View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

A Pseudo-Linear Time Algorithm for the Optimal Discrete Speed Minimizing Energy Consumption

Bruno Gaujal, Alain Girault, Stéphan Plassart

To cite this version:

Bruno Gaujal, Alain Girault, Stéphan Plassart. A Pseudo-Linear Time Algorithm for the Optimal Discrete Speed Minimizing Energy Consumption. Discrete Event Dynamic Systems, Springer Verlag, inPress, 10.1007/s10626-020-00327-9 . hal-03030416v2

HAL Id: hal-03030416

https://hal.archives-ouvertes.fr/hal-03030416v2

Submitted on 30 Nov 2020

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.

Noname manuscript No.
(will be inserted by the editor)

A Pseudo-Linear Time Algorithm for the Optimal Discrete Speed Minimizing Energy Consumption

Bruno Gaujal · Alain Girault · Stéphan Plassart

Received: date / Accepted: date

Abstract We consider the classical problem of minimizing off-line the total energy consumption required to execute a set of n real-time jobs on a single processor with a finite number of available speeds. Each real-time job is defined by its release time, size, and deadline (all bounded integers). The goal is to find a processor speed schedule, such that no job misses its deadline and the energy consumption is minimal. We propose a *pseudo-linear time* algorithm that checks the schedulability of the given set of n jobs and computes an optimal speed schedule. The time complexity of our algorithm is in $\mathcal{O}(n)$, to be compared with $\mathcal{O}(n \log(n))$ for the best known solution. Besides the complexity gain, the main interest of our algorithm is that it is based on a completely different idea: instead of computing the *critical intervals*, it sweeps the set of jobs and uses a *dynamic programming* approach to compute an optimal speed schedule. Our linear time algorithm is still valid (with some changes) when arbitrary (non-convex) power functions and when switching costs are taken into account.

Keywords Real time systems; Dynamic Voltage and Frequency Scaling; Energy minimization; Dynamic programming.

1 Introduction

Among numerous hardware and software techniques used to reduce energy consumption of a processor, supply voltage reduction, and hence reduction of

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'Avenir.

Bruno Gaujal · Alain Girault · Stéphan Plassart
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
E-mail: bruno.gaujal@inria.fr E-mail: alain.girault@inria.fr E-mail: stephan.plassart@inria.fr

CPU speed, is particularly effective. This is because the energy consumption of the processor is a function at least quadratic in the speed of the processor in most models of CMOS circuits. Nowadays, variable voltage processors are readily available and a lot of research has been conducted in the field of Dynamic Voltage and Frequency Scaling (DVFS). Under real-time constraints, the extent to which the system can reduce the CPU frequency (or processor speed in the following) depends on the jobs' features (execution time, arrival date, deadline) and on the underlying scheduling policy. Several algorithms have been proposed in the literature to adapt processor speed using DVFS.

There are two classes of such problems, *on-line* and *off-line*. In the off-line case, all the jobs are known a priori with their characteristics, and they are in a finite number, while in the on-line case the characteristics of the job are "discovered" when they are released, and the number of jobs can be infinite. We focus in this paper on the off-line case. Our goal is to minimize the energy consumption under the constraint that no job misses its deadline. Checking this constraint is known as checking the *feasibility* of the set of real-time jobs.

The problem of computing off-line DVFS schedules to minimize the energy consumption has been well studied in the literature, starting from the seminal paper of Yao et al. in 1995 [16]. All the previous algorithms proposed in the literature compute the *critical interval* of the set of jobs[1], using more and more refined techniques to do so. This started in 1995 with [16] and [13] where it was independently shown that one can compute the optimal speed schedule with complexity $\mathcal{O}(n^3)$, where n is the number of real-time jobs to schedule[2]. Later, [5] showed in 2007 that the complexity can be reduced to $\mathcal{O}(n^2 L)$, where L is the nesting level of the set of jobs. Finally the complexity has been reduced to $\mathcal{O}(n^2)$ in the most recent work in 2017 [9].

When the number of available speeds is *finite*, equal to m, [10] gave in 2007 a $\mathcal{O}(n^2)$ algorithm, while [8] proposed in 2005 a $\mathcal{O}(mn \log n)$ algorithm. In their most recent work, the same authors showed in 2017 that the complexity can be further reduced to $\mathcal{O}(n \log(\max\{m, n\}))$ [9].

In this paper, we present a *dynamic programming* solution that sweeps the set of jobs and computes the best speed at each time step while checking feasibility, even when the power function is not convex. The complexity is equal to Kn, where the constant K depends linearly on the maximal speed and quadratically on the maximal relative deadline of the jobs. Therefore, our solution is competitive with the solution in [9] when the number of jobs is large and all deadlines are bounded. More details about this comparison are given in Section4.4. Our solution is inspired from a Markov Decision Process approach proposed by Gaujal et al. in [3] in the on-line case when statistical data on the job characteristics (arrival time, WCET, and deadline) are known. Their

[1] The critical interval is the time interval with the highest load per time unit, to be precisely defined later.

[2] The arithmetic complexity of an algorithm is the number of elementary operations it requires, regardless of the size of their arguments.

algorithm computes the optimal on-line speed scaling policy that minimizes the expected energy consumption. Here, we show that their algorithm can be adapted to the off-line case where the characteristics of the jobs are given as inputs to the algorithm. It does not use the *critical intervals* on which in all previous approaches are based and therefore, it is still valid under more realistic cases with non-convex power functions and when switching costs are taken into account while approaches based on critical intervals collapse in these cases.

We introduce in Section 2 the system model. Then we detail in Section 3 the state space. Our dynamic programming solution is detailed in Section 4. We then study extensions of our algorithm. We show in Section 5 that the dynamic programming approach can also be used in the case where switching from one speed to another is not free, but instead takes some time and some extra energy, which is a more realistic model. Finally we provide concluding remarks in Section 6.

This paper is a long and improved version of [4]: We have removed the assumption that the power function is convex, and the condition that speeds must be consecutive. This version also contains a new algorithm with a greatly improved complexity and the new Theorem 4 for switching costs.

2 System Model

We consider a set of n jobs $\{J_i\}_{i=1..n}$ to be executed by a single core processor equipped with dynamic voltage and frequency scaling (DVFS). Each job J_i is defined by the triplet (τ_i, c_i, d_i), where τ_i is the *inter-arrival time* between J_i and J_{i-1} (with $\tau_1 = 1$ by convention), c_i is the *size* (also called its WCET), and d_i the *relative deadline* bounded by Δ. From the inter-arrival times and the relative deadlines we can reconstruct the *release times* r_i and the *absolute deadlines* D_i of the job J_i as follows:

$$r_i = \sum_{k=1}^{i} \tau_k \quad \forall i \geq 1, \tag{1}$$

$$D_i = r_i + d_i. \tag{2}$$

Since specifying a set of jobs as $\{(\tau_i, c_i, d_i)\}_{i=1..n}$ or as $\{(r_i, c_i, D_i)\}_{i=1..n}$ is equivalent, we use both notations in our examples.

We assume that all these quantities are in $\mathbb{N}$. We also denote by T the last deadline among all jobs, called the *time horizon* of our system. It is defined as:

$$T = \max_{i=1}^{n}\{D_i\}. \tag{3}$$

The single core processor is equipped with m processing speeds also assumed to be in $\mathbb{N}$, and $s_{\max}$ denotes the maximal speed. The set of available

speeds is denoted $\mathcal{S}$. The speeds are not necessarily consecutive integers. In the first part of the paper, we assume that the cost of switching speeds is null. This will be generalized in Section 5 to include speed switching costs using the same idea as in [3].

In this paper, all jobs are scheduled by the *Earliest Deadline First* (EDF) preemptive scheduling policy. A key advantage of EDF is that it is optimal for *feasibility*. A set of jobs is *feasible* if and only if there exists a job schedule so that no deadline is missed when the processor always uses its maximal speed $s_{\max}$. In this respect, the optimality of EDF means that, if a set of jobs is feasible, then it is also feasible under EDF.

When the processor runs at speed $s(t)$ it executes an amount of work equal to $\int_t^{t+1} s(u)\mathrm{d}u$ in one time unit, provided that enough jobs are active in the system at this time t (*i.e.*, released and not yet finished). For instance, assume that two jobs J_1 and J_2 are active at time t, with respective size $c_1 = 1$ and $c_2 = 2$, and with absolute deadlines such that $D_2 > D_1 > t$. If the processor runs at the constant speed $s = 2$ during the time slot $[t, t+1]$, then during this time slot it executes entirely J_1 and half of J_2.

The power dissipated at any time t by the processor running at speed $s(t)$ is denoted $Q(s(t))$ (arbitrary, not necessarily convex). According to these notations, the total energy consumption E is:

$$E = \int_1^T Q(s(t))\mathrm{d}t. \tag{4}$$

Given a set of n jobs $\{J_i\}_{i=1..n}$, the goal is to find an *optimal speed schedule* $\{s(t), t \in [1, T]\}$ that will allow the processor to execute all the jobs before their deadlines while minimizing the total energy consumption E.

3 State Space

3.1 State Description

The central idea of this paper is to define the *state* of the system at time t. We denote $\mathcal{W}$ the set of all states of the system.

A natural state of the system at time t is the set of all jobs present at time t, *i.e.*, $\{J_i = (r_i, c_i, d_i) | r_i \leq t \leq r_i + d_i\}$. Yet, in order to compute the speed of the processor, one does not need to know the set of actual jobs but only the *cumulative remaining work* present at time t, corresponding to these jobs. Therefore, a more compact state will be the *remaining work function* $w_t(\cdot)$ at time t: for any $u \in \mathbb{R}^+$, $w_t(u)$ is the amount of work that must be executed before time $u+t$, taking into account all the jobs J_i present at time t (*i.e.*, with a release time $r_i \leq t$ and deadline $r_i + d_i > t$). By definition, the remaining work $w_t(\cdot)$ is a *staircase function*.

To derive a formula for $w_t(\cdot)$, let us introduce the work quantity that arrives at any time t: to achieve this, we define in Def. 1 a new function $a_t(\cdot)$. For any $u \in \mathbb{R}^+$, the quantity $a_t(u)$ is the amount of work that arrives at time t and must be executed before time $t+u$.

Definition 1 The amount of work that arrived at time t and must be executed before time $t+u$ is

$$a_t(u) = \sum_{i \mid r_i = t} c_i H_{r_i+d_i}(t+u), \tag{5}$$

where $H_{d_i}(\cdot)$ is the discontinuous step function defined $\forall x \in \mathbb{R}$ by

$$H_{r_i+d_i}(x) = \begin{cases} 0 & \text{if } x < r_i + d_i, \\ 1 & \text{if } x \geq r_i + d_i. \end{cases}$$

To illustrate the definition of $a_t(\cdot)$, let us consider an example with 3 jobs J_1, J_2, J_3 with respective release times $r_1 = r_2 = r_3 = t$, sizes $c_1 = 1$, $c_2 = 2$, $c_3 = 1$ and relative deadlines $d_1 = 2$, $d_2 = 3$, $d_3 = 5$. In this case, the function $a_t(\cdot)$ is displayed in Figure 1b by the red line.

Def. (1) allows us to describe the state change formula when moving from time $t-1$ to time t, using speed $s(u)$ in the whole interval $[t-1, t]$.

Lemma 1 *At time $t \in \mathbb{N}$ the remaining work function is given by:*

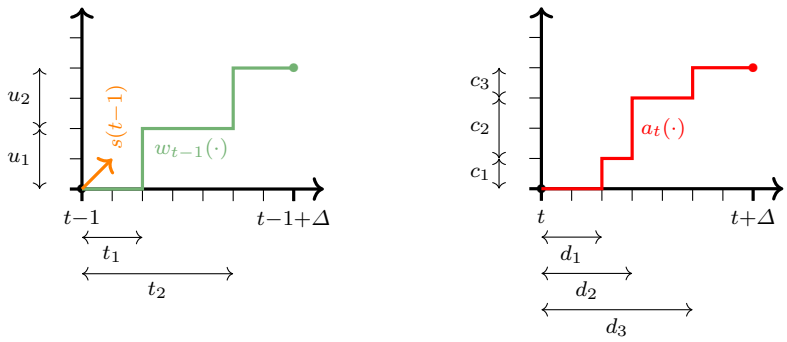
$$w_t(\cdot) = \mathbb{T}\left[\left(w_{t-1}(\cdot) - \int_{t-1}^{t} s(u)\mathrm{d}u\right)^+\right] + a_t(\cdot), \tag{6}$$

with $\mathbb{T}f$ the shift on the time axis of function f, defined as: $\mathbb{T}f(t) = f(t+1)$ for all $t \in \mathbb{R}$, and $f^+ = \max(f, 0)$, the positive part of a function f.

Proof Eq. (6) defines the evolution of the remaining work over time (see Figure 1 for an illustration). The remaining work at time t is the remaining work at $t-1$ minus the amount of work executed by the processor from $t-1$ to t (which is exactly $\int_{t-1}^{t} s(u)\mathrm{d}u$) plus the work arriving at t. The "max" with 0 makes sure that the remaining work is always positive and the $\mathbb{T}$ operation performs a shift of the reference time from $t-1$ to t. □

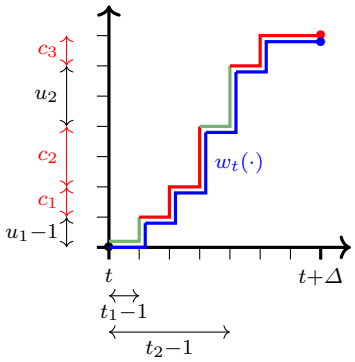
3.2 Size of the State Space

As said in Section 3.1, $\mathcal{W}$ is the set of all states of the system. $\mathcal{W}$ is therefore the set of all possible remaining work functions that can be reached by any *feasible* set of jobs, when the processor only changes its speed at integer times, and when no job has missed its deadline before time t. The size of the state space $\mathcal{W}$ is denoted by G.



(a) State of the system at $t-1$.

(b) Arrival of three new jobs.



(c) Resulting state at t.

Figure 1: **Figure 1a**: State of the system at $t-1$. The green line depicts the remaining work function $w_{t-1}(\cdot)$. The constant speed chosen between times $t-1$ and t is $s(t-1)=1$; u_1 stands for $w_{t-1}(2)$ and u_2 stands for $w_{t-1}(5)-w_{t-1}(2)$. **Figure 1b**: Arrival of three new jobs (r_i, c_i, d_i) at t: $J_1=(t,1,2)$, $J_2=(t,2,3)$, and $J_3=(t,1,5)$. The red line depicts the corresponding arrival work function $a_t(\cdot)$. **Figure 1c**: The blue line depicts the resulting state at t, $w_t(\cdot)$, obtained by shifting the time from $t-1$ to t, by executing 1 unit of work (because $s(t-1)=1$), and by incorporating the jobs arrived at t. Above the blue line are shown in green the "parts" of $w_t(\cdot)$ that come from $w_{t-1}(\cdot)$ and in red those from $a_t(\cdot)$.

Since all jobs have a relative deadline bounded by Δ, then for all $t \in \mathbb{N}$ we have:

$$w_t(u) = w_t(\Delta) \text{ for all } u \geq \Delta. \tag{7}$$

Since $w_t(u)$ is constant after $u=\Delta$, and since w_t is a staircase function with steps at integer times, w_t is completely specified by its first values, $w_t(0), w_t(1), w_t(2), \ldots, w_t(\Delta)$.

The fact that no job has missed its deadline before time t implies $w_t(0) = 0$ (no work with deadline at most t is left at time t).

The number of such remaining work functions depends on the processor characteristics, and in particular of the maximal processor speed $s_{\max}$. Indeed, the processor can execute at most $s_{\max}u$ amount of work during a time interval of size u. As a consequence, in order to take into account only the feasible remaining work functions (*i.e.*, the remaining work functions such that all jobs J_i can be executed before their deadline), one only needs to consider the remaining work functions that satisfy the following equation:

$$\forall 0 < u \leq \Delta,\ w_t(u) \leq s_{\max}u. \tag{8}$$

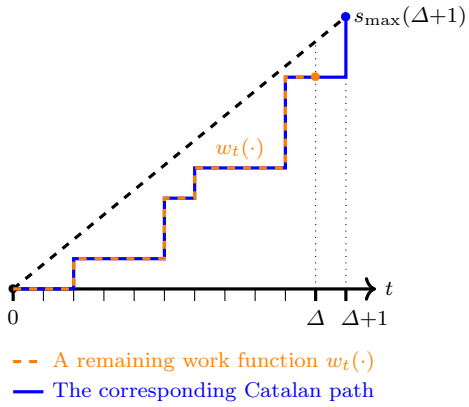


Figure 2: Bijection between remaining work functions $w_t(\cdot)$ (orange dashed staircase line) and the Catalan paths (blue solid staircase line).

Now, the set of remaining work functions $w_t(\cdot)$ that satisfy Eq. (8) can be bijectively associated to the set of increasing paths over the 2D integer lattice[3] that start from point $(0, 0)$, end into point $(\Delta+1, s_{\max}(\Delta+1))$, and that stay below the diagonal. This is done by adding one step to the $w_t(\cdot)$ staircase, from $(\Delta, w_t(\Delta))$ to $(\Delta + 1, s_{\max}(\Delta + 1))$. These paths are known under the name of *Catalan paths* [6]. This bijection is illustrated in Figure 2, where the additional step connects the orange bullet and the blue bullet.

As a consequence, the size G of the state space $\mathcal{W}$ can be computed using a generalization of the Catalan numbers [6]: The number of Catalan paths from $(0, 0)$ to $(\Delta + 1), s_{\max}(\Delta + 1))$, hence the number of all possible remaining work functions for any set of feasible jobs, is:

$$G = \frac{1}{1 + s_{\max}(\Delta + 1)} \binom{(s_{\max} + 1)(\Delta + 1)}{\Delta + 1} \approx \frac{e}{\sqrt{2\pi}} \frac{1}{(\Delta + 1)^{3/2}} (e\, s_{\max})^{\Delta}. \tag{9}$$

[3] Increasing paths over a 2D integer lattice are staircases.

4 Dynamic Programming Solution

4.1 V_{dd}-*hopping*

Any feasible speed schedules $\boldsymbol{s}$ uses speeds in $\mathcal{S}$ at any time $\boldsymbol{s}(t) \in \mathcal{S}$. In a unit interval $[t, t+1]$, the amount of work executed under $\boldsymbol{s}$ is $v_t = \int_t^{t+1} \boldsymbol{s}(u)\mathrm{d}u = \sum_{k=1}^{m} \alpha_k s_k$, where s_k are the speeds in $\mathcal{S}$ and α_k is the time when $s(t) = s_k$ in $[t, t+1]$. By definition, $\sum_k \alpha_k = 1$.

The energy spent by the speed schedule in the interval $[t, t+1]$ is:

$$E_t(\boldsymbol{s}) = \int_t^{t+1} Q(\boldsymbol{s}(u))\mathrm{d}u = \sum_{k=1}^{m} \alpha_k Q(s_k).$$

This speed schedule $\boldsymbol{s}$ can be transformed into a new speed schedule $\boldsymbol{s}'$ such that on each unit interval:

1. The executed work is the same: $v'_t = \int_t^{t+1} \boldsymbol{s}'(u)\mathrm{d}u = v_t$;
2. The energy consumption is smaller: $E_t(\boldsymbol{s}') \leq E_t(\boldsymbol{s})$;
3. The speed schedule $\boldsymbol{s}'$ uses at most two speeds in $[t, t+1]$ (called the *bounding speeds* of v_t in the following, and defined in the next paragraph).

The construction of $\boldsymbol{s}'$ is called V_{dd}-*hopping* in the following. Here is how $\boldsymbol{s}'$ is constructed. The energy consumption of $E_t(\boldsymbol{s}) = \sum_k \alpha_k Q(s_k)$ is a convex combination of $Q(s_1), \ldots Q(s_m)$. Under the constraint that $\sum_k \alpha_k s_k = v_t$, the minimum for $E_t(\boldsymbol{s})$ is reached when only two values $Q(\underline{v}_t)$ and $Q(\overline{v}_t)$ are used with $\underline{v}_t \leq v_t < \overline{v}_t$ and respective durations α' and $1 - \alpha'$ (or only one speed in the particular case where $\underline{v}_t = v_t$, with $\alpha' = 1$).

It follows that, in unit each interval $[t, t+1]$, the speed schedule $\boldsymbol{s}'$ uses speeds $\underline{v}_t$ and $\overline{v}_t$ with respective duration α' and $1 - \alpha'$. The two speeds $\underline{v}_t$ and $\overline{v}_t$ are called the *bounding speeds* of v_t. This is illustrated in Figure 3.

We further define a new power function, defined for all values of $v_t \in [0, s_{\max}]$ as $\hat{Q}(v_t) := \alpha' Q(\underline{v}_t) + (1 - \alpha')Q(\overline{v}_t)$. Note that $\hat{Q}(v_t)$ is the energy spent by the speed schedule $\boldsymbol{s}'$ in $[t, t+1]$. Also note that once v_t is given, the speed schedule $\boldsymbol{s}'$ is uniquely defined (up to a harmless permutation of $\underline{v}_t$ and $\overline{v}_t$).

4.2 Dynamic Program (First Version)

Algorithm 1 computes the optimal speed schedule. Before presenting its pseudo-code, let us provide an informal description of the behavior of the system. Under a given executed work sequence $v(1), v(2), \ldots, v(T-1)$, the state of the system evolves as follows:

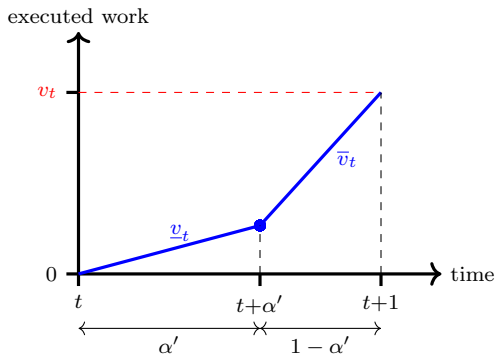


Figure 3: Amount of work executed with speed $\boldsymbol{s}$ (in red), and amount of work executed by $\boldsymbol{s}'$ using the two bounding speeds $\underline{v}_t$ and $\overline{v}_t$ (in blue).

- At time 1, no jobs are present in the system so the initial state function w_1 is the null function, which we represent by the null vector of size Δ: $w_1 = (0, \ldots, 0)$ (see line 4).
- The first job J_1 is released at time 1, maybe simultaneously with other jobs, so the new state function becomes $w_1 = w_0 + a_1$ according to Eq. (6). The case where several jobs are released at time 1 is taken care by the sum operator in Eq. (5) used to compute a_1.
- At time 1, the work executed by the processor is set to $v(1)$. The processor uses this up to time 2, incurring an energy consumption equal to $\hat{Q}(v(1))$.
- At time 2, the state function becomes $w_2 = \mathbb{T}(w_1 - v(1))^+ + a_2$ according to Eqs. (6) and (5), and so on and so forth up to time $T-1$, resulting in the sequence of state functions $w_1, w_2, \ldots, w_{T-1}$.

Now, let us denote by $E_t^*(w)$ the minimal energy consumption from time t to time T, if the state at time t is w, and if the optimal speed schedule is $v^*(t), v^*(t+1), \ldots, v^*(T-1)$. Of course, this partial optimal schedule is not known. But let us assume (using a backward induction) that the optimal speed schedule is actually known *for all possible states* $w \in \mathcal{W}$ at time t. It then becomes possible to compute the optimal speed schedule for all possible states between time $t-1$ and T using the maximum principle:

$$E_{t-1}^*(w) = \min_v \left(\hat{Q}(v) + E_t^*(\mathbb{T}(w-v)^+ + a_t) \right) \tag{10}$$

$$v^*(t-1)(w) = \arg\min_v \left(\hat{Q}(v) + E_t^*(\mathbb{T}(w-v)^+ + a_t) \right), \tag{11}$$

where $v^*(t)(w)$ denotes the optimal execution at time t if the current state is w.

When time 0 is reached, the optimal speed schedule has been computed between 0 and T *for all possible initial states*. To obtain an optimal speed schedule for the sequence of states $w_1, \ldots, w_{T-1}$, we just have to return the

speeds $s^*(1)(w_1), \ldots, v^*(T-1)(w_{T-1})$ (see line 26). Note that, because of the "arg min" operator in Eq. (11), the optimal speed schedule is not necessarily unique.

Algorithm 1 Dynamic program computing the optimal executed work.

1: **input:** $\{J_i = (\tau_i, c_i, d_i), i = 1..n\}$ % Set of jobs to schedule
% Initializations
2: **for all** $i = 1$ to n **do** $r_i \leftarrow \sum_{k=0}^{i} \tau_k$ **end for** % Release times
3: $T \leftarrow \max_i(r_i + d_i)$ % Time horizon
4: $w_1 \leftarrow (0, \ldots, 0)$
5: **for all** $w \in \mathcal{W}$ **do** $E_T^*(w) \leftarrow 0$ **end for** % Energy at the horizon
% Main loop
6: $t \leftarrow T$ % Start at the horizon
7: **while** $t \geq 1$ **do**
8: **for all** $w \in \mathcal{W}$ **do**
9: $E_{t-1}^*(w) \leftarrow +\infty$
10: **for all** $v \in \mathcal{P}(w)$ **do**
11: $w' \leftarrow \mathbb{T}\left[(w-v)^+\right] + a_t$ % Computation of the next state
12: **if** $w' \notin \mathcal{W}$ **then**
13: $E_t^*(w') \leftarrow +\infty$ % The next state is unfeasible
14: **end if**
15: **if** $E_{t-1}^*(w) > \hat{Q}(v) + E_t^*(w')$ **then**
16: $E_{t-1}^*(w) \leftarrow \hat{Q}(v) + E_t^*(w')$ % Update the energy in state w at $t-1$
17: $v^*(t-1)(w) \leftarrow v$ % Update the optimal speed in state w at $t-1$
18: **end if**
19: **end for**
20: **end for**
21: $t \leftarrow t - 1$ % Backward computation
22: **end while**
% Return the result
23: **if** $E_1^*(w_1) = +\infty$ **then**
24: **return "not feasible"**
25: **else**
26: **return** $\{v^*(t)(w_t)\}_{t=1\ldots T}$
27: **end if**

This is what Algorithm 1 does. E^* is computed using the backward induction described previously, which is a special case of the finite horizon optimization algorithm provided in [11].

The cases where the set of jobs is unfeasible are taken into account by setting the energy function $E_t^*(w')$ to infinity if the state w' is unfeasible, that is, if $w' \notin \mathcal{W}$ (see lines 12 and 13) since $\mathcal{W}$ is the set of feasible states by definition.

If $v(t)(w)$ is the work executed by the processor in $[t, t+1]$ in state w, then the deadline constraint on the jobs imposes that $v(t)(w)$ must be large enough to execute the remaining work at the next time step, and cannot exceed the total work present at time t. This means:

$$\forall t, \forall w, \quad w(\Delta) \geq v(t)(w) \geq w(1). \tag{12}$$

This set of *admissible* executions in state w will be denoted by $\mathcal{P}(w)$ and formally defined as :

$$\mathcal{P}(w) = \big\{v \in \mathbb{N} \mid v \leq s_{\max} \text{ and } w(\Delta) \geq v \geq w(1)\big\}. \tag{13}$$

Our first result is Theorem 1, which states that Algorithm 1 computes the optimal speed schedule.

Theorem 1 *If the set of jobs is not feasible, then Algorithm 1 outputs "not feasible". Otherwise it outputs an optimal execution in each unit interval (and hence an optimal speed schedule via V_{dd}-hopping) that minimizes the total energy consumption.*

Proof **Case A: The set of jobs is not feasible.** Then, at some time t, the state w_t will get out of the set of feasible states, for all possible choices of v. Hence its value $E_t^*(w_t)$ will be set to infinity (see line 13) and this will propagate back to time 1 (see line 16). In conclusion, $E_1^*(w_1)$ will be infinite and Algorithm 1 will return "not feasible" (see line 24).

Case B: The set of jobs is feasible. The proof proceeds in two stages. In the first stage we show that there exists an optimal solution that executes an integer amount of work in each unit interval. In the second stage, we show that Algorithm 1 finds an optimal speed schedule among all solutions that execute an integer amount of work in each unit interval.

Case B – first stage.

A solution $\boldsymbol{s}(t)$ is feasible if no job misses its deadline under $\boldsymbol{s}$. This can be translated into linear constraints on the variables v_t, the quantity of work executed in the unit intervals $[t, t+1]$: $v_t = \int_t^{t+1} s(u)\mathrm{d}u$.

For any interval $[t, t']$, the feasibility constraints are

$$\forall 0 < t < t' \leq T, \quad v_t + v_{t+1} + \cdots + v_{t'-1} \geq \sum_{i: r_i \geq t, r_i + d_i \leq t'} c_i.$$

There is also an upper constraint on the total amount of work that can be executed:

$$\forall 0 < t < T, \quad v_1 + \cdots + v_t \leq \sum_{i: r_i \leq t} c_i.$$

Finally, the speed being limited by $s_{\max}$, a final constraint is

$$\forall 0 < t < T, \quad 0 \leq v_t \leq s_{\max}$$

All these constraints form a constraint matrix with the *consecutive-ones property* (*i.e.*, each line is only made of 0s and a consecutive sequence of 1s) so that the constraint matrix is *totally unimodular* (*i.e.*, every square submatrix has determinant -1, 0 or 1).

As for the cost to be minimized, $E(\boldsymbol{s})$, let us consider a feasible solution $\boldsymbol{s}^*$ whose cost is minimal. The cost $E(s^*)$ can also be expressed as a function of the variables v_t^*.

$$\begin{aligned} E(\boldsymbol{s}^*) &= \int_1^T Q(s^*(u))\mathrm{d}u \\ &= \sum_{t=0}^{T-1} \int_t^{t+1} Q(s^*(u))\mathrm{d}u \\ &= \sum_{t=0}^{T-1} F_t(v_t^*), \end{aligned}$$

where $F_t(\cdot)$ is defined as follows. Let $\underline{v}_t^*$ and $\overline{v}_t^*$ be the two bounding speeds of v_t^* in $\mathcal{S}$. By V_{dd}*-hopping*,

$$F_t(x) = \frac{Q(\overline{v}_t^*) - Q(\underline{v}_t^*)}{\overline{v}_t^* - \underline{v}_t^*}(x - \underline{v}_t^*) + Q(\underline{v}_t^*). \tag{14}$$

This is an affine function of x. The minimum of the sum $\sum_{t=0}^{T-1} F_t$ under totally unimodular constraints is reached at an integer point (see [12] for example).

Case B − second stage. In the second stage of the proof, we show that Algorithm 1 finds an optimal speed selection among all solutions that execute an integer amount of work in each unit interval. Together with the first stage, this will end the proof. Proving the optimality of Algorithm 1 is classical in dynamic programming. This is done by a backward induction on the time t. Let us show that $E_t^*(w)$, as computed by the algorithm, is the optimal energy consumption from time t to time T under any possible state w at time t.

Initial step: $t = T$. We set $E_T^*(w) = 0$ for all w. Indeed no jobs are present after time T, so that the state reached at time t must be $w_T = (0, 0, \ldots, 0)$ because no work is left at time T and the value $E_T^*(w_T) = 0$ is therefore correct. No speed has to be chosen at time T.

Induction: Assume that the property is true at time $t + 1$. At time t, Algorithm 1 computes $\forall w \in \mathcal{W}$, $E_t^*(w)$. In particular, if the set of jobs is feasible, then the actual state at time t, namely w_t, must be in $\mathcal{W}$. Therefore, according to lines 15 and 16, we have:

$$E_t^*(w_t) = \min_{v \in \mathcal{P}(w)} \left(\hat{Q}(v) + E_{t+1}^*(\mathbb{T}(w_t - v)^+ + a_t) \right).$$

All possible speeds at time t are tested with their optimal continuation (by induction hypothesis). Therefore, the best choice of speed at t, which minimizes the total energy from t to T, is selected by Algorithm 1.

Finally, when all the speed changes occur at integer times, the total energy consumption computed by Eq. (4) is equal to the value $E_1^*(w_1)$ computed by Algorithm 1. □

Our second result is Theorem 2, which states that the time complexity of Algorithm 1 is linear in the number of jobs n.

Theorem 2 *The time complexity of Algorithm 1 is Kn, where n is the number of jobs and the constant K depends on the maximal speed $s_{\max}$ and the maximal relative deadline Δ.*

Proof The proof proceeds by inspecting Algorithm 1 line by line. The number of operations in line 11 is equal to the number of jobs whose release time is at time t, denoted n_t:

$$n_t = |\{J_i = (r_i, c_i, d_i) \text{ s.t. } r_i = t\}|, \tag{15}$$

and the sum of all n_t is equal to the total number of jobs, n:

$$\sum_{t=1}^{T} n_t = n. \tag{16}$$

Furthermore, the number of operations in line 12 is Δ (to check if $w'(i) \leq i\, s_{\max}$ for $i = 1..\Delta$). Therefore the total number O of arithmetic operations (copies, comparisons, and additions of integers) is:

$$O = \sum_{t=1}^{T} \sum_{w \in \mathcal{W}} \sum_{s \in \mathcal{P}(w)} (n_t + \Delta + K'), \tag{17}$$

where K' is a constant that accounts for all the arithmetic operations between lines 11 and 18 in Algorithm 1.

The size of $\mathcal{P}(w)$ is bounded by $s_{\max}$[4]. Hence O is bounded by a linear function of n and T:

$$O \leq nGs_{\max} + TGs_{\max}(\Delta + K'). \tag{18}$$

We have seen previously that G is bounded by a function of $s_{\max}$ and Δ (see Eq. (9)). Now, $T = \max_{i=1}^{n}(r_i + d_i) = \max_{i=1}^{n}(d_i + \sum_{j=1}^{i} \tau_j)$. If there exists j such that $\tau_j > \Delta$, then there exists an interval of time when the processor must be idle, between the end of the execution of the first $j-1$ jobs and the release time of the jth job. In this case the problem can be split into two: all jobs from 1 to $j-1$ and all jobs from j to n. This means that one can assume with no loss of generality that all inter-arrival times are smaller than Δ, hence $T \leq n\Delta$.

It follows, the total number of arithmetic operations O is bounded:

$$O \leq nK \quad \text{with} \quad K = Gs_{\max}(\Delta^2 + \Delta K' + 1). \tag{19}$$

Finally, by replacing in Eq. (19) G by its value from Eq. (9), we conclude that exists a constant K_0 such that $O \leq n \times K_0 \sqrt{\Delta}(e\, s_{\max})^{\Delta+1}$. □

[4] To be more precise, $|\mathcal{P}(w)|$ is bounded by $|\mathcal{S}|$, and since $\mathcal{S} = \{0, 1, \ldots m-1\}$, we have $|\mathcal{S}| = s_{\max} + 1$.

4.3 Dynamic Program (Second Version)

The main term in Eq. (19) is G, the size of the state space $\mathcal{W}$. The dynamic programming algorithm 1 computes the optimal energy for all states in $\mathcal{W}$ at each time t, regardless of the fact that these states are reachable at time t.

We present in this section an improved algorithm that constructs the set of reachable states *on the fly* at each time step t, resulting in a dramatic reduction of the complexity, from $\mathcal{O}(n \times \sqrt{\Delta}(e\, s_{\max})^{\Delta+1})$ to $\mathcal{O}(n \times (s_{\max} C \Delta^2))$.

First, let us consider the cumulative evolution up to time t. Let $e(t)$ be the work executed up to time t:

$$e(t) = \sum_{i=0}^{t} v_i, \tag{20}$$

where v_i denotes the executed work in $[i, i+1]$. The cumulative executed work $e(t)$ must be smaller than the cumulative work $A(t)$ arrived before time t, and larger than the cumulative deadlines $D(t)$ at t:

$$D(t) \leq e(t) \leq A(t),$$

with

$$A(t) = \sum_{i:r_i \leq t} c_i \qquad \text{and} \qquad D(t) = \sum_{i:D_i \leq t} c_i. \tag{21}$$

At time 0, $A(0) = e(0) = D(0) = 0$ and at time T, $A(T) = e(T) = D(T) = \sum_{i=1}^{n} c_i$.

As discussed earlier, feasibility implies that at the backlog cannot become greater than $s_{\max}\Delta$. Therefore, under a feasible set of jobs, we have $A(t) - D(t) = \sum_{i:r_t \leq t < D_i} c_i \leq s_{\max}\Delta$, hence for any t the number of different values for $e(t)$ is smaller than $s_{\max}\Delta$.

To further refine these bounds on $e(t)$, we define $M(t)$ as the maximal amount of executed work:

$$M(t) = \min\big(A(t), M(t-1) + s_{\max}\big) \quad \text{with} \quad M(0) = 0. \tag{22}$$

At time t, the maximal amount of executed work $M(t)$ can be bounded by $A(t)$ as discussed above, but also by $M(t-1) + s_{\max}$. This means that at any time t we have

$$D(t) \leq e(t) \leq M(t).$$

Second, the state at time t is a function of $e(t)$. If we denote by $w_t^{e(t)}(\cdot)$ the work remaining function at time t when a quantity $e(t)$ of work has been executed up to time t, then, for all $u \geq 0$, we have:

$$w_t^{e(t)}(u) = \left(\sum_{i:r_i < t} c_i H_{r_i + d_i}(t+u) - e(t) \right)^+ + \sum_{i:r_i = t} c_i H_{r_i + d_i}(t+u). \tag{23}$$

In other words, $w_t^{e(t)}(.)$ is a function of $e(t)$. Since there are $s_{\max}\Delta$ different values of $e(t)$, the same holds for $w_t(.)$. As a result, the number of reachable states at time t is smaller than $s_{\max}\Delta$.

Finally, to make the construction of all reachable states more efficient, the dynamic programming should be done in a *forward* mode, instead of backward as it is done in Algorithm 1, because this allows us to construct the state associated to $e(t)$ incrementally and iteratively, by using the states at time $t-1$. The resulting forward algorithm is shown in Algorithm 2.

Theorem 3 *Algorithm 2 computes the optimal speed schedule using less than $n \times K s_{\max}^2 \Delta^3$ operations, where K is a constant.*

Proof Let us decompose the analysis of the complexity step by step:

- To compute the cumulative functions $A(t)$ and $D(t)$ in lines 8 and 9, the complexity is $\sum_{t=1}^{T}(n_t + n_t + K_1) \leq 2n + K_1 T$, where n_t is the number of tasks released at time t and K_1 is a constant.
- In the main temporal loop, from line 20 to 33, there are three parts:
  1. At line 21, the complexity of the energy initialization is bounded by $K_2 s_{\max}\Delta$, where K_2 is a constant.
  2. From line 22 to 33, there are 2 nested loops, one on e', bounded by $s_{\max}\Delta$ and one another on v, bounded by $s_{\max}$. In this part, we use the minimum principle to determine the minimal energy. All these computations are done in constant time except for line 30. Therefore, the time complexity of the rest is bounded by $K_3 s_{\max}^2 \Delta$, where K_3 is a constant.
  3. In line 30, the state associated to e at time t is constructed. It takes at most Δ operations to subtract v and take the positive part. Moreover n_t additions are needed to add the sizes of the new jobs arrived at t. As a result, the time complexity here is bounded by $K_4 s_{\max}^2 \Delta(\Delta + n_t)$t.

Therefore, the whole loop has a complexity $K_5(s_{\max}^2 \Delta T + s_{\max}^2 \Delta^2 T + s_{\max}^2 \Delta n_t)$.

The result output (line 38) uses T operations.

Finally, $T \leq n\Delta$ (see the proof of Theorem 2).

Putting everything together yields a number of elementary operations (copies of an integer, comparisons, additions) bounded by $n \times K s_{\max}^2 \Delta^3$, where K is a constant that does not depend on the problem instance. □

Figure 4 displays all states visited by Algorithm 2 with the set of jobs given at the left of the figure and with the set of available speeds $\{0, 1, 2\}$. The speeds considered in each state for optimizing the energy are shown as black arrows. Note that speed 0 is not considered between times 5 and 6. This is because for any point e at time 5, we have $w_5^e(1) = 1$. This value comes from Job J_3 that arrives at time 5 with a relative deadline of 1. Also note that the

Algorithm 2 Optimized dynamic program computing optimal executed work.

input: $\{J_i = (\tau_i, c_i, d_i), i = 1..n\}$ % Set of jobs to schedule
% Initializations
for all $i = 1$ to n **do** $r_i \leftarrow \sum_{k=0}^{i} \tau_k$ **end for** % Release times
$T \leftarrow \max_i(r_i + d_i)$ % Time horizon
$w_1^0 \leftarrow (0, \ldots, 0)$
$A(0) \leftarrow 0$; $D(0) \leftarrow 0$; $M(0) \leftarrow 0$
for all $t = 1$ to T **do**
  for all i s.t. $r_i = t$ **do**
    $A(t) \leftarrow A(t) + c_i$ % arrivals at t
    $D(t + d_i) \leftarrow D(t + d_i) + c_i$ % Deadlines
  end for
end for
for all $t = 1$ to T **do**
  $A(t) \leftarrow A(t-1) + A(t)$ % Cumulative arrival staircase
  $D(t) \leftarrow D(t-1) + D(t)$ % Cumulative deadline staircase
  $M(t) \leftarrow \min(A(t), M(t-1) + s_{\max})$ % Maximal executed work
  if $A(t) - D(t) > s_{\max}\Delta$ **then**
    return "not feasible"
  end if
end for
% Main loop
for all $t = 1$ to T **do** % Forward computation
  for all $e \in [D(t), M(t)]$ **do** $E_t^*(e) \leftarrow +\infty$ **end for** % Energy at each reachable state
  for all $e' \in [D(t-1), M(t-1)]$ **do**
    for all $v \in [w_{t-1}^{e'}(1), \min(s_{\max}, M(t) - e')]$ **do** % Sweep admissible executions from $t-1$ to t
      $e \leftarrow e' + v$ % Amount of executed work at time t
      if $E_t^*(e) > \hat{Q}(v) + E_{t-1}^*(e')$ **then**
        $E_t^*(e) \leftarrow \hat{Q}(v) + E_{t-1}^*(e')$ % Forward optimality equation
        $v_t^*(e) \leftarrow v$
        $prev_t^*(e) \leftarrow e'$ % Store the optimal solution backwards
      end if
      $w_t^e \leftarrow \mathbb{T}(w_{t-1}^{e'} - v)^+ + a_t$ % Build the state associated to e at t
    end for
  end for
end for
% Return the result
if $E_T^*(e_T) = +\infty$ **then**
  return "not feasible"
else
  for all t from T to 1 **do** % Output the optimal solution backward
    return $v_t^*(e_t^*)$
    $e_{t-1}^* \leftarrow prev_t^*(e_t^*)$
  end for
end if

point $e = 5$ at time 5 (the blue cross in Figure 4) is not visited because it is below $M(5) = 4$.

The two following corollaries are the main result of this paper.

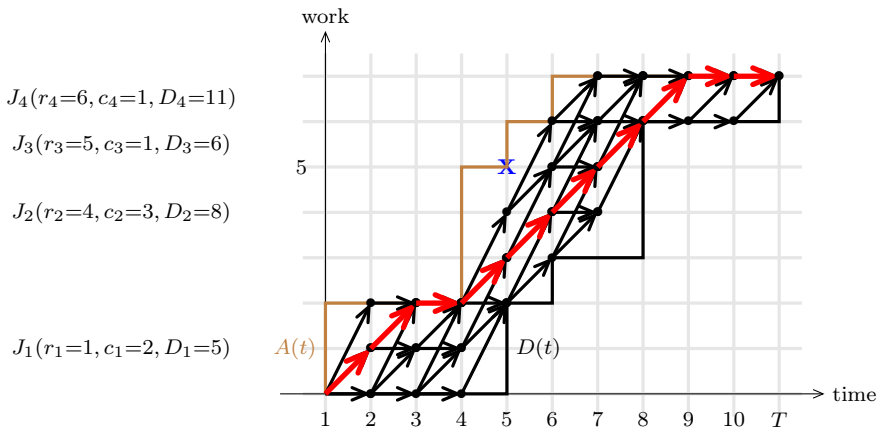


Figure 4: Execution of Algorithm 2 with 4 jobs $\{J_i\}_{1=1..4}$ and a power function $Q(s) = s^2$. The cumulative deadlines form the black staircase $D(t)$, while the cumulative arrivals form the brown staircase $A(t)$. All the states visited by the algorithm are depicted as dots, and all the speeds evaluated in these states are depicted as arrows. The optimal speed schedule computed by Algorithm 2 is shown as the bold red arrows: $(1, 1, 0, 1, 1, 1, 1, 1, 0, 0)$.

Corollary 1 *Algorithm 2 can be improved in order to compute the optimal speed schedule and use less than $n \times K s_{\max}^2 \Delta^2$ operations, where K is a constant.*

Proof The reduction from Δ^3 to Δ^2 can be achieved by replacing the state construction in line 30 in Algorithm 2 by the following code:

Algorithm 3 Modification of line 30 in Algorithm 2.

1: **if** $e \leq M(t-1)$ and $v = 0$ **then**
2: $\quad w_t^e \underset{in-place}{\longleftarrow} \mathbb{T}(w_{t-1}^{e'}) + a_t$
3: **else if** $M(t-1) < e \leq M(t)$ and $v = s_{\max}$ **then**
4: $\quad w_t^e \leftarrow \mathbb{T}(w_{t-1}^{e'} - v)^+ + a_t$
5: **end if**

Indeed, line 2 in the above code changes the vector $w_{t-1}^{e'}$ in place (symbol "$\underset{in-place}{\longleftarrow}$"), *i.e.*, we only move a pointer position for the time shift operation $\mathbb{T}$ (this can be done in constant time) and add a_t with cost $K_1 n_t$. Therefore, this line costs $K_6 n_t$ and will be visited $s_{\max}\Delta$ times at most.

As for line 4 in the above code, the copy of Δ values and the computation of the max cost $K_7\Delta$. However, this line will be visited only $s_{\max}$ times and

not for all states. So the complexity of the state construction is reduced to $K_8 s_{\max}(\Delta + n_t)$.

Therefore the complexity of this replacement of line 30 in Algorithm 2 becomes: $K_9(s_{\max}\Delta n_t + s_{\max}(\Delta + n_t))$.

By adding the other terms computed in the proof of Theorem 3 and the temporal loop, we obtain a complexity of $K_{10}(2n + T + s_{\max}^2 \Delta T + s_{\max}\Delta n + s_{\max}\Delta T + s_{\max}\Delta n)$. □

Corollary 2 *If the work arriving at any instant t is bounded (*i.e., $\forall t, \sum_{i:r_i=t} c_i \leq C$*), then Algorithm 2 computes the optimal speed schedule using less than* $n \times K s_{\max} C \Delta^2$ *operations.*

Proof The complexity change from $s_{\max}^2$ to $s_{\max}C$ comes from the fact that $A(t) - D(t) \leq C\Delta$ if the work arriving at t is bounded by C. □

4.4 Comparison with Previous Work

If we want to compare our algorithm with the best algorithm presented in [9] whose complexity is $K'' n \log(\max\{n, m\})$, obviously, we only gain when the number of jobs n is large and the number of available speeds m small. Also, our constant factor K can be larger than K''.

Under a more detailed inspection, our algorithm is based on the fact that the input is made of $O(n)$ bounded integers, or equivalently, of $O(n)$ rational numbers with bounded numerators and denominators. This can be considered as a valid assumption because elementary operations used in Algorithm 2 (only additions and comparisons between inputs) only take a constant time under this assumption. The analysis of the arithmetic complexity in [9] does not require that the job features are bounded integers. By taking into account the size of the input, the time complexity in [9] will be $K'' n \log(\max\{n, m\}) \log_2(B)$, where B is the maximal input size. Their algorithm is oblivious to the integrity of the input and both algorithms are oblivious to B. Obviously, our algorithm is only competitive over a restricted set of inputs (integer inputs with n large and B small).

We believe that the main contribution of our solution is twofold, on the one hand to show that computing the optimal speed schedule is not necessarily based on the critical interval, and on the other hand to show that this computation can be linear in the number of jobs to be scheduled.

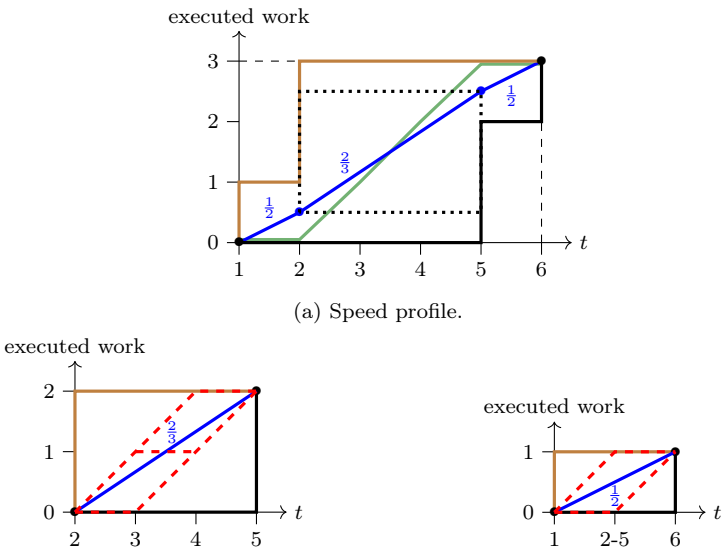
As a side remark reinforcing this fact, there exist instances of jobs where Algorithm 2 cannot be used to find the critical interval. More precisely, by tuning the order in which the speeds are examined in line 23 of Algorithm 2, *all* the optimal speed schedules with integer switching times can be found by Algorithm 2. The following three facts are then true.

- One can find two sets of jobs with different critical intervals and different corresponding critical speeds for which there exists a common optimal speed schedule. This common solution can be the output of Algorithm 2 in both cases with a convenient choice of the order of the speeds in line 23 of Algorithm 2. Consider the following example with a processor having two available speeds: $\{0, 1\}$ and $Q(s) = s^2$. The first set is made of a single job $J_1 = (r_1, c_1, D_1) = (1, 3, 6)$. The second set is made of two jobs $J_2 = (1, 1, 6)$ and $J_3 = (2, 2, 5)$. The critical interval for the set $\{J_1\}$ is $I_1^c = [1, 6]$ with critical speed $s_1^c = 3/5$. In contrast, the critical interval for the set $\{J_2, J_3\}$ is $I_2^c = [2, 5]$ with critical speed $s_1^c = 2/3$. In both cases, if Algorithm 2 sweeps the speeds in increasing order in line 23, its solution is $(0, 0, 1, 1, 1)$.
- For any two sets of jobs with different critical intervals, $I_1^c \subset I_2^c$ (or/and different critical speeds $s_1^c < s_2^c$), there exists an optimal speed schedule for the first set that is not valid for the second set. Informally, this is true because the second set is more constrained and some "extreme" solution for the first set will not satisfy the more stringent constraints of the second set. In the previous example, the schedule $(1, 1, 0, 0, 1)$ is optimal for the set $\{J_1\}$ but it is not valid for the set $\{J_2, J_3\}$ because job J_3 is not completed before its deadline (the processor only executes one unit of work in the time interval $[2, 5]$ while job J_3 is of size 2 on the same interval).
- There exist examples where *some* optimal speed schedules cannot be found by an approach based on critical intervals. For example, using again the set $\{J_2, J_3\}$ with $J_2 = (1, 1, 6)$ and $J_3 = (2, 2, 5)$, the critical interval is $I_3^c = [2, 5]$ with critical speed $s_3^c = 2/3$. Once this critical interval is collapsed and the job J_3 that is included in I_3^c is removed, there remain the two intervals $[1, 2]$ and $[5, 6]$ and the job J_2. As a result, the new critical interval after collapsing becomes $I_4^c = [1, 3]$, with critical speed $s_4^c = 1/2$. In this case, the optimal schedule $(0, 1, 1, 1, 0)$ can be found by Algorithm 2 but will never be discovered by approaches based on the critical interval, because all of them will use speed 0 exactly once in the critical interval I_3^c. This is illustrated in Figure 5.

5 Switching Costs

In this section, we show that Algorithm 2 can be adapted to compute an optimal solution in linear time even when switching from one speed to another has a time and/or energy cost.

So far, we have assumed that the time needed by the processor to change speeds is null. However, in all synchronous CMOS circuits, changing speeds does consume time and energy. The energy cost comes from the voltage regulator when switching the voltage of the circuit, while the time cost comes from the relocking of the Phase-Locked Loop when switching the frequency [15]. Burd and Brodersen have provided in [2] the equations to compute these two



(a) Speed profile.

(b) First critical interval $[2, 5]$.

(c) Second critical interval $[1, 2] \cup [5, 6]$.

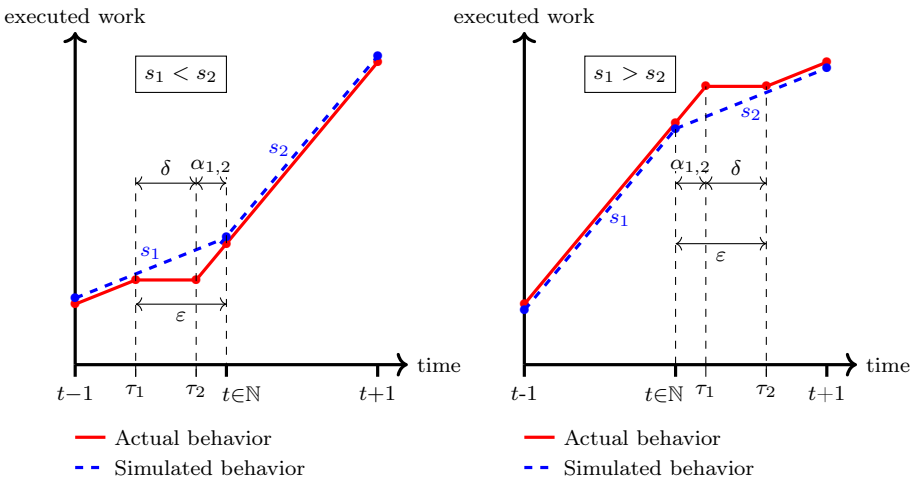
Figure 5: A system made of two jobs $J_2 = (r_2, c_2, D_2) = (1, 1, 6)$ and $J_3 = (2, 2, 5)$ (Figure 5a). The first critical interval is $I_3^c = [2, 5]$ (Figure 5b), which is materialized by the dotted rectangle in Figure 5a. Once I_3^c is collapsed, the second critical interval is $I_4^c = [1, 2] \cup [5, 6]$ (Figure 5c). The critical speed is $s^c = \frac{2}{3}$ on I_3^c and $s^c = \frac{1}{2}$ on I_4^c. All the optimal speed schedules obtained by critical interval methods are depicted as dashed red lines in Figures 5b and 5c. The optimal speed schedule $(0, 1, 1, 1, 0)$ (green curve in Figure 5a) cannot be found by algorithms based on critical intervals, but will be found by Algorithm 2.

costs. In contrast with many DVFS studies (*e.g.*, [2,1,7,14]), our formulation can accommodate energy cost to switch from speed s to s'. In the sequel, we denote this energy cost by $h_e(s, s')$.

As for the time cost, we denote by δ the time needed by the processor to change speeds. For the sake of simplicity we assume that the delay δ is the same for each pair of speeds, but our formalization can accommodate different values of δ, as computed in [2].

5.0.1 Switching Delay as an Energy Cost

When there is a time delay, the executed work by the processor has *two* slope changes, at times τ_1 and τ_2, with $\tau_2 - \tau_1 = \delta$ (the red solid line in Fiure. 6).



(a) Case $s_1 < s_2$. (b) Case $s_1 > s_2$.

Figure 6: Transformation of the time delay into an energy additional cost by shifting the switching point. Figure 6a corresponds to the $s_1 < s_2$ case and Figure 6b to the $s_1 > s_2$ case. The red line represents the actual behavior of the processor with a δ time delay. The blue dashed line represents an equivalent behavior in terms of executed work, with no time delay.

Since $\delta \notin \mathbb{N}$, we cannot have both $\tau_1 \in \mathbb{N}$ and $\tau_2 \in \mathbb{N}$. As a consequence, one of the remaining work functions $w_{\tau_1}(.)$ or $w_{\tau_2}(.)$ will not be integer valued. This is not allowed by our approach.

We propose a solution inspired from [3] and illustrated in Figure 6. It consists in *shifting* the time τ_1 when the speed change is initiated so that the global behavior can be simulated by a single speed change that occurs at an integer time (t in Figure 6). The *actual* behavior of the processor is represented by the red solid line, while the *simulated* behavior, which is equivalent in terms of the amount of work performed, is represented by the blue dashed line. The total amount of work done by the processor is identical in both cases at all integer times $t-1$, t, and $t+1$.

When $s_1 < s_2$, the speed change must be anticipated and occurs at $\tau_1 < t$ (Figure 6a). When $s_1 > s_2$, the speed change has to be delayed and occurs at $\tau_1 > t$ (Figure 6b). The exact computation of t_1 is similar in both cases and is straightforward (see [3]).

One issue remains however, due to the fact that the consumed energy will not be identical between the real behavior and the simulated behavior. Indeed, it will be higher for the actual behavior for convexity reasons. This additional energy cost of the real processor behavior must therefore be added to the energy cost of the equivalent simulated behavior.

The value of ε and $\alpha_{1,2}$ as defined in Figure 6, and the additional energy cost $h_\delta(s_1, s_2)$ incurred by this speed change are computed as follows. In the case $s_2 > s_1$, we have:

$$s_1\varepsilon = s_2\alpha_{1,2} = s_2(\varepsilon - \delta) \Longleftrightarrow \varepsilon = \delta + \alpha_{1,2} = \frac{\delta s_2}{s_2 - s_1}. \tag{24}$$

During the time delay δ, the energy is consumed by the processor as if the speed was s_1. The additional energy cost incurred in the actual behavior (the red solid line) compared with the simulated behavior (the blue dashed line), denoted $h_\delta(s_1, s_2)$, is therefore:

$$h_\delta(s_1, s_2) = \alpha_{1,2}(Q(s_2) - Q(s_1)). \tag{25}$$

Using the value of $\alpha_{1,2}$ from Eq. (24), this yields:

$$h_\delta(s_1, s_2) = \delta s_1 \left(\frac{Q(s_2) - Q(s_1)}{s_2 - s_1} \right). \tag{26}$$

When $s_1 > s_2$, the additional cost becomes:

$$h_\delta(s_1, s_2) = \delta s_2 \left(\frac{Q(s_1) - Q(s_2)}{s_1 - s_2} \right). \tag{27}$$

The global switching cost can now be defined as $h(s', s) = h_e(s', s) + h_\delta(s, s')$, where $h_\delta(s, s')$ is given by Eq. (27) if $s' < s$ and by Eq. (26) if $s < s'$. When $s' = s$, $h(s', s) = 0$.

5.1 Dynamic Program (Third Version)

We now wish to take into account the switching costs in Algorithm 2, which requires to modify the optimality equation in lines 25 – 26.

At time t, recall that e is the executed work at time t, and let s be the speed used just before time t. The optimality equation becomes for all e, s,

$$E_t^*(e, s) = \min_{v \geq w_{t-1}^{e-v}(1)} \min_{s', s'' \in \mathcal{S}} \left(E_{t-1}^*(e - v, s') + h(s', s'') + \alpha Q(s'') + h(s'', s) + (1 - \alpha)Q(s) \right). \tag{28}$$

In Eq. (28), v is the integer quantity of work executed in interval $[t-1, t]$, s'' and s are the two speeds of $\mathcal{S}$ used in interval $[t-1, t]$, in that order, respectively for a duration α and $1 - \alpha$, with $\alpha s'' + (1 - \alpha)s = v$.

Replacing the computation of the minimum in lines 25 – 26 in Algorithm 2 by this new optimality equation (28) changes the complexity to $K s_{\max}^3 \Delta^2 n$. The optimality of the new algorithm is proved in Theorem 4 below.

Theorem 4 *Algorithm 2 with the modification given by Eqs.* (28) *will always compute an optimal speed schedule when switching costs are taken into account if and only if for any* s_1, s_2, s_3 *in* $\mathcal{S}$*,* h *satisfies a triangular inequality:*

$$h(s_1, s_2) + h(s_2, s_3) \geq h(s_1, s_3). \tag{29}$$

Before proving this result, let us comment the Condition (29). Although the triangular inequality of Condition (29) looks natural, the delay cost alone, h_δ defined in Eqs. (26) – (27), does not satisfy it: indeed, for any speeds s_1, s_2, $h_\delta(s_1, 0) + h_\delta(0, s_2) = 0 \leq h_\delta(s_1, s_2)$. Therefore to satisfy the triangular inequality of Condition 29 one must rely on h_e.

Proof (of Theorem 4). We will prove that there always exists an optimal solution that executes an integer amount of work in each interval. This will end the proof because Algorithm 2 modified by Eqs (28) finds the optimal solution among all solutions with integer executed work in each interval.

Let us consider any feasible solution $\boldsymbol{s}$. On each unit interval $[i, i+1]$, $\boldsymbol{s}$ uses a sub-set of the speeds in $\mathcal{S}$, called the *support* of $\boldsymbol{s}$ in $[i, i+1]$.

Part 1: "if"

Let $\boldsymbol{s}^{opt}$ be a solution whose energy is minimal among all solutions that only uses the speeds in the support of $\boldsymbol{s}$ in each interval. Under this subset of speeds, the bounding speeds of $\boldsymbol{s}$ are modified. However, the feasibility constraints on the executed work in each unit interval are not changed. They still have the "consecutive-ones" property. Therefore, the constraints form a totally unimodular matrix (see [12] for example). This implies that there exists a solution minimizing the energy $\boldsymbol{s}^{opt}$ such that, over any interval $[i, i+1]$, the executed work $\int_i^{i+1} \boldsymbol{s}^{opt}(t)\mathrm{d}t = v^{opt}_{[i,i+1]}$ is integer. Furthermore, over $[i, i+1]$, $\boldsymbol{s}^{opt}$ only uses the two (modified) bounding speeds of $v^{opt}_{[i,i+1]}$ in the support of $\boldsymbol{s}$. We can also assume that these bounding speeds are used in the same order as in $\boldsymbol{s}$. Thanks to the triangular inequality of Condition 29, the switching cost of $\boldsymbol{s}^{opt}$ is not higher than the switching cost of $\boldsymbol{s}$. This implies that $\boldsymbol{s}^{opt}$ has a smaller cost (*i.e.*, a smaller energy plus switching cost) than $\boldsymbol{s}$.

Therefore, this new policy $\boldsymbol{s}^{opt}$ has a smaller cost than $\boldsymbol{s}$. Since this is true starting from any feasible policy $\boldsymbol{s}$, this implies that there exists an optimal policy that executes an integer amount of work in each interval.

Part 2: "only if"

We proceed by contradiction. If the triangular inequality is not satisfied (for example, assume that $h(1, 0) + h(0, 2) < h(1, 2)$), then it is easy to check that for the example given in Figure 7, the optimal *integer* solution using at most two speeds in each interval is $\boldsymbol{s}^0 = 1$ over interval $[1, 2]$ and $\boldsymbol{s}^0 = 2$ over interval $[2, 3]$. This speed schedule $\boldsymbol{s}^0$ can be improved by adding speed changes. The schedule $\boldsymbol{s}^\varepsilon$ is defined as follows: $\boldsymbol{s}^\varepsilon = 1$ over $[1, 2 - 2\varepsilon]$, $\boldsymbol{s}^\varepsilon = 0$ over $[2 - 2\varepsilon, 2 - \varepsilon]$, and $\boldsymbol{s}^\varepsilon = 2$ over $[2 - \varepsilon, 3]$. This schedule will have a smaller

cost than $\boldsymbol{s}^0$ provided that ε is small enough, more precisely if:

$$\varepsilon < \frac{h(1,2) - h(1,0) - h(0,2)}{Q(2) + Q(0) - 2Q(1)}.$$

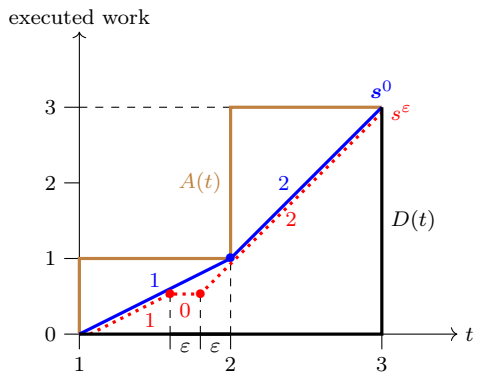


Figure 7: Illustration of the two policies $\boldsymbol{s}^0$ and $\boldsymbol{s}^\varepsilon$. If $h(1,0)+h(0,2) < h(1,2)$ and ε is small enough, then the cost of $\boldsymbol{s}^\varepsilon$ is smaller than the cost of $\boldsymbol{s}^0$.

Notice that in this case there is no optimal policy because when ε goes to 0, the cost of $\boldsymbol{s}^\varepsilon$ decreases to $Q(1) + Q(2) + h(1,0) + h(0,2) - h(1,2)$ and is smaller than the cost for $\varepsilon = 0$, equal to $Q(1) + Q(2)$. □

6 Conclusion

We have addressed the problem of minimizing off-line the total energy consumption required to execute a set of n real-time jobs on a single processor with varying speed. Each real-time job is defined by its release time, size, and deadline (all integers). The goal is to find a sequence of processor speeds, chosen among a finite set of available speeds, such that no job misses its deadline and the energy consumption is minimal. Such a sequence is called an *optimal speed schedule*.

Our main result is that computing an optimal speed schedule can be done with a linear time complexity: Kn where n is the number of real-time jobs and K is a constant. This result holds for an arbitrary power function and can also take into account speed switching costs.

Our solution with an $\mathcal{O}(n)$ complexity is an improvement compared to the previous state of the art, which was $\mathcal{O}(n \log(n))$.

References

1. M. Bandari, R. Simon, and H. Aydin. Energy management of embedded wireless systems through voltage and modulation scaling under probabilistic workloads. In *International Green Computing Conference, IGCC'14*, pages 1–10, Dallas (TX), USA, November 2014. IEEE Computer Society.
2. T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *International Symposium on Low Power Electronics and Design, ISLPED'00*, Rapallo, Italy, July 2000.
3. B. Gaujal, A. Girault, and S. Plassart. Dynamic speed scaling minimizing expected energy consumption for real-time tasks. Technical Report HAL-01615835, Inria, 2017.
4. Bruno Gaujal, Alain Girault, and Stéphan Plassart. A Linear Time Algorithm for Computing Off-line Speed Schedules Minimizing Energy Consumption. In *MSR 2019 - 12ème Colloque sur la Modélisation des Systèmes Réactifs*, pages 1–14, Angers, France, November 2019.
5. Bruno Gaujal and Nicolas Navet. Dynamic voltage scaling under EDF revisited. *Real-Time Systems*, 37(1):77–97, 2007. The original publication is available at www.springerlink.com.
6. Peter Hilton and Jean Pedersen. Catalan numbers, their generalization, and their uses. *The Mathematical Intelligencer*, 13(2):64–75, Mar 1991.
7. K. Li. Energy and time constrained task scheduling on multiprocessor computers with discrete speed levels. *J. of Parallel and Distributed Computing*, 95(C):15–28, September 2016.
8. M. Li and F. F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. *SIAM J. Comput.*, 35:658–671, 2005.
9. Minming Li, Frances F. Yao, and Hao Yuan. An $O(n^2)$ algorithm for computing optimal continuous voltage schedules. In *Annual Conference on Theory and Applications of Models of Computation, TAMC'17*, volume 10185 of *LNCS*, pages 389–400, Bern, Switzerland, April 2017.
10. Jianfeng Mao, Christos G. Cassandras, and Qianchuan Zhao. Optimal dynamic voltage scaling in energy-limited nonpreemptive systems with real-time constraints. *Trans. Mob. Comput.*, 6(6):678–688, 2007.
11. Martin L. Puterman. *Markov Decision Process : Discrete Stochastic Dynamic Programming*. Wiley, series in probability and statistics edition, February 2005.
12. Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.
13. J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publisher, 1998.
14. J. Wang, P. Roop, and A. Girault. Energy and timing aware synchronous programming. In *International Conference on Embedded Software, EMSOFT'16*, Pittsburgh (PA), USA, October 2016. ACM.
15. Q. Wu, P. Juang, M. Martonosi, and D.W. Clark. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *International Conference on High-Performance Computer Architecture, HPCA'05*, pages 178–189, San Francisco (CA), USA, February 2005. IEEE.
16. F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science*, pages 374–382, Milwaukee (WI), USA, October 1995. IEEE Computer Society.