Dynamic Speed Scaling Minimizing Expected Energy Consumption for Real-Time Tasks

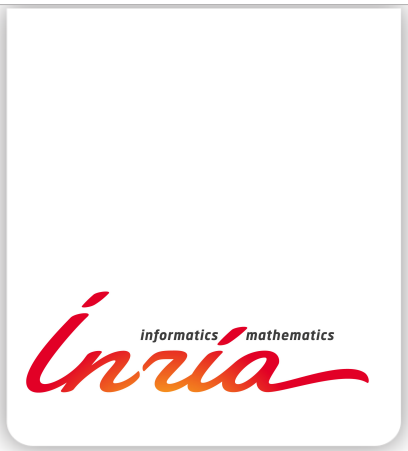
Bruno Gaujal, Alain Girault, Stéphan Plassart

Dynamic Speed Scaling Minimizing Expected Energy Consumption for Real-Time Tasks

GAUJAL Bruno, GIRAULT Alain, PLASSART Stephan

RESEARCH REPORT
N° 9101
September 2017
Project-Teams Polaris and Spades

ISSN 0249-6399 ISRN INRIA/RR--9101--FR+ENG



Dynamic Speed Scaling Minimizing Expected Energy Consumption for Real-Time Tasks

Gaujal Bruno, Girault Alain, Plassart Stephan *

Project-Teams Polaris and Spades

Research Report n° 9101 — September 2017 — 35 pages

Abstract: This paper proposes a Markov Decision Process (MDP) approach to compute the optimal on-line speed scaling policy to minimize the energy consumption of a processor executing a finite or infinite set of jobs with real-time constraints. The policy is computed off-line but used on-line. We provide several qualitative properties of the optimal policy: monotonicity with respect to the jobs parameters, comparison with on-line deterministic algorithms. Numerical experiments show that our proposition performs well when compared with off-line optimal solutions and outperforms on-line solutions oblivious to statistical information on the jobs. Several extensions are also explained when speed changes as well as context switch costs are taken into account. Non-convex power functions are also taken into account to model leakage. Finally, state space reduction using a coarser discretization is presented to deal with the curse of dimensionality of the MDP.

Key-words: Optimization, Real Time System, Markov Decision Process, Dynamic Voltage Scaling

* This work has been partially supported by the LabEx PERSYVAL-Lab.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Sélection en-ligne de la vitesse minimisant l'énergie dans les systèmes temps-réels

Résumé : Cet article propose d'utiliser la technique des processus de décision markovien (PDM) pour calculer la politique optimale en-ligne de choix de vitesses afin de minimiser l'énergie consommée par un processeur exécutant un ensemble de tâches avec des contraintes temps-réel. La politique est calculée avant l'exécution du système temps réel (hors ligne), mais utilisée en ligne. Cette méthode est efficace et proche de la solution optimale hors-ligne et elle est plus performante que les solutions en-ligne qui ne prennent pas en compte les informations statistiques sur les tâches futures.

Mots-clés : Optimisation, système temps réel, processus de décision markovien, ajustement dynamique de la fréquence

1 Introduction

Minimizing the energy consumption of embedded system is becoming more and more important. This is due to the fact that more functionalities and better performances are expected from such systems, together with a need to limit the energy consumption, mainly because batteries are becoming the standard power supplies.

The starting point of this work is the seminal paper of Yao et al. [1] followed by the paper of Bansal et al. [2], which both solve the following problem: Let $(r_i, c_i, d_i)_{i\in\mathbb{N}}$ be a set of jobs, where r_i is the release date (or *arrival time*) of job i, c_i is its *WCET* (or *workload*) *i.e.*, its execution time by the processor operating at its maximal speed, and d_i is its relative *deadline*, *i.e.*, the amount of time given to the processor to execute job i.

The problem is to choose the speed[1] of the processor as a function of time $s(t)$ such that the processor can execute all jobs before their deadlines, and such that the total energy consumption J is minimized. In our problem, J is the *dynamic* energy consumed by the processor: $J = \int_0^T j(s(t))dt$, where T is the time horizon of the problem (in the finite case) and $j(s)$ is the power consumption when the speed is s.

This problem has been solved in Yao et al. [1] in the *off-line* case, *i.e.*, when *all* jobs are known in advance, and when the power function $j(\cdot)$ is a *convex* function of the speed. But of course, the off-line case is unrealistic.

Several solutions for the *on-line* case (only the jobs released at or before time t can be used to select the speed $s(t)$) have been investigated by Bansal et al. in [2]. The authors prove that the on-line algorithm, called Optimal Available (OA), has a competitive ratio of α^α when the power dissipated by the processor working at speed s is of the form $j(s) = s^\alpha$. In CMOS circuits, the value of α is typically 3. In this case, (OA) may spend 27 times more energy than an optimal schedule in the worst case. The principle of the on-line algorithm (OA) is to choose, at each time t, the *smallest* processor speed such that all jobs released at or before time t meet their deadline, under the assumption that no more jobs will arrive after time t.

However, the assumption made by (OA) is questionable. Indeed, the speed selected by (OA) at time t will certainly need to be compensated (*i.e.*, increased) in the future due to jobs released after t, incurring an energetic inefficiency when the j function is convex. In contrast, our intuition is that the best choice is to select a speed *above* the one used by (OA) to *anticipate* on those future job arrivals.

The goal of the paper is to give a precise solution to this intuition by using statistical knowledge of the job arrivals (that could be provided by the user) to select the speed that optimizes the *expected* energy consumption. Other constructions based on statistical knowledge have been done in [3, 4] in a simpler framework, namely for one single job whose execution time is uncertain, or in [5] by using heuristic schemes. In this paper, we show that this general constrained optimization problem can be modeled as an unconstrained Markov Decision Process (MDP) by choosing a proper state space that also encodes the constraints of the problem. In particular, this implies that the optimal speed at each time can be computed using a *dynamic programming* algorithm and that the optimal speed at any time t will be a deterministic function of the current state at time t.

[1] Different communities use the term "speed" or "frequency", which are equivalent for a processor. In this paper, we use the term "speed".

Application in Practical Scenarios

Our approach is usable in several practical cases. The first one concerns real-time systems whose tasks are *sporadic*, with no *a priori* structure on the job release times, sizes, and deadlines. In such a case, a long observation of the jobs features can be used to estimate the statistical properties of the jobs: distribution of the inter-release times, distribution of the job sizes, and deadlines.

Another case where our approach is efficient is for real-time systems consisting of several *periodic* tasks, each one with some randomly missing jobs. The uncertainty on the missing jobs may be due, for example, to faulty sensors and/or electromagnetic interference causing transmission losses.

These two cases are explored in the experimental section where our solution is compared with current solutions. Our numerical simulations report a 5% improvement in the sporadic tasks case, and 30% to 50 % improvement in the periodic tasks case.

Outline of the Paper

In the first part of this paper (§ 2), we present our job model and the problem addressed in the paper. In a second part (§ 3), we construct a Markov decision process model of this problem. We investigate its state space and its complexity (§ 2.3), we propose an explicit *dynamic programming* algorithm to solve it when the number of jobs is finite (§ 3.1), and a *Value Iteration* algorithm [6] for the infinite case (§ 3.2). We prove that both solutions have stochastic monotonicity properties with respect to the sizes and the deadlines of the jobs (§ 3.4). Finally, we compute numerically the optimal policy in the finite and infinite horizon cases, and compare its performance with off-line policies and "myopic" policies like (OA), oblivious to the arrival of future jobs (§ 4).

2 Presentation of the problem

2.1 Jobs, Processor Speeds, and Power

We consider a real-time system with one uni-core processor that executes real-time jobs, sporadic and independent. Each job i is defined by 3 integer values: (r_i, c_i, d_i) where r_i is the release time, c_i is the WCET, and d_i is the relative deadline. We assume that all jobs have a bounded size and a bounded deadline:

$$\exists C \text{ s.t. } \forall i, c_i \leq C, \qquad \exists \Delta \text{ s.t. } \forall i, d_i \leq \Delta.$$

We further assume that several jobs may arrive simultaneously but that the total work brought to the processor at time t is also bounded by C.

The CPU processing speed $s(t)$ can vary in time over a finite number of rational speeds between 0 and s_{max}: $s(t) \in \mathcal{S} = \{0, s_1, ..., s_k, s_{\max}\}$, $s_{\max}$ being the maximal processor speed.

With no loss of generality, we will scale the speeds (as well as the WCETs), so that $s_1, ..., s_k, s_{\max}$ are all integer numbers. Typically, a set of admissible speeds $\{0, 1/4, 1/2, 3/4, 1\}$ is replaced by the set $\{0, 1, 2, 3, 4\}$, while all WCETs are multiplied by 4. This is done to avoid manipulating rational numbers.

We consider that the power dissipated by the CPU working at speed $s(t)$ at time t is $j(s(t))$, so that the energy consumption of the processor from time 0 to time T is $J = \int_0^T j(s(t))dt$. Classical choices for the power consumption j are convex increasing functions of the speed (see [1, 2]), based on classical models of CMOS circuits for power dissipation, or star-shaped functions [7] to further take into account static leakage. Here, the function j is *arbitrary*. However several structural properties of the optimal speed selection will only hold when the function j is convex. In the numerical experiments (§ 4), several choices of j are used, to take into account different models of power consumption.

For the sake of simplicity, we only consider the following simple case: preemption time is null, speed changes are instantaneous, and the power consumption function $j(.)$ is convex. However, preemption times, time lags for speed changes as well as non-convex energy costs can be taken into account with minimal adaptation to the current model. A detailed description of all these generalizations is available in appendix of this report.

2.2 Problem Statement

The objective is to choose at each time t the speed $s(t)$, in order to minimize the total energy consumption over the total time horizon, while satisfying all the real-time constraints. Furthermore, the choice must be made on-line, *i.e.*, it can only be based on past and current information. In other words, only the jobs released at or before time t are known.

The information (or history) $\mathcal{H}(t)$ at time t is the set of all the past and current jobs together with the past speeds selection:

$$\mathcal{H}(t) = \{(r_i, c_i, d_i) | r_i \leq t\} \cup \{s(u), u \leq t\} \tag{1}$$

Notice that in this model, unlike in [3, 4], the workload c_i and the deadline d_i are known at the release time of job i [2].

The on-line energy minimization problem $(\mathcal{P})$ is:

Find online speeds $s(t)$ (i.e. $s(t)$ can only depend on the history $\mathcal{H}(t)$) and a scheduling policy $R(t)$ to minimize $\mathbb{E}\int_0^T j(s(t))dt$ under the constraint that no job misses its deadline.

Since all release times and job sizes are integer numbers, the information $\mathcal{H}(t)$ only changes at integer points, if we consider that the speed $s(t)$ can only change at integer points. In the following we focus on integer times $t \in \mathbb{N}$.

Let (s^*, R^*) be an optimal solution to problem $(\mathcal{P})$. Since the energy consumption does not depend on the schedule (preemption is assumed to be energy-free) and since the *Earliest Deadline First* (EDF) scheduling policy is optimal for feasibility, then (s^*, EDF) is also an optimal solution to problem $(\mathcal{P})$. In the following, we will always assume with no loss of optimality that the processor uses EDF to schedule its jobs. This implies that the only useful information to compute the optimal speed at time t, out of the whole history $\mathcal{H}(t)$, is simply the *remaining work*. The remaining work at time t is an increasing function $w_t(\cdot)$ such that $w_t(u)$ is the amount of work that remains to be done before time $t + u$. Since all available speeds, job sizes and deadlines are integer numbers, the remaining work $w_t(u)$ is an integer valued *càdlàg* staircase function.

This is illustrated in Figure 1 that shows the set of jobs released just before $t = 4$, namely $J_1 = (0, 2, 4)$, $J_2 = (1, 1, 5)$, $J_3 = (2, 2, 6)$, $J_4 = (3, 2, 4)$, $J_5 = (4, 0, 6)$, as well as the speeds chosen

[2] When the actual workload can be smaller than WCET, our approach still applies by modifying the state evolution Eq. (2), to take into account early termination of jobs.

by the processor up to time $t=4$: $s_0=1$, $s_1=0$, $s_2=2$, $s_3=1$. Function $A(t)$ is the amount of work that has arrived before time t. Function $D(t)$ is the amount of work that must be executed before time t. This requires a detailed explanation: the first step of $D(t)$ is the deadline of J_1 at $t=0+4=4$; the second step is for J_2 at $t=1+5=6$; the third step is for J_4 at $t=3+4=7$; the fourth step is for J_3 at $t=2+6=8$. Hence the step for J_4 occurs *before* the step for J_3. This is because Figure 1 depicts the situation at $t=4$. At $t=3$ we would only have seen the step for J_3. Finally, function $e(t)$ is the amount of work already executed by the processor at time t; in Figure 1, the depicted function $e(t)$ has been obtained with an arbitrary policy (*i.e.*, non optimal). Finally, the remaining work function $w_t(u)$ is exactly the portion of $D(t)$ that remains "above" $e(t)$. In Fig. 1, we have depicted in red the staircase function $w_t(u)$ for $t=4$.

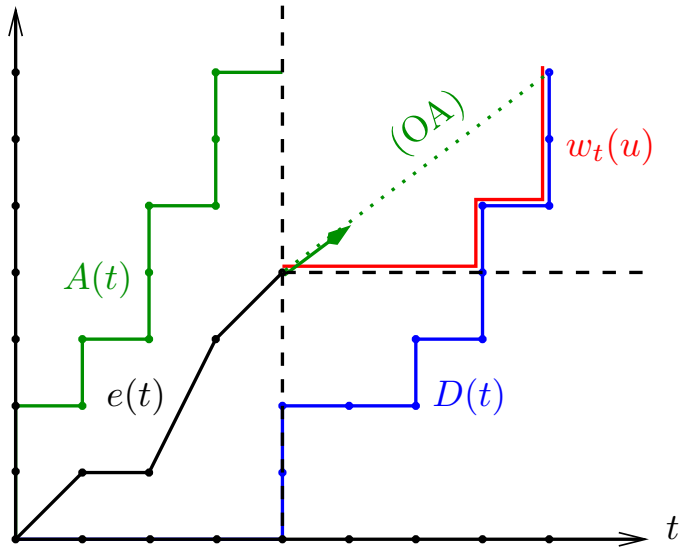


Figure 1: Construction of the remaining work function $w_t(\cdot)$ at $t=4$, for jobs $J_1=(0,2,4)$, $J_2=(1,1,5)$, $J_3=(2,2,6)$, $J_4=(3,2,4)$, $J_5=(4,0,6)$, and processor speeds $s_0=1, s_1=0, s_2=2, s_3=1$. $A(t)$ is the amount of work that has arrived before time t. $D(t)$ is the amount of work that must be executed before time t. $e(t)$ is the amount of work already executed by the processor at time t.

Remark 1. *The on-line algorithm Optimal Available (OA) mentioned in the introduction is also based on the remaining work function: The speed of the processor at time t is the smallest slope of all linear functions above w_t. This is illustrated in Figure 1: the speed that (OA) would choose at time $t=4$ is the slope of the green dotted line marked (OA); in the discrete speeds case (finite number of speeds), the chosen speed would be the smallest available speed just above the green dotted line.*

Since the remaining work function w_t is the only relevant information at time t, out of the whole history $\mathcal{H}(t)$, needed by the processor to choose its next speed, we call w_t the *state* of the system at time t.

2.3 Description of the State Space

To formally describe the state space $\mathcal{W}$ (*i.e.*, all the possible remaining work functions) and the evolution of the state over time, we introduce several constructors.

Definition 1. *We define the following operators:*

– *The time shift operator* $\mathbb{T}f$ *is the shift on the time axis of function* f*, defined as:* $\forall t \in \mathbb{R}, \quad \mathbb{T}f(t) = f(t+1)$.

– *The positive part of a function* f *is* $f^+ = max(f, 0)$.

– *The unit step function (Heaviside function), denoted* H_t*, is the discontinuous step function such that* $\forall u \in \mathbb{R}$*:*

$$H_t(u) = \begin{cases} 0 & \text{if } u < t \\ 1 & \text{if } u \geq t \end{cases}$$

Lemma 1. *Let* (r_n, c_n, d_n) *be a job that arrives at time* $t = r_n$*. If the processor speed at time* $t-1$ *is* s_{t-1}*, then at time* t *the remaining work function becomes:*

$$w_t(\cdot) = \mathbb{T}\big[(w_{t-1}(\cdot) - s_{t-1})^+\big] + c_n H_{d_n}(\cdot) \tag{2}$$

Proof. Between $t-1$ and t, the processor working at speed s_{t-1} executes s_{t-1} amount of work, so the remaining work decreases by s_{t-1}. The remaining work cannot be negative by definition, hence the term $(w_{t-1}(\cdot) - s_{t-1})^+$. After a time shift by one unit, a new job is released at time t, bringing c_n additional work with deadline $t + d_n$, hence the additional term $c_n H_{d_n}(\cdot)$. □

Let us illustrate the state change over an example. In Fig. 2, a new task arrival leads to a modification of the remaining work. The red line corresponds to the previous remaining work function in r_n, and the blue line corresponds to the new remaining work function. This new state has been modified by the job arrival (r_n, c_n, d_n). A quantity s_{n-1} of work has been executed by the processor.

The state space is finite, as stated by the following proposition. This proposition also gives the size of the state space, which will play a major role in the complexity of the dynamic programming algorithm to compute the optimal speed.

Proposition 1. *If* C *is the maximal size of a job and* Δ *its maximal deadline, then the size* $Q(C, \Delta)$ *of the state space* $\mathcal{W}$ *is:*

$$Q(C, \Delta) = \sum_{y_1=0}^{C} \sum_{y_2=y_1}^{2C} \sum_{y_3=y_2}^{3C} \cdots \sum_{y_\Delta=y_{\Delta-1}}^{\Delta C} 1. \tag{3}$$

It can be computed in closed form[3]*:*

$$Q(C, \Delta) = \frac{1}{1 + C(\Delta+1)} \binom{(C+1)(\Delta+1)}{\Delta+1} \tag{4}$$

$$\approx \frac{e}{\sqrt{2\pi}} \frac{1}{(\Delta+1)^{3/2}} (e\,C)^\Delta \tag{5}$$

[3]In Eq. (4), the notation $\binom{n}{k}$ is the binomial coefficient.

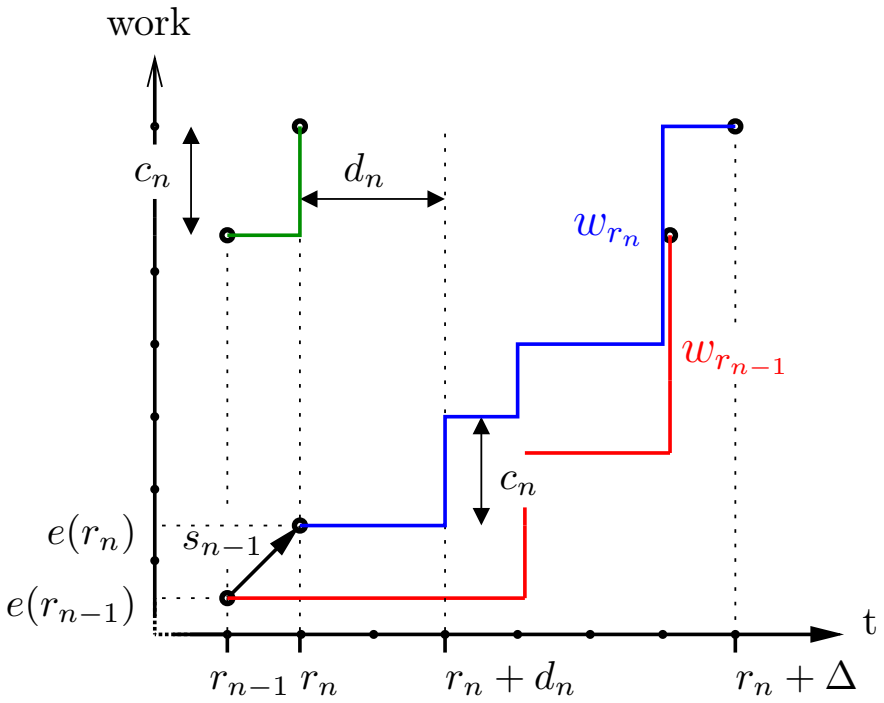


Figure 2: Task arrival at time r_n and the corresponding state change. The red line corresponds to the previous remaining work function. The blue line corresponds to the new remaining work function.

Proof. Let $w(.)$ be a valid state of the system, as time t. Since all parameters are integer numbers and the maximum deadline of a task is Δ, the maximal look-ahead at any time is Δ, hence $w(.)$ is characterized by its first Δ integer values (that are non-decreasing by definition): $w(1) \leq \cdots \leq w(\Delta)$.

Let us define the step sizes of w, starting from the end: $x_1 = w(\Delta) - w(\Delta - 1)$, and more generally, $x_j = w(\Delta - j + 1) - w(\Delta - j)$, for all $j = 1, \ldots, \Delta$, the released work being of maximal size C at any time, $x_1 \leq C$ because x_1 must be bounded by the amount of work that was released at step t. Similarly, $x_1 + x_2$ must be bounded by the amount of work that was released at steps t and $t-1$, namely $2C$, and so on and so forth up to $x_1 + x_2 + \cdots + x_\Delta \leq \Delta C$. This is the only condition for a function w to be a possible state when deadlines and sizes are arbitrary integers bounded by Δ and C respectively.

Therefore $(x_1, x_2, ..., x_\Delta)$ satisfy the following conditions :

$$\begin{cases} x_1 \leqslant C \\ x_1 + x_2 \leqslant 2C \\ x_1 + x_2 + x_3 \leqslant 3C \\ \vdots \\ x_1 + x_2 + ... + x_\Delta \leqslant \Delta C \end{cases}$$

By defining the partial sums $y_j = x_1 + \cdots + x_j$, the number of states satisfies:

$$Q(C, \Delta) = \sum_{y_1=0}^{C} \sum_{y_2=y_1}^{2C} \sum_{y_3=y_2}^{3C} \cdots \sum_{y_\Delta=y_{\Delta-1}}^{\Delta C} 1.$$

The computation of this multiple sum in closed form uses a combinatorial trick inspired by the computation of Catalan numbers. First, notice that a state characterized by its steps $(x_1, \ldots, x_\Delta)$ is in bijection with a path on the integer grid from $(0, 0)$ to $(\Delta+1, C(\Delta+1))$, which remains *below* the diagonal (see Fig. 3).

Second, counting the number of such paths is done in the following way. First, let us count the number of paths from $(0, 0)$ to $(\Delta+1, C(\Delta+1))$ without the constraint of staying below the diagonal. This is a standard computation: this number is equal to $\binom{(C+1)(\Delta+1)}{\Delta+1}$.

Third, this set of paths can be partitioned into *classes* according to the number of vertical steps taken above the diagonal. We are interested in computing the size of class 0. Using a shift (similar to what is done for Catalan numbers), a path of class k can be bijectively transformed into a path of class $k-1$. This means that all classes have the same size. Therefore, the class 0 has size $Q(C, \Delta) = \frac{1}{1+C(\Delta+1)}\binom{(C+1)(\Delta+1)}{\Delta+1}$.

Using the Stirling formula, we finally get $Q(C, \Delta) \approx \frac{e}{\sqrt{2\pi}} \frac{1}{(\Delta+1)^{3/2}} (e\, C)^\Delta$. □

3 Markov Decision Process Solution

Since the state space $\mathcal{W}$ is finite, one can effectively compute the optimal speeds in each possible state. In this section, we provide such algorithms to compute that optimal speed selection in two cases: when the time horizon is finite and when it is infinite. In the finite case, we minimize the *total* energy consumption, while in infinite case we minimize the *average* energy consumed per time unit.

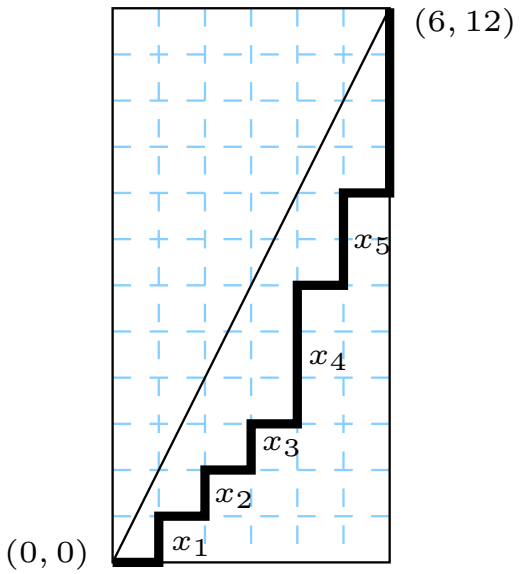


Figure 3: A valid state seen as a path below the diagonal $(0,0)-(6,12)$, for $C=2$ and $\Delta=5$.

In both cases, we compute offline the optimal *policy* σ_t^* that says which speed the processor should use at time t in all its possible states. At runtime, at each time t, the processor chooses the speed that corresponds to its current remaining work w as $s := \sigma_t^*(w)$.

The algorithms to compute the policy σ^* are based on a *Markovian evolution* of the jobs. Without loss of generality, we will assume $r_n = n$. This means that a new job arrives at each each time slot, this job being of size 0 ($c_n = 0$) if no real work actually arrived at time n. Under this point of view, the job stream is given by a family of distributions $(\phi_w(\cdot,\cdot))_{w\in\mathcal{W}}$ that give the size and the deadline of a job arriving at time t: For any $0 \le \gamma \le C$, and for any $0 \le \delta \le \Delta$,

$$\phi_w(\gamma,\delta) = \Pr(c_t = \gamma, d_t = \delta | w_t = w) \tag{6}$$

Once ϕ is given, the transition matrix $P_s(w,w')$ from state w to w' when the speed chosen by the processor is s is:

$$P_s(w,w') = \begin{cases} \phi_w(\gamma,\delta) & \text{if } w' = \mathbb{T}[(w-s)^+] + \gamma H_\delta \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

3.1 Finite Case: Dynamic Programming

We suppose in this Section that the time horizon is finite and equal to T. It means that we only consider a finite number of jobs. The goal here is to minimize the processor energy consumption during the time interval $[0,T]$.

We compute the minimal total expected processor energy consumption (J^*) from 0 to T. This means that we want to solve the following problem.

If the initial state is w_0, then

$$J^*(w_0) = \min_{\sigma} \left(\mathbb{E} \left(\sum_{t=0}^{T} j(\sigma_t(w_t)) \right) \right) \tag{8}$$

where σ is taken over all possible *policies* of the processor: $\sigma(w,t)$ is the speed used at time t if the state is w. The only constraint on $\sigma(w,t)$ is that it must belong to the set of admissible speeds, *i.e.*, $\sigma(w,t) \in \mathcal{S}$, and it must be large enough to execute the remaining work at the next time step: $\sigma(w,t) \geq w(1)$. The set of *admissible speeds* in state w is denoted $\mathcal{A}(w)$ and is therefore defined by:

$$\mathcal{A}(w) = \left\{ s \in \mathcal{S} \text{ s.t. } s \geq w(1) \right\} \tag{9}$$

J^* can be computed using a backward induction. Let $J_t^*(w)$ be the minimal expected energy consumption from time t to T, if the state at time t is w ($w_t = w$).

3.1.1 Algorithm (DP)

We use backward induction (dynamic programming) to recursively evaluate expected consumption. We use the finite Horizon-Policy Evaluation Algorithm from [6] (p. 80). We obtain an *optimal policy*, which corresponds to the processor speed that one must apply in order to minimize the energy consumption (Algorithm 1).

Algorithm 1 Dynamic Programming Algorithm (DP) to compute the optimal speed for each state and each time.

$t \leftarrow T$ % time horizon
for all $w \in \mathcal{W}$ **do** $J_T^*(w) \leftarrow 0$ **end for**
while $t \geq 1$ **do**
  for all $w \in \mathcal{W}$ **do**

$$J_{t-1}^*(w) \leftarrow \min_{s \in \mathcal{A}(w)} \left(j(s) + \sum_{w' \in \mathcal{W}} P_s(w,w') J_t^*(w') \right)$$

$$\sigma_{t-1}^*[w] \leftarrow \arg\min_{s \in \mathcal{A}(w)} \left(j(s) + \sum_{w' \in \mathcal{W}} P_s(w,w') J_t^*(w') \right)$$

  end for
  $t \leftarrow t-1$ % backward computation
end while
return all tables $\sigma_t^*[\cdot] \quad \forall t = 0 \ldots T-1$.

The complexity to compute the optimal policy $\sigma_t^*(w)$ for all possible states and time steps is $O(T|\mathcal{S}|C\Delta Q(C,\Delta))$. The combinatorial explosion of the state space makes is very large when the sizes of the jobs and the maximum deadline are large. Note however that this pre-computation is done *off-line*. At runtime, the processor simply considers the current remaining work w_t at time t and uses the pre-computed speed $\sigma_t^*(w_t)$ to execute the job with the earliest deadline.

3.1.2 Runtime Process (DP-TLU)

At runtime, the processor computes the current state w and simply uses a table look-up (TLU) to obtain its optimal speed $\sigma_t^*[w]$, the speed tables having been computed offline by (DP) (Al-

gorithm 1).

Algorithm 2 Runtime process (DP-TLU) used by the processor to apply the optimal speed

For Each $t = 0 \dots T-1$
  Update w_t using (2)
  Set $s := \sigma_t^*[w_t]$
  Process earliest deadline job(s) at speed s for one time unit.
End

3.2 Infinite Case: Value Iteration

When the time horizon is infinite, the total energy consumption is also infinite whatever the speed policy. Instead of minimizing the total energy consumption, we minimize the *average* expected energy consumption per time-unit, denoted g. We therefore find the optimal policy σ^* that minimizes g. In mathematical terms, we want to solve the following problem. Compute

$$g^* := \min_{\sigma} \mathbb{E}\left(\lim_{T\to\infty} \frac{1}{T}\sum_{t=1}^{T} j(\sigma(w_t))\right) \tag{10}$$

under the constraint that no job misses its deadline.

3.2.1 Stationary assumptions and existence of the limit

In the following we will make the following additional assumption on the jobs: The size and the deadline of the next job have *stationary distributions* (*i.e.*, they do not depend on time). We further assume that the probability that the next job is of size 0 (meaning that no work arrives in the next time slot) is strictly positive.

Under these two assumptions, the state space transition matrix is *unichain* (see [6] for a precise definition). Basically, this means that starting from state $w_0 = (0, \dots, 0)$, it is possible to go back to state w_0, no matter what speed choices have been made and what jobs have occurred. This is possible because, with positive probability, jobs of size 0 can arrive for long enough a time so that all past deadlines have been met and the state goes back to w_0.

When the state space is unichain, the limit in Eq. (10) always exists [6] and can be computed with an arbitrary precision using a *value iteration* algorithm (VI).

3.2.2 VI Algorithm

This algorithm finds a *stationary* policy σ (here σ will not depend on t), which is optimal, and gives an approximation of the gain (average reward value g^*) with an arbitrary precision. In the following value iteration algorithm, the quantity u^n can be seen as the total energy up to iteration n:

In Algorithm 3, the *span* of a vector is the difference between its maximal value and its minimal value: $span(x) = \max_i(x_i) - \min_i(x_i)$. A vector with a span equal to 0 has all its coordinates equal.

Algorithm 3 Value Iteration Algorithm (VI) to compute the optimal speeds in each state and the average energy cost per time unit.

$u^0 \leftarrow (0, 0, \ldots, 0)$, $u^1 \leftarrow (1, 0, \ldots, 0)$
$n \leftarrow 1$
$\varepsilon > 0$ % stopping criterion
while $span(u^n - u^{n-1}) \geq \varepsilon$ **do**
  for all $w \in \mathcal{W}$ **do**
    $$u^{n+1}(w) \leftarrow \min_{s \in \mathcal{A}(w)} \left\{ j(s) + \sum_{w' \in \mathcal{W}} P_s(w, w') u^n(w') \right\}$$
  end for
  $n \leftarrow n + 1$
end while
Choose any $w \in \mathcal{W}$ and let $g^* \leftarrow u^n(w) - u^{n-1}(w)$
for all $w \in \mathcal{W}$ **do**
  $$\sigma^*[w] \in \arg\min_{s \in \mathcal{A}(w)} \left\{ j(s) + \sum_{w' \in \mathcal{W}} P_s(w, w') u^n(w') \right\}$$
end for
return σ^*

Algorithm 3 computes both the optimal average energy consumption per time unit (g^*) with a precision ε as well as an ε-optimal speed to be selected in each state ($\sigma^*[w]$).

The time complexity to compute the optimal policy depends exponentially on the precision ε. The numerical experiments show that convergence occurs reasonably fast (see Section 4).

3.2.3 Runtime Look-Up (VI-TLU)

As for (DP-TLU), at any integer time $t \in \mathbb{N}$, the processor computes its current state w using Eq.(2) and retrieves its optimal speed $s := \sigma^*[w]$ by looking-up in the table σ^*, pre-computed by (VI) (Algorithm 3).

3.3 Feasibility Issues

Since the amount of work that arrives at every time unit is bounded by C and the deadlines are integer numbers larger than 1, a processor executing jobs at constant speed C will never miss a deadline.

Conversely, if $s_{\max} = C$, then any valid on-line algorithm must always set $s(t) = c_t$ at all times t: Choosing a speed smaller than c_t at time t will imply missing a deadline in some future time slot if a sequence of jobs of size C and deadline 1 arrive at times $t+1, t+2, \ldots, t+\Delta$. These jobs will preempt the processor at each time slot $t+1, \ldots, t+\Delta$ if they do not miss their deadline, and some work present at time t will not be finished at time $t+\Delta$, so one job deadline will be missed. Therefore the optimization problem $\mathcal{P}$ makes sense only when $s_{\max} > C$.

Now, if $s_{\max} > C$, one needs to compute a minimal speed $s_{\min}$ that ensures that feasibility will not be jeopardized as soon as $s(t) \geq s_{\min}$ at all times t. The computation of s_{min} is done by using a worst case scenario, similar to the worst case for the analysis of (OA) in [2].

The maximal work deficit accumulated by the processor occurs when Δ jobs of maximal

size C arrive at respective times 0, 1, ..., $\Delta - 1$, all with the same absolute deadline Δ. Then, $s_{\min}$ must satisfy the following equation:

$$(\Delta - 1)s_{\min} + s_{\max} = C\Delta \tag{11}$$

By assuming $s_{\max} = C + a$ for some $a > 0$, Eq. (11) is equivalent to:

$$s_{\min} = \left(C - \frac{a}{\Delta - 1}\right)^+ \tag{12}$$

If we further impose that the processor works at a speed larger than the speed of (OA) (see Proposition 2), then we can refine the previous compitation and set $s_{\min}$ to a lower value.

In the worst case presented above, the (OA) speed at time 0 is $s^{(OA)}(0) = C/\Delta$. At time 1, $s^{(OA)}(1) = C/\Delta + C/(\Delta - 1)$. More generally, for $0 \leq k < \Delta$, we have:

$$\begin{aligned} s^{(OA)}(k) &= \frac{C}{\Delta} + \frac{C}{\Delta - 1} + \cdots + \frac{C}{\Delta - k} \\ &= C(h_\Delta - h_{\Delta-k-1}) \end{aligned} \tag{13}$$

where h_n is the n-th harmonic number: $h_n = \sum_{i=1}^{n} 1/i$ and $h_0 = 0$. The largest speed used in this scenario is therefore $s^{(OA)}(\Delta - 1) = C \cdot h_\Delta$.

If $s_{\max} \geq C \cdot h_\Delta$, then $s_{\max}$ exceeds the largest speed needed to execute all tasks in the worst case. Therefore, $s_{\min}$ can be set to 0 with no risk to feasibility, yielding a speed range of $[0, s_{max}]$.

If $s^{(OA)}(m-1) \leq s_{\max} < s^{(OA)}(m)$, then the work deficit accumulated in the last steps is:

$$\sum_{j=m}^{\Delta-1} s^{(OA)}(j) - (\Delta - 1 - m)s_{\max} \tag{14}$$

This deficit must be compensated by using a minimal speed $s_{\min}$ in the first steps instead of $s^{(OA)}$. If $s_{\min}$ is used during the first k steps, then the surplus of work executed by the processor w.r.t. (OA) is $ks_{\min} - \sum_{i=0}^{k-1} s^{(OA)}(i)$. This surplus must be larger or equal than the deficit due to $s_{\max}$. Replacing $s^{(OA)}(\cdot)$ by its value from Eq. (13) and using the fact that

$$\sum_{j=0}^{\Delta-2} \left(h_\Delta - h_{\Delta-j-1}\right) = \Delta \tag{15}$$

implies

$$\begin{aligned} s_{\min} \geq\ & \frac{C}{k}\left(\Delta - \sum_{j=k}^{m-1}(h_\Delta - h_{\Delta-j-1})\right) \\ & - \frac{(\Delta - 1 - m)\, s_{\max}}{k} \end{aligned} \tag{16}$$

Finding the integer $k^* \in [1, m[$ that minimizes the right-hand side of Eq. (16) gives the smallest value that s_{min} can take to ensure feasibility at all times.

For example, if $C = 60$ and $\Delta = 6$, then the jobs are feasible with a constant speed $s = 60$ (just see this as the scaled unit speed).

If $s_{\max} \geq 60 \cdot h_6 = 147$ (h_6 being the sixth harmonic number), then no constraint is required on the minimal admissible speed: $s_{\min} = 0$.

If the maximal speed is $s_{\max} = 80$, then Eq. (16) says that the minimal speed should satisfy $s_{\min} \geq 143/3 \approx 47.333...$ (with $k^* = 3$). Notice that using the first naive bound given in Eq. (12) yields $s_{min} \geq 60 - 20/5 = 56$, so the speed range over which Algorithm 3 could operate would be dramatically reduced.

In the following, we will always assume that this feasibility condition is satisfied. In other words, the set of admissible speeds in every state w is further restricted to:

$$\mathcal{A}(w) = \{s \in \mathcal{S} \text{ s.t. } s \geq s^{(OA)}(w) \text{ and } s \geq s_{\min}\} \tag{17}$$

3.4 Properties of the Optimal Policy

In this section, we show several structural properties of the optimal policy σ^* that are true for both the finite and infinite case.

3.4.1 Comparison with Optimal Available (OA)

Optimal Available (OA) is an on-line policy that chooses the speed $s^{(OA)}(w_t)$ in state w_t to be the optimal speed at time t, should no further jobs arrive in the system. More precisely, at time t and in state w_t, the (OA) policy uses the speed

$$s^{(OA)}(w_t) = \max_u \frac{w_t(u)}{u} \tag{18}$$

We first show that, under any state $w \in \mathcal{W}$, the optimal speed $\sigma^*(w)$ is always larger than $s^{(OA)}(w)$.

Proposition 2. *In the finite or infinite case, the optimal speed policy satisfies $\sigma^*(w) \geq s^{(OA)}(w)$ for all state $w \in \mathcal{W}$, if the power consumption j is a convex function of the speed.*

Proof. The proof is based on the observation that (OA) uses the optimal speed assuming that no new job will come in the future. Should some job arrive later, then the optimal speed will have to increase. We first prove the result when the set of speeds $\mathcal{S}$ is the whole *real* interval $[0, s_{\max}]$ (continuous speeds).

Two cases must be considered. If $s^{(OA)}(w_t) = \max_u \frac{w_t(u)}{u}$ is reached for $u = 1$ (*i.e.*, $s^{(OA)}(w_t) = w_t(1)$), then $\sigma^*(w) \geq s^{(OA)}(w)$ by definition because the set of admissible speeds $\mathcal{A}(w_t)$ only contains speeds larger than $w_t(1)$ (see Eq. (18)).

If the maximum is reached for $u > 1$, then $\mathcal{A}(w_t)$ may enable the use of speeds below $w_t(1)$.

Between time t and $t + u$, some new job may arrive. Therefore, the optimal policy should satisfy $\sum_{i=0}^{u-1} \sigma^*(w_{t+i}) \geq w_t(u)$.

The convexity of the power function j implies[4] that the terms of the optimal sequence $\sigma^*(w_t), \ldots, \sigma^*(w_{t+u-1})$ must all be above the average value (which is larger than $w_t(u)/u = s^{(OA)}(w_t)$). In particular, for the first term, $\sigma^*(w_t) \geq s^{(OA)}(w_t)$.

[4] Actually, we use the fact that the sum $\sum_{i=0}^{u-1} j(s)$ is Schur-convex when j is convex (see [8]).

Now, if the set of speeds is finite, then the optimal value of $\sigma^*(w_t)$ must be one of the two speeds in $\mathcal{S}$ surrounding $\sigma^*(w_t)$, $s_1 < \sigma^*(w_t) \leq s_2$. If the smallest one s_1 is chosen, this implies that further choices for $\sigma^*(w_{t+i})$ will have to be larger or equal to s_2, to compensate for the work surplus resulting from choosing a speed below $\sigma^*(w_t)$. This implies that it is never sub-optimal to choose s_2 in the first place (by convexity of the j function).

This trajectorial argument is true almost surely, so that the inequality $\sigma^*(w_t) \leq s^{(OA)}(w_t)$ will also hold with the *expected* energy over both a finite or infinite time horizon. □

3.4.2 Monotonicity Properties

Let us consider two sets of jobs $\mathcal{T}_1$ and $\mathcal{T}_2$ for which we want to apply our speed scaling procedure. We wonder which of the two sets uses more energy than the other when optimal speed scaling is used for both.

Of course, since jobs have random features, we cannot compare them directly, but instead we can compare their distributions. We assume in the following that the size and the deadline of the jobs in $\mathcal{T}_1$ (resp. MT_2) follow a distribution ϕ_1 (resp. ϕ_2) independent of the current state w and of the time t.

Definition 2. *Let us define a stochastic order (denoted $\leq_{st}$) between the two sets of jobs $\mathcal{T}_1$ and $\mathcal{T}_2$ as follows. $\mathcal{T}_2 \leq_{st} \mathcal{T}_1$ if the respective distributions ϕ_1 and ϕ_2 are comparable: For any job (c_1, d_1) with distribution ϕ_1 and any job (c_2, d_2) with distribution ϕ_2, we have:*

$$\forall \gamma, \delta, \quad \Pr(c_2 \geq \gamma, d_2 \leq \delta) \leq \Pr(c_1 \geq \gamma, d_1 \leq \delta) \tag{19}$$

Proposition 3. *If $\mathcal{T}_2 \leq_{st} \mathcal{T}_1$, then:*

1. *over a finite time horizon T, the total energy consumption satisfies $J^{(2)} \leq J^{(1)}$ (computed with Eq. eq8);*
2. *in the infinite time horizon case, the average energy consumption per time unit satisfies $g^{(2)} \leq g^{(1)}$ (computed with Eq. (10)).*

Proof. **Case 1:** The definition of $\mathcal{T}_2 \leq_{st} \mathcal{T}_1$ implies that we can couple the set of jobs $\mathcal{T}_1$ with the set of jobs $\mathcal{T}_2$, such that at each time $t \leq T$, $J_t^{(1)} = (t, c_t^1, d_t^1)$ and $J_t^{(2)} = (t, c_t^2, d_t^2)$ with $c_t^2 \leq c_t^1$ and $d_t^2 \geq d_t^1$ (see [9]). In turn, this implies that the optimal sequence of speeds selected for $\mathcal{T}_1$ is admissible for $\mathcal{T}_2$, hence the optimal sequence for $\mathcal{T}_2$ should have a better performance. Since this is true for any set of jobs generated using ϕ_1, it is also true in expectation, hence $J^{(2)} \leq J^{(1)}$.

Case 2: We just use the fact that the optimal sequence for $\mathcal{T}_2$ is better than the optimal sequence for $\mathcal{T}_1$ over any finite horizon T. Letting T go to infinity shows that the average energy cost per time unit will also be better for $\mathcal{T}_2$. □

4 Numerical Experiments

4.1 Application Scenarios

In this part, we come back to the possible scenarios described in the introduction. The numerical experiments are divided in 2 cases: For § 4.3.1 and § 4.3.2, we consider a situation with a single

sporadic task whose jobs are identical. The inter-arrival times of the jobs are estimated to follow a discrete Poisson process[5], while the sizes and deadlines remain constant.

The second set of experiments deals with another type of real-time systems made of several periodic tasks. Each task is characterized by its offset, period, size, and deadline. In addition, jobs may be lost (for example due to sensor failures and/or perturbations on transmission links). These losses are modeled by a loss probability.

All the experiments reported below are based on these two scenarios.

4.2 Implementation Issues

The state space $\mathcal{W}$ has a rather complex structure and is very large. Therefore, the data structure used in the implementations of Algorithms 1 and 3 must be very efficient to sweep the state space as well as to address each particular state when state changes occur. This is done by using a hashing table to retrieve states according to a multi-dimensional *key* that represents the state (the vector $(w(1), w(2) - w(1), \ldots, w(\Delta) - \sum_{k=1}^{\Delta-1} w(k))$) and a recursive procedure based on Eq. (3) to traverse the state space.

The implementation of Algorithms 1 and 3 has been done in R to take advantage of the possibility to manipulate linear algebraic operation easily, and in C when the state space was too large to be efficiently handled in R.

4.3 Experimental Set-up, Finite Case

Our experiments are done in two steps: Firstly we compute the optimal speeds for each possible state. To compute this policy, we use Algorithm 1 (DP) or 3 (VI), and we store in the σ tables the optimal speed for each possible state of the system.

Secondly, to compare different speed policies, we simulate a sequence of jobs (produced by our real-time tasks, see § 4.1) over which we use our (DP-TLU) solution or other solutions (*e.g.*, off-line or (OA)) and we compute the corresponding energy consumption.

In a nutshell, the experiments show that our solution performs very well in practice, almost as well as the optimal *off-line* solution (§ 4.3.1). As for the comparison with (OA), in most of our experiments, (DP-TLU) outperforms (OA) by 5% on average in the sporadic case when job inter-arrival times are iid[6] (§ 4.3.2). In the periodic case, where jobs are more predictable, the gap with (OA) grows to about 50% (§ 4.3.3).

4.3.1 Comparison with the Off-line Solution

To evaluate our on-line algorithm, we compare it with the off-line solution computed on a simulated set of jobs, generated using the distribution described below. We draw the aggregated work done by the processor (the respective speeds are the slopes) in two cases: the optimal off-line solution that only uses speeds in the finite set $\mathcal{S}$, and the (DP-TLU) solution. Figure 4 displays the result of our simulations in the following case.

Example 1. *One sporadic task τ_1 with jobs of size $c = 2$ and deadline $d = 5$. The inter-arrival times are random, modeled by a geometric distribution of parameter $p = 0.6$.*

[5]Such an estimation can be obtained through the measurement of many traces of the real-time system.

[6]Independent and identically distributed random variables.

As for the processor, we consider that it can only use 4 speeds $\mathcal{S} = \{0, 1, 2, 5\}$ and that its energy consumption per time unit follows the function $j(s) = s^3$. Here, the maximal speed is large enough so that feasibility is not an issue: $5 > 2h_5$ (§3.3).

The result over one typical simulation run is displayed in Figure 4.

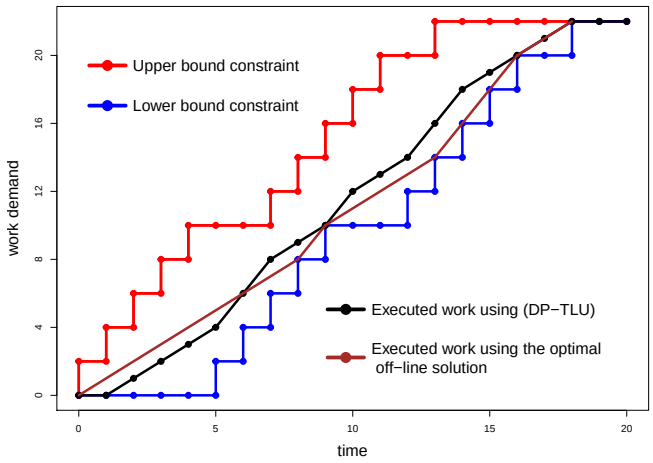


Figure 4: Comparison of the executed work of off-line and (DP) solution on one simulation of Example 1. Red curve: workload arrived between 0 and T; Blue curve: workload deadlines from 0 to T; Brown curve: work executed using the optimal off-line speeds; Black curve: work executed using the speed selection computed by (DP-TLU).

The associate energy consumption curve is:

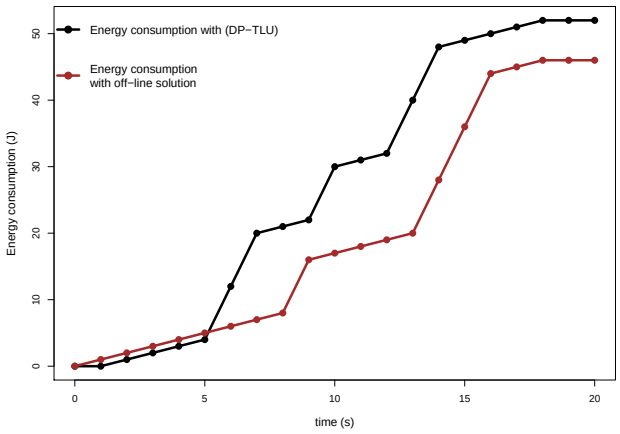


Figure 5: Comparison of the energy consumption of off-line and (DP) solution.

One can notice on Figure 4 that (DP-TLU) consumes more energy than the off-line case, as

expected. However, the off-line and the (DP-TLU) solution only differ in few points: speed 0 is used once by (DP-TLU) but is never used by the off-line solution. Speed 2 is used 5 times by (DP-TLU) and only 4 times in the off-line case. So the energy consumption gap between the two is $2^3 + 0^3 - 1^3 - 1^3 = 6$ J. The total energy consumption under the off-line solution is 46 J, while the total energy consumption under the (DP-TLU) solution is 52 J, so the gap is less than 12 % of the total energy consumption.

4.3.2 Comparisons with (OA), Sporadic Tasks

Recall that, under (OA), the speed at time t under state w_t is set to $\max_u \frac{w_t(u)}{u}$. However, when the number of speeds is finite, (OA) must be adapted: the speed $s^{(OA)}(w_t)$ is thus set to the smallest speed in $\mathcal{S}$ larger than $\max_u \frac{w_t(u)}{u}$.

As a consequence, to compare both solutions, the number of possible speeds must be large enough to get a chance to see a difference between the two. Here, we have simulated two sporadic tasks with the following parameters.

Example 2. *Two sporadic tasks τ_1 and τ_2. τ_1 has job of size $c_1 = 3$, deadline $d_1 = 3$, and arrival rate $p_1 = 0.6$, while τ_2 has jobs of size $c_2 = 6$, deadline $d_2 = 3$, and an arrival rate $p_2 = 0.2$. The processor can use 4 processor speeds $\mathcal{S} = \{0, 1, 2, 3, 4\}$ and its energy consumption per time unit follows the function $j(s) = s^3$.*

We ran an exhaustive experiment made of 10,000 simulations of sequences of jobs over which we computed the relative energy gain of (DP-TLU) over (OA) in percentage. The gain percentage of (DP-TLU) was in the range $[5.17, 5.39]$ with a 95% confidence interval and an average value of 5.28%.

Even if this gain is not very high, one should keep in mind that it comes for free once the (DP) solution has been computed. Indeed, using (DP-TLU) online takes a constant time to select the speed (table look-up) while using (OA) online takes $O(\Delta)$ to compute $\max_u \frac{w_t(u)}{u}$.

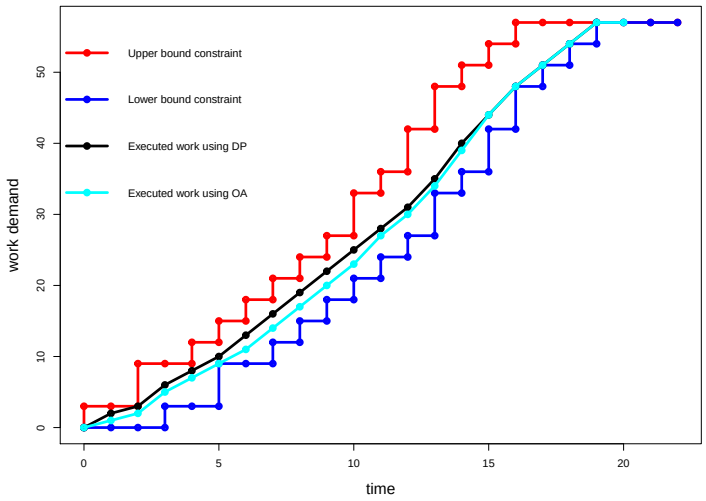


Figure 6: Comparison of the executed work between (OA) and (DP-TLU) solutions, with fixed deadlines $d_n = 3$, size $c_n \in \{0, 2, 4\}$ with respective probabilities $(0.2, 0.6, 0.2)$, and processor speeds in $\{0, 1, 2, 3, 4\}$.

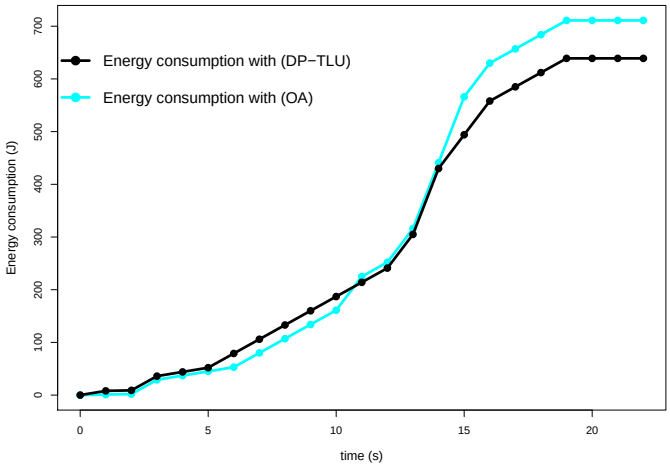


Figure 7: Comparison of the energy consumption between (OA) and (DP-TLU) solutions, with fixed deadlines $d_n = 3$, size $c_n \in \{0, 2, 4\}$ with respective probabilities $(0.2, 0.6, 0.2)$, and processor speeds in $\{0, 1, 2, 3, 4\}$.

Figure 6 shows a comparison between (OA) and (DP-TLU). The total work executed by (DP-TLU) solution is always above the total work executed by (OA), as predicted by Proposition 2. Moreover, we notice on figure 6 that the energy consumed is more important at the beginning

in (DP-TLU) than in (OA), because we anticipate the work to do. The processor executes more work and so consume more energy with (DP-TLU) before time $t = 11$, but after this time, it's the opposite, the energy consumed by (DP-TLU) is lower than the energy consumed by (OA). On all the period, (DP-TLU) outperforms (OA).

Indeed, on this example, the total energy consumption for (OA) is 711 J while that for (DP-TLU) is 639 J. (DP-TLU) outperforms (OA) policy by a margin of around 10 %. Even if this gain is not very high, one should keep in mind that it comes for free once the (DP-TLU) solution has been computed. Indeed, using (DP-TLU) online takes a constant time to select the speed (table look-up) while using (OA) online takes $O(\Delta)$ to compute $\max_u \frac{w_t(u)}{u}$.

4.3.3 Comparisons with (OA), Periodic Tasks

We now consider several examples consisting of two or more periodic tasks.

Example 3. *Two periodic tasks τ_1 and τ_2. For task τ_1, the period is 2, the offset is 0, the job size is $c_1 = 2$, and the deadline is $d_1 = 2$. The loss probability is $p_1 = 0.2$. For task τ_2, the period is 2, the offset is 1, the job size $c_2 = 4$, and the deadline $d_2 = 1$. The loss probability is $p_2 = 0.25$.*

If we do one simulation on 20 time steps, we obtain:

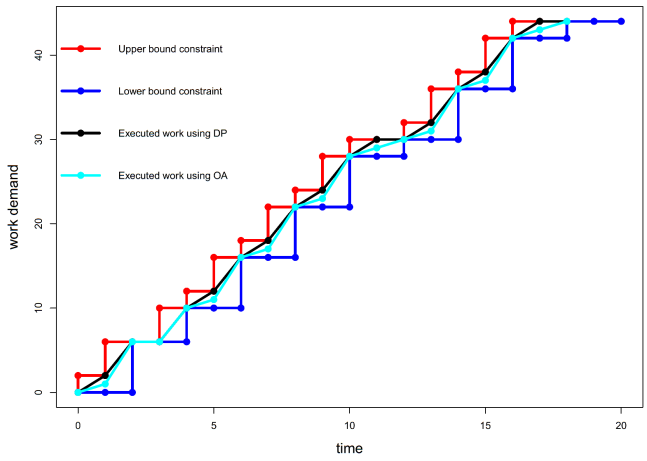


Figure 8: Executed work using (OA) (light blue curve) and (DP-TLU) (black curve) under the two task models.

The energy consumption is as follow:

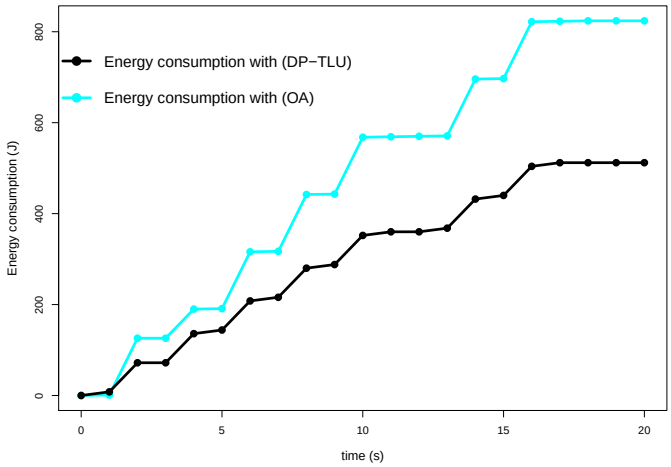


Figure 9: Energy consumption using (OA) (light blue curve) and (DP-TLU) (black curve) under the two task models.

The total energy consumption over the 20 units of time displayed in Figure 11 is 513 J for (DP-TLU), and 825 J for (OA), so more than 60 % higher. Here (DP-TLU) has a clear advantage because the job stream is more predictable than in the independent case.

Example 4. *Four periodic tasks* $\tau_1, \tau_2, \tau_3, \tau_4$ *with the same period equal to* 4 *and respective offsets* $0, 1, 2, 3$. *The loss probability is the same for all tasks* $p = 0.2$. *The other parameters are as follows:* $(c_1 = 1, d_1 = 3)$, $(c_2 = 4, d_2 = 2)$, $(c_3 = 4, d_3 = 1)$, *and* $(c_4 = 2, d_4 = 2)$.

The energy consumed by (DP) is on average 30% lower than the energy consumed by (OA). We performed 10.000 simulations over 40 time steps: the average gain is 29.30% with the following confidence interval at 95%: [29.10, 29.50]. On the graphic representation below, we report one example run only over the first 20 time step. Notice that between time steps 2 and 4, the blue curve (OA) is below the black curve (DP), meaning that (OA) first uses a speed below that of (DP), but thanks to the convexity property of the energy consumption, the energy consumed by (DP) is lower.

Example 5. *Seven periodic tasks* τ_1 *to* τ_7. *Task* τ_1 *has period* 4, *offset* 0, *and* $(c_1 = 4, d_1 = 2, p_1 = 0.2)$. *All the other tasks* $\tau_2, \ldots, \tau_7$ *have period* 8, *loss probability* $p = 0.2$, *respective offsets* $1, 2, 3, 5, 6, 7$ *(4 being for the second job of* τ_1*), and respective parameters* $(c_2 = 1, d_2 = 2)$, $(c_3 = 4, d_3 = 1)$, $(c_4 = 2, d_4 = 2)$ $(c_5 = 4, d_5 = 3)$ $(c_6 = 2, d_6 = 1)$ $(c_7 = 1, d_7 = 2)$.

The energy consumed by (DP) is on average 30% lower than the energy consumed by (OA). We performed 10.000 simulations over 80 time steps, the average gain is 29.98% with the following confidence interval at 95%: [29.87, 30.10]. On the graphic representation below, we report one example run only over the first 20 time step.

The other simulation parameters for Examples 3 to 5 are $T = 20$, $\mathcal{S} = \{0, 1, 2, 3, 4, 5\}$ and $j(s) = s^3$.

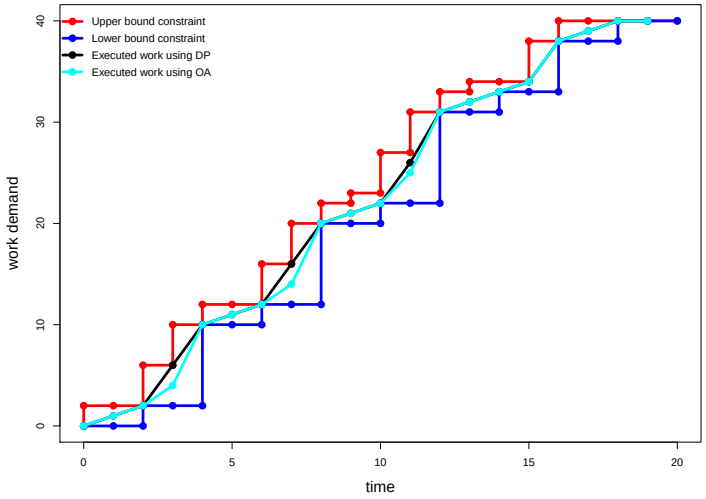


Figure 10: Executed work using (OA) (light blue curve) and (DP-TLU) (black curve) under the four task models.

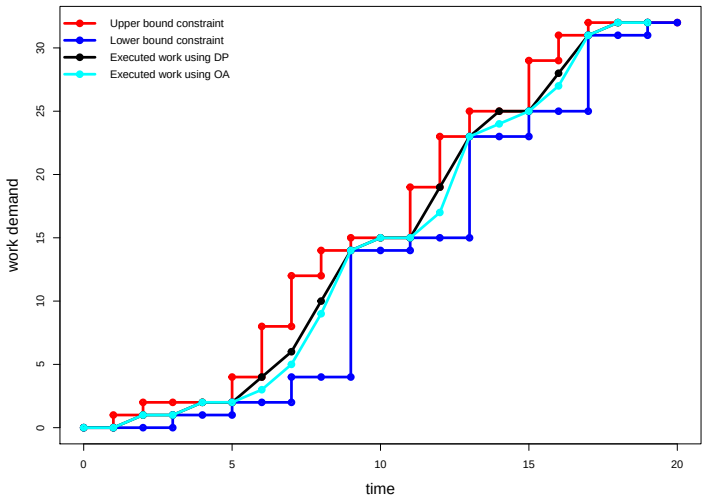


Figure 11: Executed work using (OA) (light blue curve) and (DP-TLU) (black curve) under the seven task models.

As we say before, to analyze the performance of our algorithm, we compare (OA) with (DP-TLU) in all cases described above using $10,000$ simulations for each case. We get the following results:

example	gain against (OA)	95% confidence interval
Ex. 3 (2 tasks)	49.27%	[49.04,49.50]
Ex. 4 (4 tasks)	29.30%	[29.10,29.50]
Ex. 5 (7 tasks)	29.98%	[29.87,30.10]

Table 1: Comparisons between (OA) and (DP-TLU).

In all these cases, (DP-TLU) outperforms (OA) by a larger margin than with sporadic tasks. Indeed, the job sequence is more predictable, so the statistical knowledge over which (DP-TLU) is based is more useful here than in the sporadic case.

We notice some gain differences between the different examples. A partial explanation comes from the following considerations. If there is no error (*i.e.*, no missing jobs), the gain of (DP-TLU) versus (OA) can be computed exactly, as we can notice on figure 12. It is 75% for Ex. 3 (2 tasks), 28% for Ex. 4 (4 tasks) and 52% for Ex. 5 (7 tasks). These computations explain partly the gain differences obtained in the experimental results in Table 1 with 20% losses. Furthermore, in all examples, the system has an hyper-period (respectively $2, 4, 8$). With the 7 tasks, errors accumulate over 8 time units; with the 2 tasks, the error rate is larger. Therefore, the job sequence in one hyperperiod is less predictable in both examples than with 4 tasks. This is why the gain with 7 tasks and with 2 tasks is not as well predicted by the computation made without errors, as with 4 tasks.

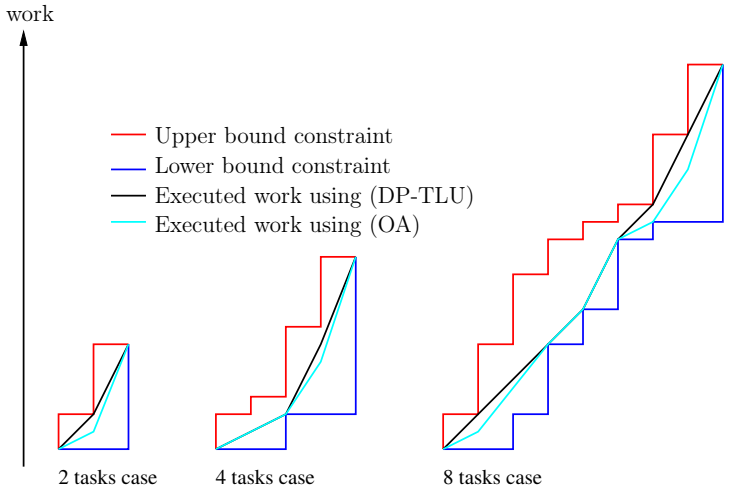


Figure 12: Executed work using (OA) (light blue curve) and (DP-TLU) (black curve) on example 4, 5 and 6 on one period in the case where no jobs are missing

4.4 Computation Experiments, Infinite Case

In this part, we run algorithm VI (algorithm 3) to compute the optimal speed to be used at each time step over an infinite horizon. We fix the stopping criterion (used in Algorithm 3) to

$\varepsilon = 1.0 * 10^{-5}$, so our computation of the average consumption is precise by at least 5 digits. We ran the program in the following cases:

Example 6. *One sporadic task with jobs of size* 2*, deadline* 3*, and arrival rate* p *that varies from* 0 *to* 1*.*

Example 7. *One sporadic task with jobs of size* 2*, deadline* 5*, and arrival rate* p *that varies from* 0 *to* 1*.*

In both examples, the available processor speeds are in $\mathcal{S} = \{0, 1, 2\}$ and the energy consumption function is $j(s) = s^2$. The only difference between Examples 6 and 7 are the deadlines.

The results of our computations are displayed in Figure 13. The three curves depict respectively the average energy consumption per time unit as a function of the arrival rate p (which varies from 0 to 1) for Examples 6 and 7, together with a theoretical lower bound.

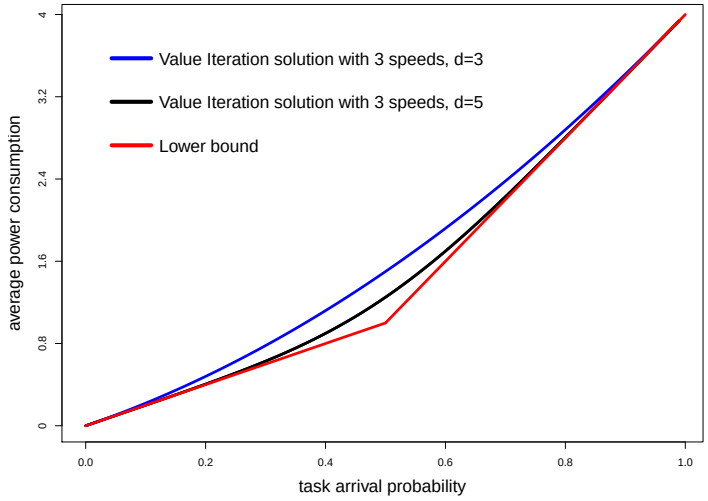


Figure 13: Average energy consumption per time unit for (VI-TLU) and lower bound (red curve), with deadlines equal to 3 (black curve) and 5 (blue curve).

The different curves in Figure 13 have the following meaning:

- The black and blue curves depict the Value Iteration solution with three processor speeds $\mathcal{S} = \{0, 1, 2\}$. These curves display g^*, as computed by Algorithm 3 (VI), as a function of p, the probability that a job of size $c = 2$ and deadline $d = 3$ (black curve) or deadline $d = 5$ (blue curve) arrives in the next instant.
- The red curve is a lower bound on g^*, oblivious of the jobs distribution and deadlines, only based on the average amount of work arriving at each time slot.

As it was expected according to Proposition 3, we notice that the higher the arrival rate, the higher the average energy consumption (both curves are increasing).

Proposition 3 also implies that larger deadlines improve the energy consumption. This is in accordance with the fact that the black curve (deadline 5) is below the blue curve (deadline 3).

What is more surprising here is how well our solution behaves when the deadline is 5. Its performance is almost indistinguishable from the theoretical lower bound (valid for all deadlines) over a large range of the rate p. More precisely, the gap between our solution with deadline equal to 5 and the lower bound is less than 10^{-3} for a rate p between 0 and 0.20, and between 0.80 and 1.

4.4.1 Lower bound

The lower bound has been obtained by solving the optimization problem without taking into account the distribution of the jobs features nor the constraint on the deadlines. Without constraints, and since the power is a convex function of the speed (here $j(s) = s^2$), the best choice is to keep the speed constant. The ideal constant speed needed to execute the jobs over a finite interval $[0, T]$ is $A(T)/T$, where $A(T)$ is the workload arrived before T. When T goes to infinity, the quantity $A(T)/T$ converges to $2p$ by the strong law of large numbers. Therefore, the optimal constant speed is $s^\infty = 2p$.

Now, if we consider the fact that only 3 processor speeds, namely $\{0, 1, 2\}$, are available, then the ideal constant speed $s^\infty = 2p$ cannot be used. In this case the computation of the lower bound is based on the following construction.

If $0 \leq p \leq \frac{1}{2}$, the the ideal constant processor speed $s^\infty = 2p$ belongs to the interval $[0, 1]$. In that case, only speeds $\{0, 1\}$ will be used. To obtain an average speed equal to $2p$, the processor must use speed 1 during a fraction $2p$ of the time and speed 0 the rest of the time. The corresponding average energy per time unit has therefore the following form:

$$g^\infty = 2p \times 1^2 + (1 - 2p) \times 0^2 = 2p \tag{20}$$

If $p \geq \frac{1}{2}$, then $s^\infty = 2p$ belongs to the interval $[1, 2]$. In that case, the processor only uses speeds 1 or 2. To get an average speed of $2p$, it must use the speed 2 during a fraction $2p - 1$ of the time and speed 1 the rest of the time. The corresponding average energy per time unit in this case is:

$$g^\infty = (2p - 1) \times 2^2 + (2 - 2p) \times 1^2 = 6p - 2 \tag{21}$$

As a result, the lower bound on g is given by: $g^\infty = 2p$ if $p \leq 1/2$ and $g^\infty = 6p - 2$ if $p \geq 1/2$. This is the red curve in Figure 13.

5 Conclusion and Perspectives

In this paper, we showed how to select online speeds to execute real-time jobs while minimizing the energy consumption by taking into account statistic information on job features. This information may be collected by using past experiments or simulations as well as deductions from the structure of job sources. Our solution provides performances that are close to the optimal off-line solutions on average, and outperforms classical on-line solutions in cases where the job features have distributions with large variances.

While the goal of this study is to propose a better choice of the speed of the processor in practice, several points are still open and will be the topic of future investigations.

The first one is about the scheduling model: In this paper we assume that jobs are executed under the *Earliest Deadline First* policy, but this is not always the case in practice. What would be the consequence of using another scheduling policy?

The second one is about the time and space complexity of our algorithms that are exponential in the deadlines of the jobs. Although these algorithms are used off-line and can be run on powerful computers, our approach remains limited to a small range of parameters. One potential solution is to simplify the state space and to aim for a sub-optimal solution (but with proven guarantees), using approximate dynamic programming.

Finally, the statistical information gathered on the job features is crucial. When this information is not accurate or even not available, Markov Decision approaches are not possible and one should use learning techniques (such as Q-learning [10]) to construct a statistical model of the jobs on-line and select speeds accordingly, which will converge to optimal speeds over time.

Appendix

We develop in this appendix several extensions to make our model more realistic.This part is based on the assumption that the processor can change speeds at any time. This assumption is not very strong, because there is no technical reason to change processor speed only at task arrival. These generalizations are the following:

1. Convexification of all power consumption functions: Non convex power functions can be advantageously replaced by their convex hull.
2. Time lag between two changes of speeds: This modification can be replaced by a additionnal cost.
3. switching time from the execution of one task to another: This switching time will be also included in the cost function.

A Convexification of the power consumption function

Our general approach does not make any assumption on the power function $j(\cdot)$. Our algorithms (DP) and (VI) will compute the optimal speed selection for any function $j(\cdot)$. However structural properties (comparison with (OA), monotonicity) need convexity assumptions. It is therefore desirable to convexify the power function.

Let us consider a processor, whose speeds belong to the set $\mathcal{S} = \{s_0, s_1, s_2, s_{max}\}$ and a function $j(.) : \mathcal{S} \to \mathbb{R}$, the power function of the processor, represented by the black curve on figure 14. The function $g(.)$ is the convex hull of $j(.)$ (red curve on figure 14).

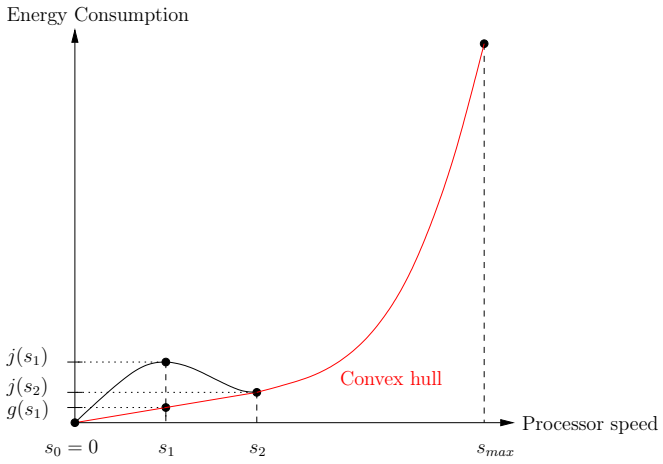


Figure 14: Convexity of energy consumption function

If the energy function is not convex, some speeds are not relevant, because using these speeds is more expensive in term of energy than using a combination of more relevant speeds. On figure 14, we note that it's better to choose speeds s_0 and s_2, rather than speed s_1 during a time step. All points of the power function curve, which are above the convex hull, will never be taken

into consideration. In fact, it is always better to only slect the speeds whose power consumption belongs to the convex hull of the power function. Indeed if $g(s_1) < j(s_1)$ (see Figure 14), instead of selecting speed s_1 during any time interval $[t, t+1)$, the processor can select speed s_2 during a fraction of time α_2 and then speed s_0 during a fraction of time α_0, such that $\alpha_0 s_0 + \alpha_2 s_2 = s_1$. The total quantity of work executed during the time interval $[t, t+1)$ will be the same as with s_1, but the energy consumption will be $g(s_1) = \alpha_0 j(s_0) + \alpha_2 j(s_2) < j(s_1)$.

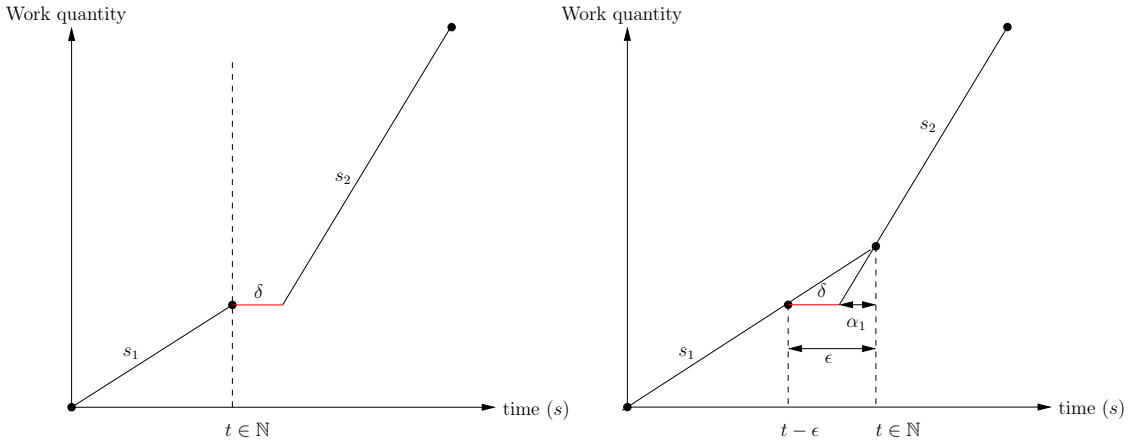
So we can always consider that the power function is convex. This is interesting in practice. Indeed, the actual power consumption of a CMOS circuit working at speed s is not convex $j(s) = Cs^\alpha + L(s)$, where the constant C depends on the activation of the logical gates, α is between 2 and 3 and $L(s)$ is the leakage, with $L(0) = 0$ and $L(s) \not= 0$ if $s > 0$. In this case, convexification removes small speeds from the set of useful speeds.

Remark: This idea to replace one speed with a combination of two speeds can also to used to simulate any speed between 0 and s_{max}. Indeed if a speed doesn't exist in the set $\mathcal{S}$, a solution is to simulate it by combining two neighboring speeds. Suppose that, in figure 14, speed s_1 does belong to $\mathcal{S}$. One can use s_0 and s_2 in $\mathcal{S}$ instead and the energy consumed by this new speed s_1 becomes $g(s_1) = \alpha_1 j(s_0) + \alpha_2 j(s_2)$. This technique is usually called VDD hopping. For our problem, this technique let the processor have more speeds to choose from so that the optimal speed computed by the (DP) algorithm will use less energy with VDD hopping than without it.

B Cost of Speed Changes

In our initial model, one assumption is that the time needed by the processor to change speeds is null. However in all synchronous CMOS circuits, changing speeds takes some time (needed to resynchronize all logical gates). We denote by δ, the time needed by the processor to change speeds. During this time the circuit logical functions are altered so no computation can be done.

Let us consider the evolution on the work executed by the processor around a speed change, occuring at time t. We denote by s_1 the speed used before t and s_2 the speed used after this time.



(a) Situation of speed change cost

(b) Energy extra-cost solution

Figure 15: Cost of speed change (case $s_1 < s_2$)

The figure 15a represents this evolution. The speed changes at time $t \in \mathbb{N}$. Before this time, the processor uses speed s_1. From time t to $t + \delta$ the processor chnages speed and no work is done. After this time delay of δ the processor effectively runs at speed s_2.

The figure 15b represents our solution to transform this time lag into an energy cost, in the case where $s_2 > s_1$. The case $s_2 < s_1$ is similar. Instead of initiating the speed change at time t, we let the processor trigger the speed change at time $t - \epsilon$[7] (where ϵ is defined by the Figure 15). This implies that the new speed s_2 starts being used at time $t - \alpha_1$ (again, α_1 is defined in the Figure).

As illustrated in the Figure 15, under this time shift on the start of the speed change, the work executed by the processor behaves as if the speed change had been instantaneuous and had occured at time t. The only difference is in the energy consumption, that suffers an extra term, computed below.

The value of ϵ and the energy cost of this speed change (the extra-cost denoted $h(s_1, s_2)$) are computed as follows.

- Case $s_2 > s_1$ (as in Figure 15):

$$\begin{aligned} & s_1\epsilon = s_2\alpha_1 \\ \Leftrightarrow\ & (s_2 - s_1)\epsilon = \delta s_2 \end{aligned}$$

  We also have:

$$\epsilon = \frac{\delta s_2}{s_2 - s_1} \tag{22}$$

  Associated total cost on this interval:

$$\begin{aligned} h(s_1, s_2) &= \alpha_1 j(s_2) - \epsilon j(s_1) \\ &= \epsilon(j(s_2) - j(s_1)) - \delta j(s_2) \end{aligned}$$

  The extra-cost is :

$$h(s_1, s_2) = \delta \left[\frac{s_1 j(s_2) - s_2 j(s_1)}{s_2 - s_1} \right] \tag{23}$$

- Case $s_2 < s_1$: The extra-cost is:

$$h(s_1, s_2) = \delta \left[\frac{s_2 j(s_1) - s_1 j(s_2)}{s_1 - s_2} \right] \tag{24}$$

This extra-cost due to speed changes will be taken in consideration in our model in the cost function. To take into account this change, we have to modify the state space $\mathcal{W}$, and add the cuurent speed on the state at $t - 1$. Therefore a new state will have the following form:

$$(w_t, s_{t-1})$$

The new main step of the (DP) algorithm 1 becomes:

$$J^*_{t-1}(w, s) \leftarrow \min_{s \in \mathcal{A}(w)} \left(j(s) + h(s, s') + \sum_{w' \in \mathcal{W}} P_s(w, w') J^*_t(w', s') \right) \tag{25}$$

with $h(s, s') = 0$ when $s = s'$ and otherwise equation 24 if $s' < s$ and equation 23 if $s < s'$.

The rest of the analysis is unchanged with respect to the case of instantaneous speed changes.

[7] This solution supposes that the processor is ϵ-claivoyant about the future task that arrives at time t, to select speed s_2.

C Cost of Context Switches

Here, we further consider that switching from the execution of one task to another one takes some non-negligible time (this is essentially the time needed to upload/download the content of the execution stack). During this context switch no useful work is being executed. In Figure 16, we consider 2 jobs with following characteristics: $\mathcal{J}_1(r_1 = 0, c_1 = 3, d_1 = 7, 5)$ and $\mathcal{J}_2(r_2 = 3, c_2 = 1, 5, d_2 = 3)$. We denote by γ the delay due to a context switch.

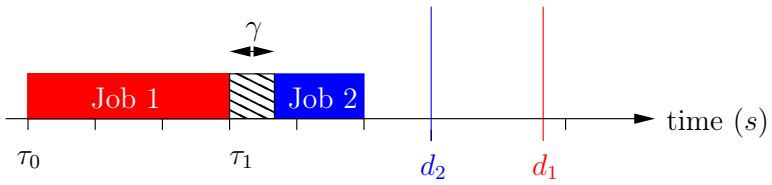


Figure 16: Cost of context switch in time

We notice that at each time the processor starts to execute a new job, it takes some time for switching for one job context to another (context switch). This switching time is denoted by γ (see the barred area in figure 16). We assume that each context switch takes γ, whether this corresponds to the beginning of a new job or the resume of a job after preemption.

The following figure represents the impact of context switches on our model:

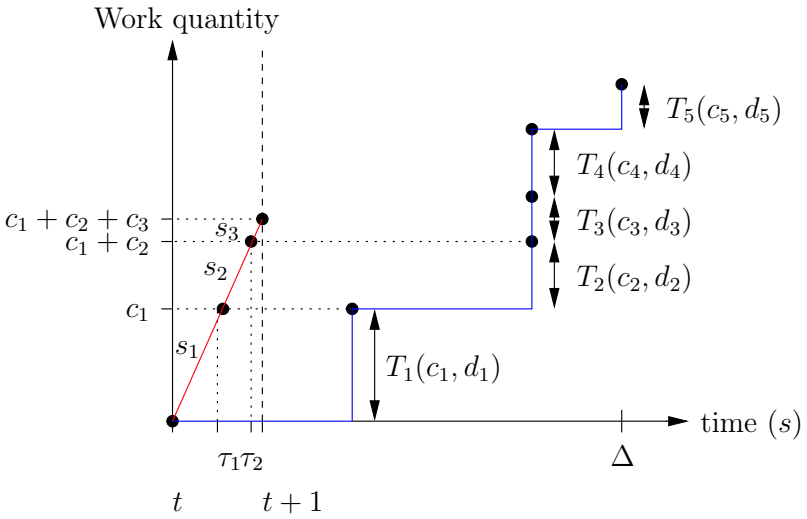


Figure 17: Cost of context switch in energy

One can see on Figure 17, that during one time interval, several switching can occur. In this example in one time step the processor completes two jobs: $\mathcal{J}_1$ and $\mathcal{J}_2$ andstarts the execution of $\mathcal{J}_3$. Times τ_1 and τ_2 are context switches, which leads to a total time delay of 2γ. As in the previous part, we will transform a delay into an energy cost: In one time unit, the evolution of the executed work under speed s, with K context switches is the same as the evolution of the executed work under speed $s(1 - K\gamma)$, with no switching delay.

The state space of the system must be modified in this situation to be able to compute K, the number of context switches in each time interval. We must keep in memory the sizes of the jobs and not only the global remaining work. Indeed, with the current state space $\mathcal{W}$, we don't know for one $w(i), i \in \{0, \ldots, \Delta\}$, the number of different jobs composing it, so we cannot know the number of context switches. We denote by $\overline{\mathcal{W}}$ the new state space.

Now, a state $\overline{w} \in \overline{\mathcal{W}}$ has the following form:

$$\overline{w} = \left[(c_1^1, \cdots, c_1^{k_1}), (c_2^1, \cdots, c_2^{k_2}), \cdots, (c_\Delta^1, \cdots, c_\Delta^{k_\Delta})\right] \tag{26}$$

where c_i^j is the work quantity of one job and k_i is the number of jobs, whose relative deadline is i time units away. Let (r_n, c_n, d_n) be a job that arrives at time $t = r_n$. If the processor speed at time $t-1$ is s_{t-1}, then at time t the next $\overline{w}$ becomes:

$$\begin{aligned}\overline{w}(t) \quad = \Big[&\left((c_2^1(t-1) - l(1,1))^+, \cdots, (c_2^{k_1}(t-1) - l(1,k-1))^+\right), \cdots, \\ &\left((c_{d_n}^1(t-1) - l(d_n,1))^+, \cdots, (c_{d_n}^{k_{d_n}}(t-1) - l(d_n,k_{d_n}))^+, c_n\right), \cdots, \\ &\left((c_\Delta^1(t-1) - l(\Delta,1))^+, \cdots, (c_\Delta^{k_\Delta}(t-1) - l(\Delta,k_\Delta))^+\right), \\ &\left((c_\Delta^1(t-1) - l(\Delta,1))^+, \cdots, (c_\Delta^{k_\Delta}(t-1) - l(\Delta,k_\Delta))^+\right)\Big]\end{aligned}$$

where $l(d,k) = \left(s_{t-1} - \sum_{i=1}^{d-1}\sum_{j=1}^{k_i} c_i^j - \sum_{j=1}^{k} c_d^j\right)^+$.

The idea of the state change is such as we modify all the c_i^j to 0 when $s < \sum_{i,j} c_i^j$, except for the job, which will be executed only partially, where we have a work rest.

The new main step of the (DP) algorithm 1 has the following form:

$$J_{t-1}^*(\overline{w}) \leftarrow \min_{s \in \mathcal{A}(\overline{w})} \left(j\left(\frac{s}{1 - K_s\gamma}\right)(1 - K_s\gamma) + \sum_{\overline{w}' \in \overline{\mathcal{W}}} P_s(\overline{w}, \overline{w}')J_t^*(\overline{w}') \right) \tag{27}$$

We note that the speed $\frac{s}{1-K_s\gamma}$ may not exist, but thanks remark of Appendix A, we can easily simulate this speed.

We denote by K_s the number of job executed if we use the speed s:

$$K_s = \sum_{i=1}^{\alpha-1} k_\alpha + \alpha$$

with α, which is such as $\min_{\alpha,k_\beta}\left(s < \sum_{i,j}^{\alpha,k_\beta} c_i^j\right)$.

The rest of the analysis is unchanged.

As a final remark, one can see that if the processor can do processor sharing, which is often the case nowadays, the state space can be simplified: One only needs to keep in memory the number of jobs, which are executed in a specific $w(i)$, instead of all their sizes, so a state $\overline{w}$ is of the following form:

$$\overline{w} = [(w(1), k_1), (w(2), k_2), \cdots, (w(\Delta), k_\Delta)] \tag{28}$$

with k_i the number of jobs, whose have the same relative deadline of value i. We have also a modification of change state function. Let (r_n, c_n, d_n) be a job that arrives at time $t = r_n$. If the processor speed at time $t-1$ is s_{t-1}, then at time t the next $\overline{w}$ becomes:

$$w_t(\cdot) = \mathbb{T}\big[(w_{t-1}(\cdot) - s_{t-1})^+\big] + c_n H_{d_n}(\cdot)$$

and the number k that corresponds to the deadline d_n of the new task increases by one:

$$k_{d_n} = k_{d_n} + 1.$$

In the processor sharing case, the penalty cost due to the time of processor sharing is proportional to the number of executed jobs. If we denote by μ the total swiching time between two jobs over one time unit, the Bellman equation is the same as (27) replacing K_s by

$$K_s = \sum_i \frac{k_i}{s}(s - \sum_{j=1}^{i-1} w_j) + k_1 \frac{w_1}{s}$$

References

[1] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proceedings of lEEE Annual Foundations of Computer Science*, pp. 374–382, 1995.

[2] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *Journal of the ACM*, vol. 54, no. 1, 2007.

[3] F. Gruian, "On energy reduction in hard real-time systems containing tasks with stochastic execution times," in *IEEE Workshop on Power Management for Real-Time and Embedded Systems*, pp. 11–16, 2001.

[4] J. Lorch and A. Smith, "Improving dynamic voltage scaling algorithms with pace," in *ACM SIGMETRICS 2001 Conference*, pp. 50–61, 2001.

[5] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 89–102, Oct. 2001.

[6] M. L. Puterman, *Markov Decision Process : Discrete Stochastic Dynamic Programming*. Wiley, wiley series in probability and statistics ed., February 2005.

[7] B. Gaujal, N. Navet, and C. Walsh, "Shortest Path Algorithms for Real-Time Scheduling of FIFO tasks with Minimal Energy Use," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, no. 4, 2005.

[8] A. Marshall and I. Olkin, *Inequalitites: Theory of Majorization and Its Applications*, vol. 143 of *Mathematics in Science and Engineering*. Academic Press, 1979.

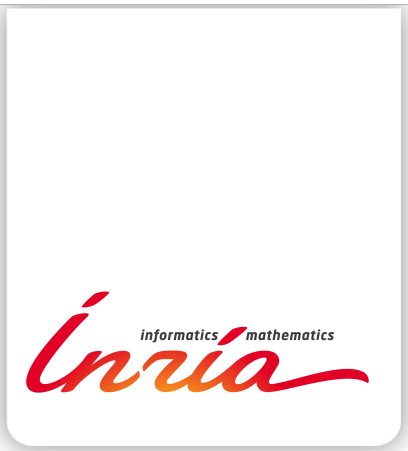
[9] A. Müller and D. Stoyan, *Comparison Methods for Stochastic Models and Risks*. No. ISBN: 978-0-471-49446-1 in Wiley Series in Probability and Statistics, Wiley, 2002.

[10] D. Bertsekas and J. Tsitsiklis, *Neuro-dynamic programming*. Belmont, Mass., USA: Athena Scientific, 1996.

[11] H. Yun and J. Kim, "On energy-optimal voltage scheduling for fixed priority hard real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 2, no. 3, pp. 393–430, 2003.

[12] M. Li and F. F. Yao, "An efficient algorithm for computing optimal discrete voltage schedules," *SIAM J. Comput.*, vol. 35, pp. 658–671, 2005.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399