parameters: $\Delta = 3$, $W = 4$, $\tau \in \{1, 2, 3, 4\}$, $B = 4$.

Table 1: Influence of the inter-arrival time distribution (IATD) on the over-consumption of PACE versus MOSP (P v. M), OA versus MOSP (O v. M), and EL versus MOSP (E v. M) with uniform deadline and size distributions. #jobs is the average number of jobs present at each time step in the system.

Tab. 1 shows that PACE incurs a significant over-consumption w.r.t. MOSP: from 44,4% for $\tau \in \{0, 1\}$ to 76.0% for $\tau \in \{3/4, 1/4\}$. More interestingly, this over-consumption increases when the average number of jobs present simultaneously in the

system increases: this is particularly visible when $\tau \in \{0, 1\}$, *i.e.*, in Tab. 1a. This is expected because PACE has been designed to be optimal in expectation when at most one job is present at each time. In the experiments reported in Tab. 1b, the average number of jobs present in the system is lower because at most one job can arrive at any time ($\tau > 0$), and we see that the performances of PACE are better.

In Tab. 1, we also report the execution times of VI for executing MOSP (Alg. 1). Each reported value is the sum over 100 runs. We observe that, as expected given the increasing of the size of the state space, the execution time grows as the maximum inter-arrival time does. Our implementation is in C++ but it is not particularly optimized for efficiency.

7.2 Impact of Deadlines

We now study the impact of the deadline distribution on the different policies. These experiments are performed for a system with jobs with a *fixed* inter-arrival time τ , that is $\tau = L$. Two cases are investigated: in Tab. 2a we take $\tau = L = 1$, so several jobs may be present simultaneously in the system; while in Tab. 2b we take $\tau = L = 3$, so at most one job may be present in the system at any time because $\Delta \leq L$, which implies that each job finishes before the next job arrival. Job sizes W are uniformly distributed between 1 and 4, the maximal deadline is set to $\Delta = 3$, and the buffer size is $B = 4$.

Deadline distribution			over-consumption			
$d = 1$	$d = 2$	$d = 3$	P v. M	O v. M	E v. M	Algo. 1 execution time
1	0	0	0.0%	0.0%	0.0%	6.51e-05 min
1/2	1/2	0	20.0%	20.4%	20.4%	0.039 min
0	1	0	61.2%	11.3%	0.0%	0.045 min
1/3	1/3	1/3	44.4%	11.0%	10.7%	2.19 min
1/2	0	1/2	65.9%	5.0%	0.3%	2.26 min
0	1/2	1/2	57.4%	11.6%	9.1%	2.29 min
0	0	1	46.7%	6.0%	0.0%	2.36 min

(a) Parameters: $W = 4$, $\tau = L = 1$, $B = 4$.

Deadline distribution			over-consumption			
$d = 1$	$d = 2$	$d = 3$	P v. M	O v. M	E v. M	Algo. 1 execution time
1	0	0	0.0%	0.0%	0.0%	1.11e-03 min
1/2	1/2	0	0.0%	0.0%	0.0%	0.29 min
0	1	0	0.0%	0.0%	0.0%	0.27 min
1/3	1/3	1/3	1.9%	3.7%	0.0%	14.92 min
1/2	0	1/2	2.2%	4.3%	0.0%	12.87 min
0	1/2	1/2	8.6%	17.0%	0.0%	16.48 min
0	0	1	26.3%	52.3%	0.0%	18.61 min

(b) Parameters: $W = 4$, $\tau = L = 3$, $B = 4$.

Table 2: Influence of the deadline distribution on the over-consumption of OA versus MOSP (O v. M), PACE versus MOSP (P v. M), and EL versus MOSP (E v. M) with uniform size distributions.

When all the job deadlines d are equal to 1 (first line of Tabs. 2a and of 2b), the set of admissible speeds \mathcal{A} , computed in Eq. (23), contains only the speeds greater or equal to the maximum job size W . It follows that all the policies select at each instant the speed $\min(\mathcal{A})$, and are therefore optimal.

When $\tau = L = 1$ (Tab. 2a), the over-consumption of OA is less than that of PACE. The situation is the opposite when $\tau = L = 3$ (Tab. 2b). This is because, when $\tau = L = 3$, there is only one job in the buffer at each time instant, so PACE is optimal in expectation. Note that the significant over-consumption of PACE in the deterministic deadline case ($d = 3$, ninth line: 26.3%) is due to its sub-optimal discretization.

When the deadlines d are equal to 3 (seventh line of Tab. 2b), EL benefits from an exact knowledge of the probability distributions and is able to compute the optimal speed (its over-consumption is 0%), because in this particular case we have $d = L = 3$. In contrast, since OA is oblivious of the probability distributions, it results in a significant energy over-consumption (its over-consumption is 52.3%).

More generally, the average power over-consumption of OA is always larger than that of EL for Tab. 2a and 2b. This is an excellent result for EL since its runtime cost is, as OA, very low.

Finally, the overall behavior of EL is very good: when $\tau = L = 1$ its over-consumption is close from the optimal of MOSP (Tab. 2a), while when $\tau = L = 3$ it is optimal (Tab. 2b).

We also report in Tab. 2 the execution times of the VI for executing MOSP (Alg. 1). We observe that, as expected given the increasing of the size of the state space, the execution time grows as the inter-arrival time and the deadline increase. More specifically, we notice the strong impact of the deadline on the execution time: when the deadline increases by one unit, the execution time increases by at least 40 times, whereas when the inter-arrival time increases by one unit, the execution time increases by at least 5 times.

7.3 Impact of Job Sizes

We now study the impact of the size distribution on the different speed policies. All jobs have a fixed deadline $d = \Delta = 3$ and identical maximal size $W = 4$. The buffer size is $B = 3$. To analyze the impact of size distributions, we investigate two cases where jobs are generated by periodic tasks:

1. At each time step, a job arrives in the system. In other words, the inter-arrival time is $\tau = L = 1$ with probability 1.
2. At one time step out of three, a job arrives in the system. The inter-arrival time is $\tau = L = 3$ with probability 1. This condition implies that at most one job is present in the buffer of the processor at each instant, because the inter-arrival time has the same value as the deadline (implicit deadline model).

In this section, the randomness is only on the size of incoming jobs:

$$\mathbb{P}(w = \{1, 2, 3, 4\}) = \left\{ \frac{i_1}{10}, \frac{i_2}{10}, \frac{i_3}{10}, \frac{i_4}{10} \right\} \text{ for each } i_1, i_2, i_3 \in \{0, \dots, 10\}$$

$$\text{and } i_4 \in \{1, \dots, 10\} \text{ and } i_1 + i_2 + i_3 + i_4 = 10. \quad (30)$$

We compute below the over-consumption of the three speed policies OA, PACE, and EL w.r.t. MOSP for all possible distributions satisfying Eq. (30). Figs. 7a, 7b, 7c, and 7d depict the respective over-consumption of OA vs. MOSP, PACE vs. MOSP, EL vs. MOSP, and OA vs. EL, each time with $\tau = L = 1$ (left) and with $\tau = L = 3$ (right) as a function of the mean of the job size. The red curve depicts the average value of the over-consumption. The mean execution time of Algo. 1 for a simulation with $\tau = 1$ is 1.88 minutes with a standard deviation of 0.09 minutes, while for $\tau = 3$ it is 12.58 minutes with a standard deviation of 0.56 minutes. Again, as expected, the computation time is longer for $\tau = 3$ than for $\tau = 1$ (seven times longer).

Fig. 7a, OA vs. MOSP. Whatever the inter-arrival times, the energy consumed with OA converges towards that consumed with MOSP when the mean of the job sizes converges towards W (here equal to 4). This is expected since OA is optimal when the actual size of each job is equal to W .

The over-consumption of OA is smaller when $L = 1$ than with $L = 3$. Indeed, $L = 1$ implies that the system is more heavily loaded, requiring higher processor speeds; therefore, a smaller range of speeds is available for both OA and MOSP. In some sense, this situation leaves less “space” for MOSP to outperform OA.

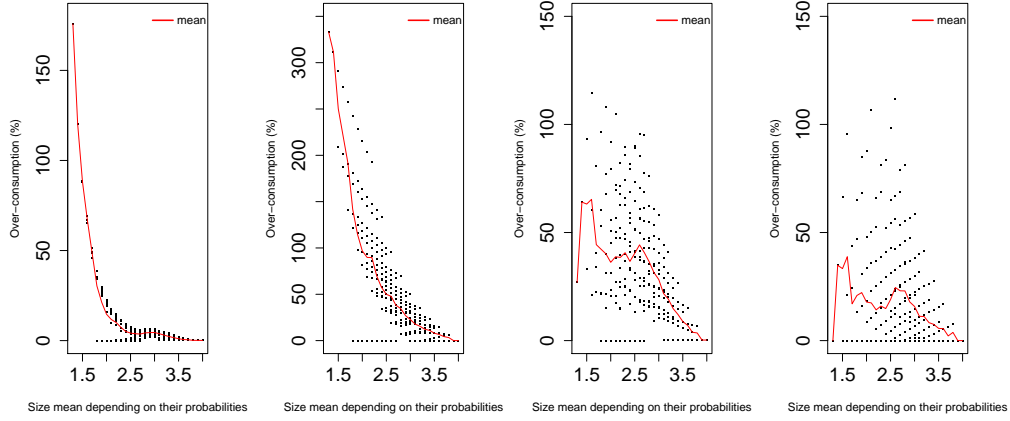
Fig. 7b, PACE vs. MOSP. The results are in line with the policy definition: Indeed, PACE is only optimal for a job in isolation, which is not the case in the left graph. When the inter-arrival time is 1, several jobs can be present in the buffer at the same time. In contrast, when the inter-arrival time is 3 and the deadline is also 3, there is at most one job in the buffer at any time.

Even if PACE is optimal for one job in isolation, we note anyway in the right graph that PACE incurs a mean over-consumption of 15% versus MOSP. The difference comes from the fact that PACE is discretized (which is more realistic in practice, because the processor speeds are finite and the decision times are also discrete).

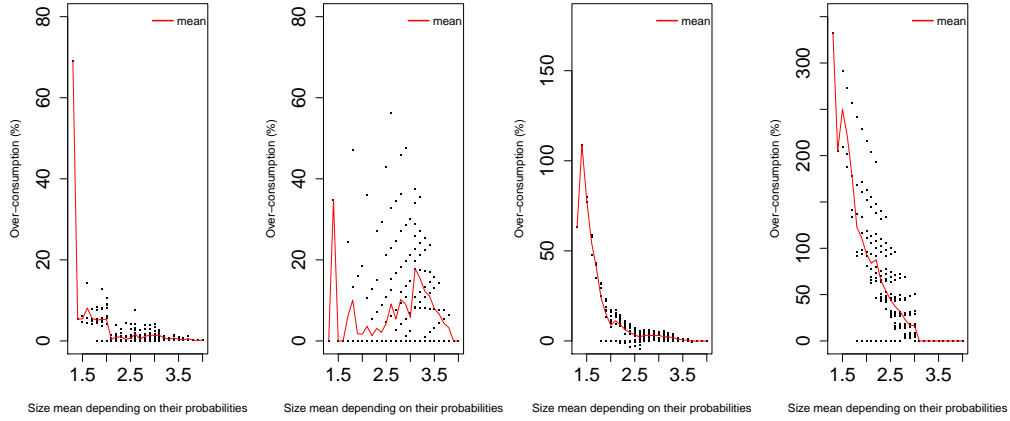
When the inter-arrival time is 3, the size distribution has an important impact on the energy consumption. This is because the speed selected by PACE may be close – or not – to an integer. For example, with $\mathbb{P}(\{w = (1, 2, 3, 4)\}) = \{2/10, 0, 7/10, 1/10\}$, the difference between PACE and MOSP is only 0%. In contrast, with $\mathbb{P}(\{w = (1, 2, 3, 4)\}) = \{6/10, 3/10, 0, 1/10\}$, the over-consumption of PACE is 95.52%. This is due to the speed discretization.

Fig. 7c, EL vs. MOSP. As expected, MOSP performs better on average than EL, but here the average and max over-consumptions are much lower than for PACE and OA.

Indeed, the *maximal over-consumption* when $\tau = 1$ is only 69% for EL, whereas it is 176% for OA and 114% for PACE. When $\tau = 3$, the *maximal over-consumption* is only 56% for EL, whereas it is 333% for OA and 112% for PACE.



(a) OA versus MOSP with $\tau = 1$ (left) and $\tau = 3$ (right). (b) PACE versus MOSP with $\tau = 1$ (left) and $\tau = 3$ (right).



(c) EL versus MOSP with $\tau = 1$ (left) and $\tau = 3$ (right). (d) OA versus EL with $\tau = 1$ (left) and $\tau = 3$ (right).

Fig. 7: Influence of the size distribution on the over-consumption with fixed jobs deadline $d = 3$, fixed inter-arrival time (either $\tau = L = 1$ or $\tau = L = 3$), and a fixed buffer size $B = 3$.

Regarding the average over-consumption, we observe the same behavior. When $\tau = 1$, the *average over-consumption* by size mean depending on their probability is only 4.4% for EL, whereas it is 23% for OA and 30% for PACE. For $\tau = 3$, the *average*

over-consumption by size mean depending on their probability is only 6.7% for EL, whereas it is 81% for OA and 15% for PACE.

Fig. 7d, OA vs. EL. Here, EL performs almost always better on average than OA. More precisely, when the inter-arrival time is 3, EL is better on average than OA for all the job size configurations. When the inter-arrival time is 1, EL is better on average than OA for 94% of the job size configurations.

7.4 Real Life Case Study: Edge Detection Algorithm

As a real-life embedded system, we study an edge detection algorithm. It produces as output a video stream of the image’s edges of its input video stream. This system displays a great variety of jobs whose size depend both on the input images and on the initial state of the hardware. We executed it 1,000 times to build the distribution of the job size on a Macbook Pro with a quad-core Intel Core i7 running at 1.7 GHz and a LPDDR3 memory of 16 GB at 2,133 MHz. The resulting job size distribution is depicted in Fig. 8.

The granularity of the job size has a direct impact on the state space. Here we have chosen a granularity equal to $1,500 \cdot 10^9$ elementary operations for one job size unit: as a result, the job size belongs to the interval [5, 19]. In addition, each job has a deadline $d = 3$ and an inter-arrival $\tau = 3$. With these parameters, the size of the MDP, when applying Algo. 1 is 5,028 states. Finally, by tuning parameter K in EL, we have found the best value to be $K = d_i + 2$.

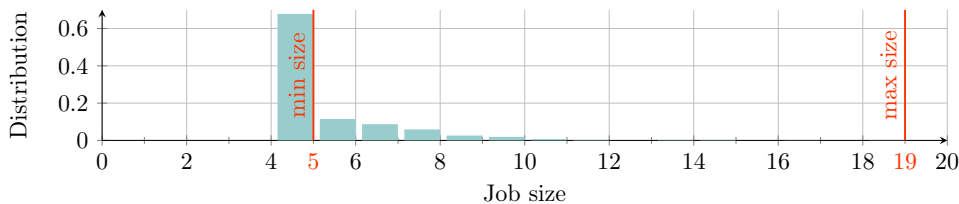


Fig. 8: Distribution of the size for the edge detection algorithm over 1,000 executions (min= 5 and max= 19).

The shape of this distribution of the execution times, skewed towards the smaller end of the spectrum, is classical in hard-real time systems. The reason is that most functions exhibit a usual behavior the execution time of which is close to the average, but also some rare behaviors far from the average.

Regarding the inter-arrival time, the sequence of jobs here is generated by a periodic task. But as soon as several tasks are combined (some periodic, some sporadic), the inter-arrival times will follow no particular distribution. This will also be true for the execution times and the deadlines. Nonetheless, and this is a strength of our paper, MOSP will still achieve the optimal expected energy consumption, as we have proved in Proposition 1.

We compared PACE, OA, and EL against MOSP on this case study, over 1,000 runs of 1,000 time steps each.

- The over-consumption of OA vs. MOSP is 110%: this is because OA’s Eq. (4) uses only the size, but here the mean of the size distribution (5.79) is significantly lower than the maximal size (19).
- The over-consumption of PACE vs. MOSP is 69%: this is because of the discretization of the speeds. PACE has been designed to be optimal when the jobs run in isolation (which is the case here since $d = \tau$), but the closed form solution of Eq. (5) computes *real* values for the speed, while the available speeds are *integer* values. As a final remark, PACE is better than OA, again because jobs are executed in isolation since $d = \tau$.
- The over-consumption of EL vs. MOSP is 0.48%, meaning that EL is very close to the optimal solution. Besides, the over-consumption of OA vs. EL is 109% and the over-consumption of PACE vs. EL is 69%. Of course, MOSP is always better, but it takes several days to compute this speed policy. This is the price of optimality.

8 Extensions

Our MDP framework assumes on the one hand that the time required for context switch between two real-time jobs is null, and on the other hand that the energy and time required to change the operating speed of the processor is null. We discuss in this section how these two assumptions can be relaxed.

8.1 Context switches

There are two types of context switches: (1) when a job is preempted its context must be saved, and (2) when a job is resumed its context must be reloaded. Be it the preemption of a job or the resumption of a job partially executed, any context switch induces a small interval of time where the processor speed is still s but no work is done on the active job. For simplicity, we assume that both require the same amount of *work*¹, denoted η and assumed to be such that $\eta \ll 1$. Accordingly, the *time* required by each context switch is equal to η divided by the current speed s of the processor. This is coherent with our processor model exposed in Sec. 2.1.1. How small should η will is discussed in Subsection 8.1.4 (see Eq. (34)).

Taking non-null context switch into account requires to modify Algo. 1 to incorporate the number of context switches into the computation of the total energy and of the optimal speed, as well as Algo. 2. The idea is to compensate all context switches *within each time-step* $(t, t + 1]$ by slightly accelerating the processor for an adequate duration so that they have no side effect outside this time interval $(t, t + 1]$.

8.1.1 Modification of Algorithm 1

Consider a given time interval $(t, t + 1]$. During this time interval, let s be the selected processor speed and let m be the number of context switches (supposed to be known for now, see Subsection 8.1.2). The processor must therefore execute an additional

¹our approach is also valid when they are distinct by counting the number of resumes and the number of preemptions

amount of work equal to $m\eta$. To compensate for this extra computation inside the time interval, we increase the processor speed to s_{next} (the smallest speed in \mathcal{S} larger than s) during a portion of this time interval. This compensation will incur an over-consumption but will make sure that the total work executed in $(t, t+1]$ will be the same as if context switches were instantaneous. Let $d_{s_{next}}$ be the duration when the processor will operate at speed s_{next} during $(t, t+1]$. To get a perfect compensation, we need to make sure that the work done at speed s_{next} is equal to the work done if switches were instantaneous:

$$\begin{aligned} d_{s_{next}} s_{next} &= m\eta + s d_{s_{next}} \\ \iff d_{s_{next}} &= m\eta / (s_{next} - s). \end{aligned} \quad (31)$$

To account for the effect of this speed increase on the power consumption, the former instantaneous cost \tilde{F} must be replaced by a new instantaneous expected cost $\tilde{\tilde{F}}$ equal to

$$\begin{aligned} \tilde{\tilde{F}} &= \underbrace{(\beta_s(\mathbf{x}) - d_{s_{next}})}_{\text{expected time spent at speed } s} F(s) + \underbrace{d_{s_{next}}}_{\text{time spent at speed } s_{next}} F(s_{next}) + \underbrace{(1 - \beta_s(\mathbf{x}))}_{\text{expected idle time}} P_{idle} \\ &= \left(\beta_s(\mathbf{x}) - \frac{m\eta}{s_{next} - s} \right) F(s) + \frac{m\eta}{s_{next} - s} F(s_{next}) + (1 - \beta_s(\mathbf{x})) P_{idle}. \end{aligned} \quad (32)$$

Notice that the new instantaneous expected cost depends on $\beta_s(\mathbf{x})$ (introduced in Eq.(22)) as previously, but also on m , the number of context switches occurring in the time interval, which is new.

Now, the question is to verify if the number of context switches m during the interval $(t, t+1]$ can be predicted by only knowing the state \mathbf{x} at time t and the current speed s . This is actually not possible. To be able to predict the number of switches, one must also know the next state \mathbf{x}' reached at time $t+1$. Fortunately, the theory of MDP actually covers this case: it suffices to denote $\tilde{\tilde{F}}(\mathbf{x}, s, \mathbf{x}')$ the new instantaneous cost and to change the Bellman equation (20) into:

$$c^* + h^*(\mathbf{x}, \ell) = \min_{s \in \mathcal{A}(\mathbf{x})} \left(\sum_{\substack{(\mathbf{x}', \ell') \in \\ \mathcal{X} \times L}} \mathbf{P}((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) (\tilde{\tilde{F}}(\mathbf{x}, s, \mathbf{x}') + h^*(\mathbf{x}', \ell')) \right). \quad (33)$$

Accordingly, lines 7 and 17 in Algo. 1 must be replaced respectively by:

$$u^{n+1}((\mathbf{x}, \ell)) \leftarrow \min_{s \in \mathcal{A}((\mathbf{x}, \ell))} \left\{ \sum_{\substack{(\mathbf{x}', \ell') \in \\ \mathcal{X} \times \{0, \dots, L\}}} P((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) \left(u^n((\mathbf{x}', \ell')) + \tilde{\tilde{F}}(\mathbf{x}, s, \mathbf{x}') \right) \right\}$$

$$\sigma^{\text{MDP}}((\mathbf{x}, \ell)) \in \arg \min_{s \in \mathcal{A}((\mathbf{x}, \ell))} \left\{ \sum_{\substack{(\mathbf{x}', \ell') \in \\ \mathcal{X} \times \{0, \dots, L\}}} P((\mathbf{x}, \ell), s, (\mathbf{x}', \ell')) \left(u^n((\mathbf{x}', \ell')) + \tilde{F}(\mathbf{x}, s, \mathbf{x}') \right) \right\}$$

8.1.2 Computation of the number of context switches

The remaining question is to show that we can compute the number m of context switches during a given interval $(t, t+1]$ given \mathbf{x} , the state at time t and \mathbf{x}' , the state at time $t+1$. During the time interval $(t, t+1]$, three types of events can happen: some jobs from the list \mathbf{x} may be completed, at most one job may be partially executed and several jobs may arrive at time $t+1$. This induces the following facts:

- (F1) Jobs belonging to both state lists \mathbf{x} and \mathbf{x}' appear in \mathbf{x} as (e_i, d_i) and in \mathbf{x}' as (e'_j, d'_j) with $e'_j = e_i$ (all these jobs have been untouched during $(t, t+1]$) except for at most one job, partially executed in $(t, t+1]$ (for which $e'_j > e_i$) and for all these jobs $d'_j = d_i - 1$ (all deadlines decrease by one).
- (F2) At most one job preemption can take place, at $t+1$, due to a job arrival.
- (F3) The jobs that are resumed or preempted have been partially executed, so they belong to \mathbf{x} or \mathbf{x}' and are of the form (e_i, d_i) or (e'_j, d'_j) with $e_i > 0$ or $e'_j > 0$.

The first partially executed job in \mathbf{x}' will play a crucial role in the following. If such a job exists, it is denoted (e'_{j_1}, d'_{j_1}) and is the same as the job denoted (e_{i_1}, d_{i_1}) in \mathbf{x} .

Example 1. *Let us consider the following example that shows the importance of the crucial job:*

$$\begin{aligned} \mathbf{x} &= (2, 2), (1, 3), (0, 3), (0, 4), (2, 5), (\mathbf{0}, \mathbf{5}), (3, 6) \\ \mathbf{x}' &= (0, 3), (\mathbf{2}, \mathbf{4}), (3, 5), (0, 5) \end{aligned}$$

The first step is to identify the first partially executed job in \mathbf{x}' : It is the second job in \mathbf{x}' , with features $(2, 4)$.

Now, what was its position in \mathbf{x} ? It cannot be $(2, 2)$, $(1, 3)$, $(0, 3)$, or $(0, 4)$ because the deadlines are not good. It cannot be $(2, 5)$ because this would imply that jobs $(0, 5)$ would still be in the list \mathbf{x}' with features $(0, 4)$, after $(2, 4)$. Therefore, the job in \mathbf{x} corresponding to $(2, 4)$ is $(0, 5)$.

From here, we can rebuild the scenario of what happened during the interval $(t, t+1]$ and compute the number of resumes and preemptions unambiguously. The jobs in \mathbf{x} $(2, 2)$, $(1, 3)$, $(0, 3)$, $(0, 4)$, $(2, 5)$ have been completed; two units of work have been done on job $(0, 5)$ but it was still uncompleted at time $t+1$ and two new jobs have arrived at $t+1$ with respective deadlines 3 and 5.

This gives the following number of resumes:

- *no resume for the first job $(2, 2)$, because the resumes and preemption at t are not counted in the semi-open interval $(t, t+1]$;*
- *one resume for job $(1, 3)$;*
- *no resume for job $(0, 3)$ that was unstarted;*
- *no resume for job $(0, 4)$ for the same reason;*

- one resume for job (2, 5);
- no resume for job (0, 5) that was also unstarted.

As for the preemptions, job (0, 3) is \mathbf{x}' arrived at $t + 1$ and preempted (2, 4).

In total, this gives 2 resumes and 1 preemption.

The general case, inspired by this example is described in the following procedure. If there is no crucial job (all jobs in \mathbf{x}' are unstarted) then we have:

Number of resumes: all the jobs (e_i, d_i) in \mathbf{x} that are partially executed have been completed and they all contribute one resume, except the first one because resumes at t are not counted, (i.e. if $e_i > 0$ and $i > 1$).

Number of preemptions: No preemption.

If the crucial job exists (denoted (e_{i_1}, d_{i_1}) in \mathbf{x} and (e'_{j_1}, d'_{j_1}) in \mathbf{x}'), then

Number of resumes: All jobs $\{(e_i, d_i)\}_{(i < i_1)}$ in \mathbf{x} up to the crucial job (excluded) are jobs that have been completed during $(t, t + 1]$. They all contribute one resume if partially executed except for the head job because resumes at time t are not counted (i.e. $e_i > 0$ and $i > 1$). As for the crucial job (e_{i_1}, d_{i_1}) , it has undergone one resume if partially executed and resumed during $(t, t + 1)$ (i.e., if $e'_{j_1} > e_{i_1} > 0$ and $i_1 > 1$), or resumed exactly at $t + 1$ (i.e., if $e'_{j_1} = e_{i_1} > 0$ and $j_1 = 1$).

Number of preemptions: A preemption has occurred at time $t + 1$ if and only if the crucial job namely (e_{i_1}, d_{i_1}) , has been partially executed (i.e., if $e'_{j_1} > e_{i_1}$) and is not the head job in \mathbf{x}' (i.e., if $j_1 > 1$).

8.1.3 Modification of Algorithm 2

As explained in the beginning of this section, we also need to modify Algo. 2 to compensate each context switch in real-time:

1. At line 4, if the new job arrival causes a preemption, then set $m \leftarrow m + 1$.
2. At line 11, if the job completion is followed by a resume then $m \leftarrow m + 1$.
3. At the end of the **foreach** loop, increase the processor speed to s_{next} for a duration $m\eta/(s_{next} - s)$.

8.1.4 Validity issue

A final issue concerns the validity of these modifications. Let $(t, t + 1]$ be the current time interval, and let $g_m \in [0, 1)$ denote the *relative* time instant at which the m^{th} context switch occurs during this time interval.

Intuitively, we need the last context switch not to occur “too late” in the time interval, in order to give the processor enough time to compensate for this loss of work. Formally, this means $1 - g_m > d_{s_{next}} = m\eta/(s_{next} - s)$. Since all jobs have integer size, then $1 - g_m \geq 1/s$ and $m \leq s$. This gives a *sufficient* condition for validity on the size of a context switch:

$$\eta \leq \frac{s_{next} - s}{s^2}. \quad (34)$$

8.2 Non-null speed change costs

A speed change incurs two costs: a power consumption overhead and a time lag to switch from one speed to another. Jointly taking into account these two costs has

already been solved in [30]–Section 5. In a nutshell, the power consumption overhead is simply incorporated in the power function. The time lag to change between two speeds is a bit more tricky: this time lag during which the processor is inoperative (it executes no work) is compensated by modifying the instant when the speed change occurs, such that the amount of executed work during one time interval remains the same. Technically, changing from s_i and s_j means that the processor is operating at s_i during $(t-1, t]$ and at s_j during $(t, t+1]$. When $s_i > s_j$, this behavior is simulated by operating at s_i from t to $t + \alpha_{i,j}$, then at 0 from $t + \alpha_{i,j}$ to $t + \alpha_{i,j} + \eta$, and finally at s_j from $t + \alpha_{i,j} + \eta$ to $t + 1$. The processor is therefore inoperative during a duration equal to η which accounts for the time cost of the speed change from s_i to s_j . The parameter $\alpha_{i,j}$ is computed in such a way that the amount of executed work remains the same (see [30]). The case when $s_i < s_j$ is analogous.

Taking into account the speed changes due to context switches is done as follows. Within one time interval, context switches (not matter how many) only induce two speed changes (see subsection 8.1.3). We therefore operate the processor at s_i from t to $t + \alpha_{i,j}$, then at 0 from $t + \alpha_{i,j}$ to $t + \alpha_{i,j} + 3\eta$, and finally at s_j from $t + \alpha_{i,j} + 3\eta$ to $t + 1$. The processor is therefore inoperative during a duration at most equal to 3η which accounts for three actual speed changes, one due to the speed change from s_i to s_j (as explained in the above paragraph), and two due to the context switches (if any).

9 Conclusion

We have proposed and implemented a Markov Decision Process solution (MOSP) to get the **optimal online speed policy** for the **single-core hard real-time energy minimization problem**, in the **non-clairvoyant case**. To the best of our knowledge, MOSP is the first proposition to be optimal in expectation for this framework. We have also provided counter-examples to prove that the two previous state-of-the-art algorithms, namely OA and PACE, are both sub-optimal here.

Our MOSP speed policy is optimal in terms of energy consumption, but this comes at the price of a significant computation time to build this policy (the offline phase), due to an important state space size. For this reason, we have also proposed a heuristic speed policy, called Expected Load (EL), which uses the statistical information (as MOSP) while having a very cheap offline phase (as OA).

Our simulations show that our MOSP policy outperforms the existing online policies (OA and PACE), and is in particular very efficient when the mean value of the size distribution is far from the worst case. Moreover, simulations establish that PACE is not suitable for the multi-job arrival case, and that our heuristic EL is, in average, quite close to our optimal policy MOSP in this case.

We conclude by sketching several future work directions. A first one is the reduction of the time and space complexity of our algorithm, because it is exponential in the maximal deadline of the jobs, which limits its applicability. A potential solution would be to reduce the state space by merging some “close” states of the MDP. However, this may lead to a sub-optimal solution. It will be interesting to establish sub-optimality bounds formally.

We have seen that our new EL speed policy performs well compared to OA and PACE. In the future, we plan to work on more efficient heuristic speed policies.

Another future work direction involves *unsupervised learning* techniques (such as Q-learning) to estimate the distributions of the features of the real-time jobs as well as to learn the optimal processor speed policy *online*.

Finally, it will be interesting to investigate the *single-processor multi-core* context, and then the *distributed multi-processors* context. This will raise many difficulties, including the relationship between the speed policy and the scheduling policy.

References

- [1] Chen, J., Kuo, C.: Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In: Real-Time Computing Systems and Applications, RTCSA'07, pp. 28–38. IEEE Computer Society, Daegu, Korea (2007)
- [2] Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced CPU energy. In: IEEE Annual Foundations of Computer Science, FOCS'95, pp. 374–382. IEEE, Milwaukee (WI), USA (1995)
- [3] Bansal, N., Kimbrel, T., Pruhs, K.: Speed scaling to manage energy and temperature. *J. ACM* **54**(1), 1–39 (2007)
- [4] Gaujal, B., Girault, A., Plassart, S.: Feasibility of on-line speed policies in real-time systems. *Real Time Syst.* **56**(3), 254–292 (2020)
- [5] Li, M., Yao, F.: An efficient algorithm for computing optimal discrete voltage schedules. *SIAM J. Comput.* **35**, 658–671 (2005)
- [6] Li, M., Yao, F.F., Yuan, H.: An $O(n^2)$ algorithm for computing optimal continuous voltage schedules. In: Annual Conference on Theory and Applications of Models of Computation, TAMC'17. LNCS, vol. 10185, pp. 389–400. Bern, Switzerland (2017)
- [7] Zhong, X., Xu, C.: Energy-aware modeling and scheduling for dynamic voltage scaling with statistical real-time guarantee. *IEEE Trans. Computers* **56**(3), 358–372 (2007)
- [8] Chen, J., Kuo, T.: Voltage scaling scheduling for periodic real-time tasks in reward maximization. In: Proceedings of the 26th IEEE Real-Time Systems Symposium, RTSS'05, Miami (FL), USA, pp. 345–355 (2005)
- [9] AlEnawy, T.A., Aydin, H.: On energy-constrained real-time scheduling. In: Euromicro Conference on Real-Time Systems, ECRTS'04, Catania, Italy, pp. 165–174 (2004)
- [10] Colin, A., Puaut, I.: Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.* **18**(2/3), 249–274 (2000)

- [11] Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: International Workshop on Embedded Software, EMSOFT'01. LNCS, vol. 2211, pp. 469–485. Springer, Tahoe City (CA), USA (2001)
- [12] Kelter, T., Marwedel, P.: Parallelism analysis: Precise WCET values for complex multi-core systems. *Science of Computer Programming* **133**, 175–193 (2017)
- [13] Ballabriga, C., Forget, J., Lipari, G.: Symbolic WCET computation. *ACM Trans. Embedd. Comput. Syst.* **17**(2), 1–26 (2017)
- [14] Davis, R., Altmeyer, S., Indrusiak, L.S., Maiza, C., Nelis, V., Reineke, J.: An extensible framework for multicore response time analysis. *Real-Time Syst.* **54**(3), 607–661 (2018)
- [15] Davis, R., Cucu-Grosjean, L.: A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems* **6**(1), 60 (2019)
- [16] Lorch, J.R., Smith, A.J.: Improving dynamic voltage scaling algorithms with PACE. In: Joint International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS'01, pp. 50–61. ACM, Cambridge (MA), USA (2001)
- [17] Lorch, J.R., Smith, A.J.: PACE: A new approach to dynamic voltage scaling. *IEEE Trans. Computers* **53**(7), 856–869 (2004)
- [18] Barnett, J.A.: Dynamic task-level voltage scheduling optimizations. *IEEE Trans. Computers* **54**(5), 508–520 (2005)
- [19] Xu, R., Xi, C., Melhem, R., Mossé, D.: Practical PACE for embedded systems. In: International Conference on Embedded Software, EMSOFT'04, pp. 54–63. ACM, Pisa, Italy (2004)
- [20] Bini, E., Scordino, C.: Optimal two-level speed assignment for real-time systems. *Int. J. of Embedded Systems* **4**(2), 101–111 (2009)
- [21] Rusu, C., Melhem, R.G., Mossé, D.: Maximizing the system value while satisfying time and energy constraints. In: Real-Time Systems Symposium, RTSS'02, pp. 246–255. IEEE, Austin (TX), USA (2002)
- [22] Gruian, F., Kuchcinski, K.: Uncertainty-based scheduling: Energy-efficient ordering for tasks with variable execution time. In: International Symposium on Low Power Electronics and Design, ISPLED'03, pp. 465–468. ACM, Seoul, Korea (2003)
- [23] Zhang, Y., Lu, Z., Lach, J., Skadron, K., Stan, M.R.: Optimal procrastinating voltage scheduling for hard real-time systems. In: Design Automation Conference,

DAC'05, pp. 905–908. ACM, San Diego (CA), USA (2005)

- [24] Xu, R., Mossé, D., Melhem, R.G.: Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Trans. Comput. Syst.* **25**(4), 9 (2007)
- [25] Xu, R., Melhem, R.G., Mossé, D.: A unified practical approach to stochastic DVS scheduling. In: *International Conference on Embedded Software, EMSOFT'07*, pp. 37–46. ACM, Salzburg, Austria (2007)
- [26] Bandari, M., Simon, R., Aydin, H.: On minimizing expected energy usage of embedded wireless systems with probabilistic workloads. *Sustain. Comput. Informatics Syst.* **11**, 50–62 (2016)
- [27] Yuan, W., Nahrstedt, K.: Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In: *ACM Symp. on Operating System Principles, SOSP'03*, pp. 149–163. ACM, Bolton Landing (NY), USA (2003)
- [28] Pillai, P., Shin, K.G.: Real-time dynamic voltage scaling for low-power embedded operating systems. In: *ACM Symp. on Operating System Principles, SOSP'01*, Banff, Alberta, Canada, pp. 89–102 (2001)
- [29] Aydin, H., Melhem, R.G., Mossé, D., Mejía-Alvarez, P.: Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Computers* **53**(5), 584–600 (2004)
- [30] Gaujal, B., Girault, A., Plassart, S.: Dynamic speed scaling minimizing expected energy consumption for real-time tasks. *J. of Scheduling* **23**(5), 555–574 (2020)
- [31] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in hard real-time environment. *J. ACM* **20**(1), 46–61 (1973)
- [32] Plassart, S.: Online optimization in dynamic real-time systems. (optimisation en ligne pour les systèmes dynamiques en temps-réel). PhD thesis, Univ. Grenoble Alpes, France (2020)
- [33] Puterman, M.L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Series in Probability and Statistics. Wiley, Hoboken, USA (1994)

A Construction of the Probability Transition Matrix

In this section we provide an explicit construction of the transition matrix $\mathbf{P}((\mathbf{x}, \ell_1), s, (\mathbf{y}, \ell_2))$ that gives the probability to go from state (\mathbf{x}, ℓ_1) to state (\mathbf{y}, ℓ_2) over time step $(t, t+1]$, when the processor uses speed s . The matrix \mathbf{P} is a function of the distributions of the inter-arrival times, the sizes, and the deadlines of jobs, when jobs are independent and identically distributed (i.i.d.).

In the non-i.i.d. case, the matrix \mathbf{P} can still be constructed when the jobs form a marked Markov Arrival Process (marked MAP) using a similar construction as for the i.i.d. case for each state of the underlying Markov chain. Since MAPs are dense among all point processes, this construction can approximate any distribution of the jobs arbitrarily closely by using an increasing number of states.

A typical example of a marked MAP is when the jobs come from several periodic tasks. All the jobs of one task have the same probability distribution, but these probabilities differ from task to task. In this case, the current state of the MAP includes the task of the current job, and the transition matrix giving the next state may transition to another task depending on its period and offset.

In the non-i.i.d. case and when the jobs do not form a natural marked MAP, the transition matrix is obtained by simulation, as demonstrated in the case study in Sec. 7.4.

A.1 State Evolution

To describe the evolution of the system state from time t to $t+1$, we formalize the space space \mathcal{X} (*i.e.*, all the possible lists of jobs) and its evolution over time. The evolution of ℓ is trivial and will not be detailed here.

Let $\mathbf{x} = ((e_1^{\mathbf{x}}, d_1^{\mathbf{x}}), \dots, (e_n^{\mathbf{x}}, d_n^{\mathbf{x}}))$ and $\mathbf{y} = ((e_1^{\mathbf{y}}, d_1^{\mathbf{y}}), \dots, (e_n^{\mathbf{y}}, d_n^{\mathbf{y}}))$ be two lists of jobs. We define two binary operations:

- $\mathbf{x} \oplus \mathbf{y}$ returns the sorted union of the two job lists \mathbf{x} and \mathbf{y} . The list is sorted by the job deadlines to comply with the EDF policy.
- $\mathbf{x} \ominus \mathbf{y}$ returns the sorted list of jobs of \mathbf{x} that are not in \mathbf{y} (sorted by the job deadlines). By definition, $\mathbf{x} \subset \mathbf{y} \Rightarrow \mathbf{x} \ominus \mathbf{y} = \emptyset$.

Recall that \mathbf{x}_t denotes the list of jobs present at time t . Formally,

$$\mathbf{x}_t = ((e_1^{\mathbf{x}_t}, d_1^{\mathbf{x}_t}), \dots, (e_n^{\mathbf{x}_t}, d_n^{\mathbf{x}_t})). \quad (35)$$

Let u denote the number of jobs completed during the time interval $(t, t+1]$ with the processor speed s . Accordingly, the $(u+1)^{\text{th}}$ job may have been partially executed, and let r denote the amount of work done on this job. Moreover, no work has been done on the subsequent jobs, that is, the $(u+2)^{\text{th}}$ to the n^{th} jobs. Let $Shift_u$ be the operator that implements all these modifications on the job list \mathbf{x}_t :

$$Shift_u(\mathbf{x}_t, r) = ((e_{u+1}^{\mathbf{x}_t} + r, d_{u+1}^{\mathbf{x}_t} - 1), (e_{u+2}^{\mathbf{x}_t}, d_{u+2}^{\mathbf{x}_t} - 1), \dots, (e_n^{\mathbf{x}_t}, d_n^{\mathbf{x}_t} - 1)), \quad (36)$$

because the first u jobs have been completed, all the jobs remaining in $Shift_u(\mathbf{x}_t, r)$ are one time unit closer to their deadline.

Finally, we have to consider the jobs released at time $t + 1$. Let $\mathbf{a}_{t+1} = (0, d_1), \dots, (0, d_k)$ denote the list of k jobs released at time $t + 1$, ordered by their deadlines. As a conclusion, the next job list \mathbf{x}_{t+1} is:

$$\mathbf{x}_{t+1} = \text{Shift}_u(\mathbf{x}_t, r) \oplus \mathbf{a}_{t+1}. \quad (37)$$

A.2 Probability Transition Matrix

We now show how to construct the transition matrix $\mathbf{P}((\mathbf{x}, \ell_1), s, (\mathbf{y}, \ell_2))$ that gives the probability to go from state (\mathbf{x}, ℓ_1) to state (\mathbf{y}, ℓ_2) over time step $(t, t + 1]$, when the processor uses speed s . Recall that we are in the i.i.d. case:

- The i.i.d. inter-arrival times have a common distribution denoted θ : $\forall 0 \leq t$, $\theta(t) = \mathbb{P}(\tau_i = t)$ for any job J_i .
- The i.i.d. sizes have a common distribution denoted μ : $\forall 1 \leq w \leq W$, $\mu(w) = \mathbb{P}(w_i = w)$ for any job J_i .
- The i.i.d. deadlines have a common distribution denoted δ : $\forall 1 \leq d \leq \Delta$, $\delta(d) = \mathbb{P}(d_i = d)$ for any job J_i .

When all jobs are i.i.d., the transition probability $\mathbf{P}((\mathbf{x}, \ell_1), s, (\mathbf{y}, \ell_2))$ does not depend on t and can be decomposed into several parts, depending on the number of completed jobs. We introduce the probability $Q((\mathbf{x}, \ell_1), u, s, (\mathbf{y}, \ell_2))$ to go from state (\mathbf{x}, ℓ_1) to (\mathbf{y}, ℓ_2) under speed s while u jobs are completed during the time step $(t, t+1]$. By construction, we have:

$$\mathbf{P}((\mathbf{x}, \ell_1), s, (\mathbf{y}, \ell_2)) = \sum_{u=\lfloor s/W \rfloor}^{\min(s, |\mathbf{x}|)} Q((\mathbf{x}, \ell_1), u, s, (\mathbf{y}, \ell_2)). \quad (38)$$

where $|\mathbf{x}|$ is the cardinality of the list \mathbf{x} . In Eq. (38), u varies between $\lfloor s/W \rfloor$ and $\min(s, |\mathbf{x}|)$ because the speed s used by the processor cannot be shared among less than $\lfloor s/W \rfloor$ jobs (if their size is W) and cannot be shared among more than s jobs (if their size is 1), and the total number of jobs present in state (\mathbf{x}, ℓ_1) is $|\mathbf{x}|$.

To compute the probability $Q((\mathbf{x}, \ell_1), u, s, (\mathbf{y}, \ell_2))$, we introduce the random variables $(k_i)_{1 \leq i \leq u}$, where k_i denotes the remaining size of job J_i , and we denote \mathbf{k} the vector $\mathbf{k} = (k_1, \dots, k_u)$.

We then introduce the probability $P_{exec}(\mathbf{k}, \mathbf{x}, u, r)$ that u jobs from \mathbf{x} are completed under speed $s = \sum_{i=1}^u k_i$ and a partial execution of r units of work on the $(u + 1)^{\text{th}}$ job, and the probability $P_{arrival}((\mathbf{x}, \ell_1), u, (\mathbf{y}, \ell_2))$, independent of P_{exec} , that a list \mathbf{a} of jobs arrives in interval $(t, t + 1]$; *i.e.*, \mathbf{a} is the list of new jobs in \mathbf{y} that are not in \mathbf{x} . We distinguish the two following cases.

- If $u = |\mathbf{x}|$, *i.e.*, all jobs from \mathbf{x} are completed, then we have:

$$Q((\mathbf{x}, \ell_1), u, s, (\mathbf{y}, \ell_2)) = \sum_{k_1 + \dots + k_u \leq s} P_{exec}(\mathbf{k}, \mathbf{x}, u, 0) \cdot P_{arrival}((\mathbf{x}, \ell_1), u, (\mathbf{y}, \ell_2)), \quad (39)$$

because, by construction, $k_1 + \dots + k_u$ is the minimum amount of work that must be executed in the current instant to complete all the jobs in \mathbf{x} .

- Otherwise, $u < |\mathbf{x}|$, so we have $\forall 1 \leq i \leq u$ and $\forall 1 \leq k_i \leq W$

$$Q((\mathbf{x}, \ell_1), u, s, (\mathbf{y}, \ell_2)) = \sum_{\substack{k_1 + \dots + k_u = \\ s - (e_{\alpha}^{\mathbf{y}} - e_{u+1}^{\mathbf{x}})}} P_{exec}(\mathbf{k}, \mathbf{x}, u, e_{\alpha}^{\mathbf{y}}) \cdot P_{arrival}((\mathbf{x}, \ell_1), u, (\mathbf{y}, \ell_2)), \quad (40)$$

where α is the index of the first job in the list \mathbf{y} with the same deadline as the first job in the list $Shift_u(\mathbf{x}, e_{\alpha}^{\mathbf{y}} - e_{u+1}^{\mathbf{x}})$. By definition, we have $0 \leq e_{u+1}^{\mathbf{x}} \leq e_{\alpha}^{\mathbf{y}} \leq W - 1$. The work spent on job $u + 1$ during this transition is $r = s - (k_1 + \dots + k_u)$, which is equal to $e_{\alpha}^{\mathbf{y}} - e_{u+1}^{\mathbf{x}}$. The work previously spent on this job was $e_{u+1}^{\mathbf{x}}$, hence adding r gives $e_{u+1}^{\mathbf{x}} + r = e_{\alpha}^{\mathbf{y}}$.

We now compute each probability P_{exec} and $P_{arrival}$. First, $P_{exec}(\mathbf{k}, \mathbf{x}, u, e_{\alpha}^{\mathbf{y}})$ is the probability that the u first jobs of \mathbf{x} are completed and that $e_{\alpha}^{\mathbf{y}}$ units of work have been spent on the $(u + 1)^{\text{th}}$ job. It depends on (1) the probability of k_i , the remaining work executed during $(t, t + 1]$, on job $i \in [1, u]$, and on (2) the probability that the job $u + 1$ has been partially executed, given the work quantity already executed in the past (*i.e.*, before t). We distinguish two cases, $u \neq 0$ and $u = 0$.

- If $u \neq 0$, we have:

$$\begin{aligned} P_{exec}(\mathbf{k}, \mathbf{x}, u, e_{\alpha}^{\mathbf{y}}) &= \left(\prod_{i=1}^u \mathbb{P}(w_i = k_i + e_i^{\mathbf{x}} \mid w_i > e_i^{\mathbf{x}}) \right) \cdot \mathbb{P}(w_{u+1} > e_{\alpha}^{\mathbf{y}} \mid w_{u+1} > e_{u+1}^{\mathbf{x}}) \\ &= \frac{\prod_{i=1}^u \mu(k_i + e_i^{\mathbf{x}})}{\sum_{k=e_i^{\mathbf{x}}}^W \mu(k)} \cdot \frac{\sum_{k=e_{\alpha}^{\mathbf{y}}}^W \mu(k)}{\sum_{k=e_{u+1}^{\mathbf{x}}}^W \mu(k)}. \end{aligned} \quad (41)$$

- Otherwise $u = 0$ and we have:

$$P_{exec}(\mathbf{k}, \mathbf{x}, 0, e_{\alpha}^{\mathbf{y}}) = \frac{\sum_{k=e_{\alpha}^{\mathbf{y}}}^W \mu(k)}{\sum_{k=e_{u+1}^{\mathbf{x}}}^W \mu(k)}. \quad (42)$$

Regarding $P_{arrival}((\mathbf{x}, \ell_1), u, (\mathbf{y}, \ell_2))$, it is the probability that the new job arrival is the list $\mathbf{a} = ((0, d_i))_{i=1..n}$, list which is computed as:

$$\mathbf{a} = \mathbf{y} \ominus Shift_u(\mathbf{x}, e_{\alpha}^{\mathbf{y}} - e_{u+1}^{\mathbf{x}}). \quad (43)$$

Eq. (43) returns a list of new jobs present in the new list of jobs \mathbf{y} and not present in the previous list \mathbf{x} , so they must be fresh arrivals.

To compute the probability $P_{arrival}$, we introduce the following notations:

- $n_{\mathbf{y}}$ is the number of new jobs in \mathbf{y} .
- k_d is the number of different job deadlines in the set of the new jobs in \mathbf{y} .
- n_{d_i} is the number of jobs having d_i as their deadline. By construction it satisfies $n_{\mathbf{y}} = \sum_{i=1}^{k_d} n_{d_i}$.
- $succ(\mathbf{x})$ is the set of all possible successors of \mathbf{x} .

- $M_u(\mathbf{x})$ is the maximal number of new jobs in any successor state \mathbf{y} of \mathbf{x} if u jobs have been completed during this time step: $M_u(\mathbf{x}) = \max_{\mathbf{y} \in \text{succ}(\mathbf{x})} |\mathbf{a}| = \max_{\mathbf{y} \in \text{succ}(\mathbf{x})} |\mathbf{y} \ominus \text{Shift}_u(\mathbf{x}, e_{\alpha}^{\mathbf{y}} - e_{u+1}^{\mathbf{x}})|$. Taking into account the maximal capacity buffer of size B (see Sec. 2.1.1), we have $M_u(\mathbf{x}) = B - (|\mathbf{x}| - u)$.

The function P_{arrival} satisfies different properties that are stated below. Two cases must be considered. First, if $\mathbf{y} \notin \text{succ}(\mathbf{x})$, then \mathbf{y} is not a possible successor for \mathbf{x} , so we have $P_{\text{arrival}} = 0$. Second, if $\mathbf{y} \in \text{succ}(\mathbf{x})$, then we distinguish several sub-cases, depending on the set of the new jobs $\mathbf{a} = \mathbf{y} \ominus \text{Shift}_u(\mathbf{x}, e_{\alpha}^{\mathbf{y}} - e_{u+1}^{\mathbf{x}})$.

Case $\mathbf{a} = \emptyset$: No new job arrives.

- If $\ell_2 \neq (\ell_1 + 1) \wedge L$, then $P_{\text{arrival}} = 0$. Indeed, if no new job arrives between \mathbf{x} and \mathbf{y} , then the time of the last job arrival is the same for the two states \mathbf{x} and \mathbf{y} . Thus, if the inter-arrival time of \mathbf{x} is ℓ_1 , then the inter-arrival time of \mathbf{y} has to be equal to $\ell_1 + 1$, and so if $\ell_2 \neq (\ell_1 + 1) \wedge L$ and $\mathbf{a} = \emptyset$, then the configuration is not possible and P_{arrival} is null.
- If $\ell_2 = (\ell_1 + 1) \wedge L$, then $P_{\text{arrival}} = \mathbb{P}(\tau \geq \ell_2 | \tau \geq \ell_1) = \frac{\mathbb{P}(\tau \geq \ell_2)}{\mathbb{P}(\tau \geq \ell_1)} = 1 - \frac{\theta(\ell_1)}{\sum_{i=\ell_1}^L \theta(i)}$.
Indeed, since the inter-arrival time of \mathbf{x} and \mathbf{y} are possible, the situation where $\mathbf{a} = \emptyset$ is possible depends on the inter-arrival time distribution. Since $\mathbf{a} = \emptyset$, then it means that no job arrive in ℓ_1 . Thus P_{arrival} is equal to the probability that a job arrives strictly after ℓ_1 .

Case $\mathbf{a} \neq \emptyset$: One or more new jobs arrive.

- If $\ell_2 \neq 0$, then $P_{\text{arrival}} = 0$ (the time elapsed since the latest arrival must be reset to 0).
- If $\ell_2 = 0$, then using W (the biggest job size), the general case for the probability P_{arrival} is presented below $\forall 1 \leq n_{\mathbf{y}} \leq M_u(\mathbf{x})$.

Let us analyze the most general case: $\mathbf{a} \neq \emptyset$ and $\ell_2 = 0$. In a first step, we suppose that the number of new jobs that arrive does not lead to a full buffer, *i.e.*, $n_{\mathbf{y}} < M_u(\mathbf{x})$. We begin by analyzing the inter-arrival time values. In this situation, since there is at least one arriving job ($\mathbf{a} \neq \emptyset$), it means that we have to consider the inter-arrival time ℓ_1 , which is chosen among all the possible inter-arrival times, *i.e.*, all values between ℓ_1 and L . This probability, which is the probability that the first job of \mathbf{a} arrives, is $\frac{\theta(\ell_1)}{\sum_{i=\ell_1}^L \theta(i)}$. Each additional job in \mathbf{a} depends on the probability of the inter-arrival time to be null, because these jobs must arrive simultaneously. For each of them, the arrival probability is $\theta(0)$. Since there are $n_{\mathbf{y}} - 1$ job arrivals after the first job in \mathbf{a} , the probability for all these jobs is $\theta(0)^{n_{\mathbf{y}} - 1}$. Finally, since there are exactly $n_{\mathbf{y}}$ new jobs, we multiply by the probability of non zero inter-arrival time $(1 - \theta(0))$ for the next job arrival. Our first factor in the expression of P_{arrival} in this case is therefore

$$\frac{\theta(\ell_1) \theta(0)^{n_{\mathbf{y}} - 1} (1 - \theta(0))}{\sum_{i=\ell_1}^L \theta(i)}. \quad (44)$$

Regarding the deadlines, the $|\mathbf{a}|$ job deadlines are independent, so the probability to have n_{d_i} jobs of deadline d_i is $\delta(d_i)^{n_{d_i}}$. Considering all the possible deadlines yields

the second factor

$$\prod_{i=1}^{k_d} \delta(d_i)^{n_{di}}. \quad (45)$$

But since jobs $(0, d_i)$ with the same deadline d_i are not ordered in the set \mathbf{a} , the product in Eq. (45) captures several cases that correspond to the same state. Since there are $n_{di}!$ possibilities for the truncated list of new jobs of deadline d_i , we have to consider all possible combinations of jobs divided by all the combinations for jobs of same deadline, because those are not ordered. This is obtained by multiplying the probability by a third factor equal to $n_{\mathbf{y}}! / \prod_{i=1}^{k_d} n_{di}!$.

These analyses lead to the following job arrival probability for $\mathbf{a} \neq \emptyset$ and $\ell_2 = 0$, in the case where $n_{\mathbf{y}} < M_u(\mathbf{x})$:

$$P_{arrival}((\mathbf{x}, \ell_1), u, (\mathbf{y}, \ell_2)) = \frac{\theta(\ell_1) \theta(0)^{n_{\mathbf{y}}-1} (1 - \theta(0))}{\sum_{i=\ell_1}^L \theta(i)} \cdot \prod_{i=1}^{k_d} \delta(d_i)^{n_{di}} \cdot \frac{n_{\mathbf{y}}!}{\prod_{i=1}^{k_d} n_{di}!}. \quad (46)$$

In the particular case where $n_{\mathbf{y}} = M_u(\mathbf{x})$, Eq. (46) is simplified by removing the probability that there is not an extra $(|\mathbf{a}| + 1)^{\text{th}}$ job:

$$P_{arrival}((\mathbf{x}, \ell_1), u, (\mathbf{y}, \ell_2)) = \frac{\theta(\ell_1) \theta(0)^{n_{\mathbf{y}}-1}}{\sum_{i=\ell_1}^L \theta(i)} \cdot \prod_{i=1}^{k_d} \delta(d_i)^{n_{di}} \cdot \frac{n_{\mathbf{y}}!}{\prod_{i=1}^{k_d} n_{di}!}. \quad (47)$$

Putting everything together gives the transition probability \mathbf{P} .