New Results for Non-Preemptive Speed Scaling

Chien-Chung Huang[1] and Sebastian Ott[2]

[1] Chalmers University, Göteborg, Sweden villars@gmail.com
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany ott@mpi-inf.mpg.de

Abstract. We consider the speed scaling problem introduced in the seminal paper of Yao et al. [24]. In this problem, a number of jobs, each with its own processing volume, release time, and deadline, needs to be executed on a speed-scalable processor. The power consumption of this processor is $P(s) = s^\alpha$, where s is the processing speed, and $\alpha > 1$ is a constant. The total energy consumption is power integrated over time, and the objective is to process all jobs while minimizing the energy consumption.
The preemptive version of the problem, along with its many variants, has been extensively studied over the years. However, little is known about the non-preemptive version of the problem, except that it is strongly NP-hard and allows a (large) constant factor approximation [5, 8, 16]. Up until now, the (general) complexity of this problem is unknown. In the present paper, we study an important special case of the problem, where the job intervals form a laminar family, and present a quasipolynomial-time approximation scheme for it, thereby showing that (at least) this special case is not APX-hard, unless NP $\subseteq$ DTIME$(2^{poly(\log n)})$.
The second contribution of this work is a polynomial-time algorithm for the special case of equal-volume jobs. In addition, we show that two other special cases of this problem allow fully polynomial-time approximation schemes.

1 Introduction

Speed scaling is a widely applied technique for energy saving in modern microprocessors. Its general idea is to strategically adjust the processing speed, with the dual goals of finishing the tasks at hand in a timely manner while minimizing the energy consumption. The following theoretical model was introduced by Yao et al. in their seminal paper of 1995 [24]. We are given a set of jobs, each with its own *volume* v_j (number of CPU cycles needed for completion of this job), *release time* r_j (when the job becomes available), and *deadline* d_j (when the job needs to be finished), and a processor with power function $P(s) = s^\alpha$, where s is the processing speed, and $\alpha > 1$ is a constant (typically between two and three for modern microprocessors [13, 23]). The energy consumption is power integrated over time, and the objective is to process all given jobs within their *time windows* $[r_j, d_j)$, while minimizing the total energy consumption.
Most work in the literature focuses on the *preemptive* version of the problem, where the execution of a job may be interrupted and resumed at a later point of

time. For this setting, Yao et al. [24] gave a polynomial-time exact algorithm to compute the optimal schedule. The *non-preemptive* model, where a job must be processed uninterruptedly until its completion, has so far received surprisingly little attention, even though it is often preferred in practice and widely used in current real-life applications. For example, most current real-time operating systems for automotive applications use non-preemptive scheduling as defined by the OSEK/VDX standard [22]. The advantage of this strategy lies in the significant lower overhead (preemption requires to memorize and restore the state of the system and the job) [5], and the avoidance of synchronization efforts for shared resources [22]. From a theoretical point of view, the non-preemptive model is of interest, since it is a natural variation of Yao et al.'s original model. So far, little is known about the complexity of the non-preemptive speed scaling problem. On the negative side, no lower bound is known, except that the problem is strongly NP-hard [5]. On the positive side, Antoniadis and Huang [5] showed that the problem has a constant factor approximation algorithm, although the obtained factor $2^{5\alpha-4}$ is rather large. Recently, Bampis et al. [8] and Cohen-Addad et al. [16] have significantly improved on the constant.

1.1 Our Results and Techniques

In this paper, we work towards better understanding the complexity of the non-preemptive speed scaling problem, by considering several special cases and presenting (near-)optimal algorithms. In the following, we give a summary of our results.

Laminar Instances: An instance is said to be *laminar* if for any two different jobs j_1 and j_2, either $[r_{j_1}, d_{j_1}) \subseteq [r_{j_2}, d_{j_2})$, or $[r_{j_2}, d_{j_2}) \subseteq [r_{j_1}, d_{j_1})$, or $[r_{j_1}, d_{j_1}) \cap [r_{j_2}, d_{j_2}) = \emptyset$. The problem remains strongly NP-hard for this case [5]. We present the first $(1+\epsilon)$-approximation for this problem, with a quasipolynomial running time (i.e. a running time bounded by $2^{poly(\log n)}$ for any fixed $\epsilon > 0$); a so-called quasipolynomial-time approximation scheme (QPTAS). Our result implies that laminar instances are not APX-hard, unless NP $\subseteq$ DTIME($2^{poly(\log n)}$). We remark that laminar instances form an important subclass of instances that not only arise commonly in practice (e.g. when jobs are created by recursive function calls [19]), but are also of theoretical interest, as they highlight the difficulty of the non-preemptive speed scaling problem: Taking instances with an "opposite" structure, namely *agreeable instances* (here for any two jobs j_1 and j_2 with $r_{j_1} < r_{j_2}$, it holds that $d_{j_1} < d_{j_2}$), the problem becomes polynomial-time solvable [5]. On the other hand, further restricting the instances from laminar to *purely-laminar* (see next case) results in a problem that is only weakly NP-hard and admits an FPTAS.

Purely-Laminar Instances: An instance is said to be *purely-laminar* if for any two different jobs j_1 and j_2, either $[r_{j_1}, d_{j_1}) \subseteq [r_{j_2}, d_{j_2})$, or $[r_{j_2}, d_{j_2}) \subseteq [r_{j_1}, d_{j_1})$. We present a fully polynomial-time approximation scheme (FPTAS) for this class of instances. This is the best possible result (unless P = NP), as the problem is still (weakly) NP-hard [5].

Equal-Volume Jobs: If all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$, we present a polynomial-time algorithm for computing an (exact) optimal schedule. We thereby improve upon a recent result of Bampis et al. [7], who proposed a 2^{α}-approximation algorithm, and answer their question for the complexity status of this problem.

Bounded Number of Time Windows: If the total number of different time windows is bounded by a constant, we present an FPTAS for the problem. This result is again optimal (unless P = NP), as the problem remains (weakly) NP-hard even if there are only two different time windows [5].

The basis of all our results is a discretization of the problem, in which we allow the processing of any job to start and end only at a carefully chosen set of *grid points* on the time axis. We then use dynamic programming to solve the discretized problem. For laminar instances, however, even computing the optimal discretized solution is hard. The main technical contribution of our QPTAS is a relaxation that decreases the exponential size of the DP-tableau without adding too much energy cost. For this, we use a kind of overly compressed representation of job sets in the bookkeeping. Roughly speaking, we "lose" a number of jobs in each step of the recursion, but we ensure that these jobs can later be scheduled with only a small increment of energy cost.

1.2 Related Work

The study of dynamic speed scaling problems for reduced energy consumption was initiated by Yao, Demers, and Shenker in 1995. In their seminal paper [24], they presented a polynomial-time algorithm for finding an optimal schedule when preemption of jobs is allowed. Furthermore, they also studied the online version of the problem (again with preemption of jobs allowed), where jobs become known only at their release times, and developed two constant-competitive algorithms called *Average Rate* and *Optimal Available*.

Over the years, a rich spectrum of variations and generalizations of the original model have been investigated, mostly with a focus on the preemptive version. Irani et al. [18], for instance, considered a setting where the processor additionally has a sleep state available. Another extension of the original model is to restrict the set of possible speeds that we may choose from, for example by allowing only a number of discrete speed levels [15, 20], or bounding the maximum possible speed [9, 14, 17]. Variations with respect to the objective function have also been studied, for instance by Albers and Fujiwara [2] and Bansal et al. [11], who tried to minimize a combination of energy consumption and total flow time of the jobs. Finally, the problem has also been studied for arbitrary power functions [10], as well as for multiprocessor settings [1, 3, 12].

In contrast to this diversity of results, the non-preemptive version of the speed scaling problem has been addressed rarely in the literature. Only in 2012, Antoniadis and Huang [5] proved that the problem is strongly NP-hard, and gave a $2^{5\alpha-4}$-approximation algorithm for the general case. Recently, the approximation ratio has been improved to $2^{\alpha-1}(1+\epsilon)\tilde{B}_{\alpha}$, where $\tilde{B}_{\alpha}$ is the α-th generalized

Bell number, by Bampis et al. [8], and to $(12(1+\epsilon))^{\alpha-1}$ by Cohen-Addad et al. [16]. For the special case where all jobs have the same volume, Bampis et al. [7] proposed a 2^{α}-approximation algorithm. Independently of our result for this setting, Angel et al. [4] also gave a polynomial-time exact algorithm for such instances, with the same complexity of $\mathcal{O}(n^{21})$.

Very recently, multi-processor non-preemptive speed scaling also started to draw the attention of researchers. See [7, 16] for details.

1.3 Organization of the Paper

Our paper is organized as follows. In section 2 we give a formal definition of the problem and establish a couple of preliminaries. In section 3 we present a QPTAS for laminar instances, and in section 4 we present a polynomial-time algorithm for instances with equal-volume jobs. Our FPTASs for purely-laminar instances and instances with a bounded number of different time windows can be found in the appendix. Most proofs are also deferred to the appendix.

2 Preliminaries and Notations

The input is given by a set $\mathcal{J}$ of n jobs, each having its own release time r_j, deadline d_j, and volume $v_j > 0$. The power function of the speed-scalable processor is $P(s) = s^{\alpha}$, with $\alpha > 1$, and the energy consumption is power integrated over time. A *schedule* specifies for any point of time (i) which job to process, and (ii) which speed to use. A schedule is called *feasible* if every job is executed entirely within its time window $[r_j, d_j)$, which we will also call the *allowed interval* of job j. Preemption is not allowed, meaning that once a job is started, it must be executed entirely until its completion. Our goal is to find a feasible schedule of minimum total energy consumption.

We use $E(S)$ to denote the total energy consumed by a given schedule S, and $E(S, j)$ to denote the energy used for the processing of job j in schedule S. Furthermore, we use OPT to denote the energy consumption of an optimal schedule. A crucial observation is that, due to the convexity of the power function $P(s) = s^{\alpha}$, it is never beneficial to vary the speed during the execution of a job. This follows from Jensen's Inequality. We can therefore assume that in an optimal schedule, every job is processed using a uniform speed.

In the following, we restate a proposition from [5], which allows us to speed up certain jobs without paying too much additional energy cost.

Proposition 1. *Let S and S' be two feasible schedules that process j using uniform speeds s and $s' > s$, respectively. Then $E(S', j) = (s'/s)^{\alpha-1} \cdot E(S, j)$.*

As mentioned earlier, all our results rely on a discretization of the time axis, in which we focus only on a carefully chosen set of time points. We call these points *grid points* and define *grid point schedules* as follows.

Definition 1 (Grid Point Schedule). *A schedule is called* grid point schedule *if the processing of every job starts and ends at a grid point.*

We use two different sets of grid points, $\mathcal{P}_{\text{approx}}$ and $\mathcal{P}_{\text{exact}}$. The first set, $\mathcal{P}_{\text{approx}}$, is more universal, as it guarantees the existence of a near-optimal grid point schedule for any kind of instances. On the contrary, the set $\mathcal{P}_{\text{exact}}$ is specialized for the case of equal-volume jobs, and on such instances guarantees the existence of a grid point schedule with energy consumption exactly OPT. We now give a detailed description of both sets. For this, let us call a time point t an *event* if $t = r_j$ or $t = d_j$ for some job j, and let $t_1 < t_2 < \ldots < t_p$ be the set of ordered events. We call the interval between two consecutive events t_i and t_{i+1} a *zone*. Furthermore, let $\gamma := 1 + \lceil 1/\epsilon \rceil$, where $\epsilon > 0$ is the error parameter of our approximation schemes.

Definition 2 (Grid Point Set $\mathcal{P}_{\text{approx}}$). *The set $\mathcal{P}_{\text{approx}}$ is obtained in the following way. First, create a grid point at every event. Secondly, for every zone (t_i, t_{i+1}), create $n^2\gamma - 1$ equally spaced grid points that partition the zone into $n^2\gamma$ many subintervals of equal length $L_i = \frac{t_{i+1} - t_i}{n^2\gamma}$. Now $\mathcal{P}_{\text{approx}}$ is simply the union of all created grid points.*

Note that the total number of grid points in $\mathcal{P}_{\text{approx}}$ is at most $\mathcal{O}\big(n^3(1 + \frac{1}{\epsilon})\big)$, as there are $\mathcal{O}\big(n\big)$ zones, for each of which we create $\mathcal{O}\big(n^2\gamma\big)$ grid points.

Lemma 1. *There exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{approx}}$, such that $E(\mathcal{G}) \leq (1 + \epsilon)^{\alpha - 1}\text{OPT}$.*

Definition 3 (Grid Point Set $\mathcal{P}_{\text{exact}}$). *For every pair of events $t_i \leq t_j$, and for every $k \in \{1, \ldots, n\}$, create $k - 1$ equally spaced grid points that partition the interval $[t_i, t_j]$ into k subintervals of equal length. Furthermore, create a grid point at every event. The union of all these grid points defines the set $\mathcal{P}_{\text{exact}}$.*

Clearly, the total number of grid points in $\mathcal{P}_{\text{exact}}$ is $\mathcal{O}\big(n^4\big)$.

Lemma 2. *If all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$, there exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{exact}}$, such that $E(\mathcal{G}) = \text{OPT}$.*

3 Laminar Instances

In this section, we present a QPTAS for laminar problem instances. We start with a small example to motivate our approach, in which we reuse some ideas of Muratore et al. [21] for a different scheduling problem. Consider Figure 1, where we have drawn a number of (laminar) time intervals, purposely arranged in a tree structure. Imagine that for each of those intervals I_k, we are given a set of jobs J_k whose allowed interval is equal to I_k. Furthermore, let us make the simplifying assumption that no job can "cross" the boundary of any interval I_k during its execution. Then, in any feasible schedule, the set of jobs J_1 at the root of the tree decomposes into two subsets; the set of jobs processed in the left child I_2, and the set of jobs processed in the right child I_3. Having a recursive procedure in mind, we can think of the jobs in the root as being split up and handed down to the respective children. Each child then has a set of "inherited"

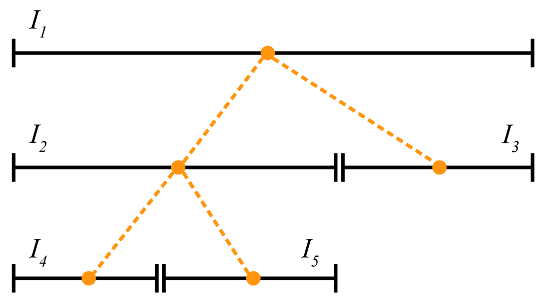

Fig. 1. Time intervals of a laminar instance, arranged in a tree structure.

jobs, plus its own original jobs to process, and both are available throughout its whole interval. Now, the children also split up their jobs, and hand them down to the next level of the tree. This process continues until we finally reach the leaves of the tree, where we can simply execute the given jobs at a uniform speed over the whole interval.
Aiming for a reduced running time, we reverse the described process and instead compute the schedules in a bottom-up manner via dynamic programming, enumerating all possible sets of jobs that a particular node could "inherit" from its ancestors. This dynamic programming approach is the core part of our QPTAS, though it bears two major technical difficulties. The first one is that a job from a father node could also be scheduled "between" its children, starting in the interval of child one, stretching over its boundary, and entering the interval of child two. We overcome this issue by taking care of such jobs separately, and additionally listing the truncated child-intervals in the dynamic programming tableau. The second (and main) difficulty is the huge number of possible job sets that a child node could receive from its parent. Reducing this number requires a controlled "omitting" of small jobs during the recursion, and a condensed representation of job sets in the DP tableau. At any point of time, we ensure that "omitted" jobs only cause a small increment of energy cost when being added to the final schedule. We now elaborate the details, beginning with a rounding of the job volumes. Let $\mathcal{I}$ be the original problem instance.

Definition 4 (Rounded Instance). *The* rounded instance $\mathcal{I}'$ *is obtained by rounding down every job volume* v_j *to the next smaller number of the form* $v_{\min}(1+\epsilon)^i$*, where* $i \in \mathbb{N}_{\geq 0}$ *and* $v_{\min}$ *is the smallest volume of any job in the original instance. The numbers* $v_{\min}(1+\epsilon)^i$ *are called* size classes*, and a job* belongs *to size class* $\mathcal{C}_i$ *if its rounded volume is* $v_{\min}(1+\epsilon)^i$.

Lemma 3. *Every feasible schedule* S' *for* $\mathcal{I}'$ *can be transformed into a feasible schedule* S *for* $\mathcal{I}$ *with* $E(S) \leq (1+\epsilon)^\alpha E(S')$.

From now on, we restrict our attention to the rounded instance $\mathcal{I}'$. Remember that our approach uses the inherent tree structure of the time windows. We proceed by formally defining a tree T that reflects this structure.

Definition 5 (Tree T). *For every interval $[t_i, t_{i+1})$ between two consecutive events t_i and t_{i+1}, we introduce a vertex v. Additionally, we introduce a vertex for every time window $[r_j, d_j)$, $j \in \mathcal{J}$ that is not represented by a vertex yet. If several jobs share the same allowed interval, we add only one single vertex for this interval. The interval corresponding to a vertex v is denoted by I_v. We also associate a (possibly empty) set of jobs J_v with each vertex v, namely the set of jobs j whose allowed interval $[r_j, d_j)$ is equal to I_v. Finally, we specify a distinguished root node r as follows. If there exists a vertex v with $I_v = [r^*, d^*)$, where r^* is the earliest release time and d^* the latest deadline of any job in $\mathcal{J}$, we set $r := v$. Otherwise, we introduce a new vertex r with $I_r := [r^*, d^*)$ and $J_r := \emptyset$. The edges of the tree are defined in the following way. A node u is the son of a node v if and only if $I_u \subset I_v$ and there is no other node w with $I_u \subset I_w \subset I_v$. As a last step, we convert T into a binary tree by repeating the following procedure as long as there exists a vertex v with more than two children: Let v_1 and v_2 be two "neighboring" sons of v, such that $I_{v_1} \cup I_{v_2}$ forms a contiguous interval. Now create a new vertex u with $I_u := I_{v_1} \cup I_{v_2}$ and $J_u := \emptyset$, and make u a new child of v, and the new parent of v_1 and v_2. This procedure eventually results in a binary tree T with $\mathcal{O}(n)$ vertices.*

The main idea of our dynamic program is to stepwise compute schedules for subtrees of T, that is for the jobs associated with the vertices in the subtree (including its root), plus a given set of "inherited" jobs from its ancestors. Enumerating all possible sets of "inherited" jobs, however, would burst the limits of our DP tableau. Instead, we use a condensed representation of those sets via so-called *job vectors*, focusing only on a logarithmic number of size classes and ignoring jobs that are too small to be covered by any of these. To this end, let δ be the smallest integer such that $n/\epsilon \leq (1+\epsilon)^\delta$, and note that δ is $\mathcal{O}(\log n)$ for any fixed $\epsilon > 0$.

Definition 6 (Job Vector). *A* job vector $\overrightarrow{\lambda}$ *is a vector of $\delta + 1$ integers $\lambda_0, \ldots, \lambda_\delta$. The first component λ_0 specifies a size class, namely the largest out of δ size classes from which we want to represent jobs (therefore $\lambda_0 \geq \delta - 1$). The remaining δ components take values between 0 and n each, and define a number of jobs for each of the size classes $\mathcal{C}_{\lambda_0}$, $\mathcal{C}_{\lambda_0 - 1}$, $\ldots$, $\mathcal{C}_{\lambda_0 - \delta + 1}$ in this order. For example, if $\delta = 2$, the job vector $(4, 2, 7)$ defines a set containing 2 jobs with volume $v_{\min}(1+\epsilon)^4$ and 7 jobs with volume $v_{\min}(1+\epsilon)^3$.*
We refer to the set of jobs defined by a job vector $\overrightarrow{\lambda}$ as $J(\overrightarrow{\lambda})$.

Remark: We do not associate a strict mapping from the jobs defined by a job vector $\overrightarrow{\lambda}$ to the real jobs (given as input) they represent. The jobs $J(\overrightarrow{\lambda})$ should rather be seen as dummies that are used to reserve space and can be replaced by any real job of the same volume.

Definition 7 (Heritable Job Vector). *A job vector $\overrightarrow{\lambda} = (\lambda_0, \ldots, \lambda_\delta)$ is* heritable *to a vertex v of T if:*

1. *At least λ_i jobs in $\bigcup_{u \text{ ancestor of } v} J_u$ belong to size class $\mathcal{C}_{\lambda_0 - i + 1}$, for $1 \leq i \leq \delta$.*

2. $\lambda_1 > 0$ *or* $\lambda_0 = \delta - 1$.

The conditions on a heritable job vector ensure that for a fixed vertex v, λ_0 can take only $\mathcal{O}(n)$ different values, as it must specify a size class that really occurs in the rounded instance, or be equal to $\delta - 1$. Therefore, in total, we can have at most $\mathcal{O}(n^{\delta+1})$ different job vectors that are heritable to a fixed vertex of the tree. In order to control the error caused by the laxity of our job set representation, we introduce the concept of δ-omitted schedules.

Definition 8 (δ-omitted Schedule). *Let J be a given set of jobs. A* δ-omitted schedule *for J is a feasible schedule for a subset $R \subseteq J$, s.t. for every job $j \in J \setminus R$, there exists a job* $\text{big}(j) \in R$ *with volume at least $v_j(1+\epsilon)^\delta$ that is scheduled entirely inside the allowed interval of j. The jobs in $J \setminus R$ are called* omitted jobs*, the ones in R* non-omitted jobs.

Lemma 4. *Every δ-omitted schedule S' for a set of jobs J can be transformed into a feasible schedule S for all jobs in J, such that $E(S) \leq (1+\epsilon)^\alpha E(S')$.*

The preceding lemma essentially ensures that representing the δ largest size classes of an "inherited" job set suffices if we allow a small increment of energy cost. The smaller jobs can then be added safely (i.e. without increasing the energy cost by too much) to the final schedule. We now turn to the central definition of the dynamic program. All schedules in this definition are with respect to the rounded instance $\mathcal{I}'$, and all grid points relate to the set $\mathcal{P}_{\text{approx}}$.

Definition 9. *For any vertex v in the tree T, any job vector $\overrightarrow{\lambda}$ that is heritable to v, and any pair of grid points $g_1 \leq g_2$ with $[g_1, g_2) \subseteq I_v$, let $G(v, \overrightarrow{\lambda}, g_1, g_2)$ denote a minimum cost grid point schedule for the jobs in the subtree of v (including v itself) plus the jobs $J(\overrightarrow{\lambda})$ (these are allowed to be scheduled anywhere inside $[g_1, g_2)$) that uses only the interval $[g_1, g_2)$. Furthermore, let $S(v, \overrightarrow{\lambda}, g_1, g_2)$ be a δ-omitted schedule for the same set of jobs in the same interval $[g_1, g_2)$, satisfying $E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big) \leq E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$.*

Dynamic Program. Our dynamic program computes the schedules $S(v, \overrightarrow{\lambda}, g_1, g_2)$. For ease of exposition, we focus only on computing the energy consumption values $E(v, \overrightarrow{\lambda}, g_1, g_2) := E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big)$, and omit the straightforward bookkeeping of the corresponding schedules. The base cases are the leaves of T. For a particular leaf node ℓ, we set

$$E(\ell, \overrightarrow{\lambda}, g_1, g_2) := \begin{cases} 0 & \text{if } J_\ell \cup J(\overrightarrow{\lambda}) = \emptyset \\ \frac{V^\alpha}{(g_2 - g_1)^{\alpha-1}} & \text{otherwise,} \end{cases}$$

where V is the total volume of all jobs in $J_\ell \cup J(\overrightarrow{\lambda})$. This corresponds to executing $J_\ell \cup J(\overrightarrow{\lambda})$ at uniform speed using the whole interval $[g_1, g_2)$. The resulting schedule is feasible, as no release times or deadlines occur in the interior of I_ℓ. Furthermore, it is also optimal by the convexity of the power function. Thus $E(\ell, \overrightarrow{\lambda}, g_1, g_2) \leq E\big(G(\ell, \overrightarrow{\lambda}, g_1, g_2)\big)$.

When all leaves have been handled, we move on to the next level, the parents of the leaves. For this and also the following levels up to the root r, we compute the values $E(v, \overrightarrow{\lambda}, g_1, g_2)$ recursively, using the procedure Compute in Figure 2. An intuitive description of the procedure is given below.

```
Compute $(v, \overrightarrow{\lambda}, g_1, g_2)$:

Let $v_1$ and $v_2$ be the children of $v$, such that $I_{v_1}$ is the earlier of the intervals $I_{v_1}, I_{v_2}$. Furthermore, let $g$ be the grid point at which $I_{v_1}$ ends and $I_{v_2}$ starts.
**Initialize** MIN $:= \infty$.
**For** all gridpoints $\tilde{g_1}, \tilde{g_2}$, s.t. $g_1 \le \tilde{g_1} < g < \tilde{g_2} \le g_2$, and all jobs $j \in J_v \cup J(\overrightarrow{\lambda})$, **do:**
    $E := \frac{{v_j}^\alpha}{(\tilde{g_2} - \tilde{g_1})^{\alpha-1}}$;   $\tilde{J} := \big(J_v \cup J(\overrightarrow{\lambda})\big) \setminus \{j\}$;   $\overrightarrow{\gamma} :=$ Vector $(\tilde{J})$.
    MIN $:= \min\big\{$MIN,
        $\min\{E + E(v_1, \overrightarrow{\gamma_1}, g_1, \tilde{g_1}) + E(v_2, \overrightarrow{\gamma_2}, \tilde{g_2}, g_2) : J(\overrightarrow{\gamma_1}) \cup J(\overrightarrow{\gamma_2}) = J(\overrightarrow{\gamma})\}\big\}$.
$\tilde{J} := J_v \cup J(\overrightarrow{\lambda})$;   $\overrightarrow{\gamma} :=$ Vector $(\tilde{J})$.
$a_1 := \min\{g_1, g\}$; $a_2 := \min\{g_2, g\}$; $b_1 := \max\{g_1, g\}$; $b_2 := \max\{g_2, g\}$.
$E(v, \overrightarrow{\lambda}, g_1, g_2) := \min\big\{$MIN,
    $\min\{E(v_1, \overrightarrow{\gamma_1}, a_1, a_2) + E(v_2, \overrightarrow{\gamma_2}, b_1, b_2) : J(\overrightarrow{\gamma_1}) \cup J(\overrightarrow{\gamma_2}) = J(\overrightarrow{\gamma})\}\big\}$.
--------------------------------------------------------------------------------
Vector $(\tilde{J})$:

Let $\mathcal{C}_\ell$ be the largest size class of any job in $\tilde{J}$.
$i := \max\{\ell, \delta - 1\}$.
**For** $k := i - \delta + 1, \ldots, i$ **do:**  $x_k := |\{p \in \tilde{J} : p$ belongs to size class $\mathcal{C}_k\}|$.
**Return**  $(i, x_i, x_{i-1}, \ldots, x_{i-\delta+1})$.
```

Fig. 2. Procedure for computing the remaining entries of the DP.

Our first step is to iterate through all possible options for a potential "crossing" job j, whose execution interval $[\tilde{g_1}, \tilde{g_2})$ stretches from child v_1 into the interval of child v_2. For every possible choice, we combine the optimal energy cost E for this job (obtained by using a uniform execution speed) with the best possible way to split up the remaining jobs between the truncated intervals of v_1 and v_2. Here we consider only the δ largest size classes of the remaining jobs $\tilde{J}$, and omit the smaller jobs. This omitting happens during the construction of a vector representation for $\tilde{J}$ using the procedure Vector. Finally, we also try the option that no "crossing" job exists and all jobs are split up between v_1 and v_2. In this case we need to take special care of the subproblem boundaries, as $g_1 > g$ or $g_2 < g$ are also valid arguments for Compute.

Lemma 5. *The schedules $S(v, \overrightarrow{\lambda}, g_1, g_2)$ constructed by the above dynamic program are δ-omitted schedules for the jobs in the subtree of v plus the jobs $J(\overrightarrow{\lambda})$. Furthermore, they satisfy $E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big) \leq E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$.*

Combining Lemmas 1, 3, 4, and 5 we can now state our main theorem. A detailed proof is provided in the appendix.

Theorem 1. *The non-preemptive speed scaling problem admits a QPTAS if the instance is laminar.*

4 Equal-Volume Jobs

In this section, we consider the case that all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$. We present a dynamic program that computes an (exact) optimal schedule for this setting in polynomial time. All grid points used for this purpose relate to the set $\mathcal{P}_{\text{exact}}$.

As a first step, let us order the jobs such that $r_1 \leq r_2 \leq \ldots \leq r_n$. Furthermore, let us define an ordering on schedules as follows.

Definition 10 (Completion Time Vector). *Let $C_1, \ldots, C_n$ be the completion times of the jobs $j_1, \ldots, j_n$ in a given schedule S. The vector $\overrightarrow{S} := (C_1, \ldots, C_n)$ is called the* completion time vector *of S.*

Definition 11 (Lexicographic Ordering). *A schedule S is said to be* lexicographically smaller *than a schedule S' if the first component in which their completion time vectors differ is smaller in $\overrightarrow{S}$ than in $\overrightarrow{S'}$.*

We now elaborate the details of the DP, focusing on energy consumption values only.

Definition 12. *Let $i \in \{1, \ldots, n\}$ be a job index, and let $g_1, g_2,$ and g_3 be grid points satisfying $g_1 \leq g_2 \leq g_3$. We define $E(i, g_1, g_2, g_3)$ to be the minimum energy consumption of a grid point schedule for the jobs $\{j_k \in \mathcal{J} : k \geq i \ \wedge\ g_1 < d_k \leq g_3\}$ that uses only the interval $[g_1, g_2)$.*

Dynamic Program. Our goal is to compute the values $E(i, g_1, g_2, g_3)$. To this end, we let

$$E(i, g_1, g_2, g_3) := \begin{cases} 0 & \text{if } \{j_k \in \mathcal{J} : k \geq i \ \wedge\ g_1 < d_k \leq g_3\} = \emptyset \\ \infty & \text{if } \exists k \geq i : g_1 < d_k \leq g_3 \ \wedge\ [r_k, d_k) \cap [g_1, g_2) = \emptyset. \end{cases}$$

Note that if $g_1 = g_2$, one of the above cases must apply. We now recursively compute the remaining values, starting with the case that g_1 and g_2 are consecutive grid points, and stepwise moving towards cases with more and more grid points in between g_1 and g_2. The recursion works as follows. Let $E(i, g_1, g_2, g_3)$ be the value we want to compute, and let j_q be the smallest index job in $\{j_k \in \mathcal{J} : k \geq i \ \wedge\ g_1 < d_k \leq g_3\}$. Furthermore, let $\mathcal{G}$ denote a lexicographically smallest optimal grid point schedule for the jobs $\{j_k \in \mathcal{J} : k \geq i \wedge g_1 < d_k \leq g_3\}$, using only the interval $[g_1, g_2)$. Our first step is to "guess" the grid points b_q and e_q that mark the beginning and end of j_q's execution interval in $\mathcal{G}$, by minimizing over all possible options. We then use the crucial observation that in $\mathcal{G}$, all jobs $J^- := \{j_k \in \mathcal{J} : k \geq q+1 \ \wedge \ g_1 < d_k \leq e_q\}$ are processed completely before j_q, and all jobs $J^+ := \{j_k \in \mathcal{J} : k \geq q+1 \ \wedge \ e_q < d_k \leq g_3\}$ are processed completely after j_q. For J^- this is obviously the case because of the deadline constraint. For J^+ this holds as all these jobs have release time at least r_q by the ordering of the jobs, and deadline greater than e_q by definition of J^+. Therefore any job in J^+ that is processed before j_q could be swapped with j_q, resulting in a lexicographic smaller schedule; a contradiction. Hence, we can use the following recursion to compute $E(i, g_1, g_2, g_3)$.

$$E(i, g_1, g_2, g_3) := \min \Big\{ \frac{{v_q}^\alpha}{(e_q - b_q)^{\alpha-1}} + E(q+1, g_1, b_q, e_q) + E(q+1, e_q, g_2, g_3) : \\ (g_1 \leq b_q < e_q \leq g_2) \wedge (b_q \geq r_q) \wedge (e_q \leq d_q) \Big\}.$$

Once we have computed all values, we output the schedule S corresponding to $E(1, r^*, d^*, d^*)$, where r^* is the earliest release time and d^* the latest deadline of any job in $\mathcal{J}$. Lemma 2 implies that $E(S) = \mathrm{OPT}$. The running time complexity of this algorithm is $\mathcal{O}(n^{21})$: There are $\mathcal{O}(n^4)$ grid points in $\mathcal{P}_{\text{exact}}$, and thus $\mathcal{O}(n^{13})$ entries to compute. To calculate one entry, we need to minimize over $\mathcal{O}(n^8)$ different options.

Theorem 2. *The non-preemptive speed scaling problem admits a polynomial time algorithm if all jobs have the same volume.*

5 Conclusion

In this paper, we made a step towards narrowing down the complexity of the non-preemptive speed scaling problem. The most interesting open question is whether a (Q)PTAS is also possible for general instances. Some of our techniques, such as the grid point discretization or δ-omitted schedules, can also be applied to this setting. The problematic part is that our QPTAS relies on the tree structure of the time windows, which is only given in laminar instances. It is unclear whether and how this approach can be refined to deal with the general case.

References

1. Albers, S., Antoniadis, A., Greiner, G.: On multi-processor speed scaling with migration: extended abstract. In: SPAA. pp. 279–288. ACM (2011)
2. Albers, S., Fujiwara, H.: Energy-efficient algorithms for flow time minimization. In: STACS. LNCS, vol. 3884, pp. 621–633. Springer (2006)
3. Albers, S., Müller, F., Schmelzer, S.: Speed scaling on parallel processors. In: SPAA. pp. 289–298. ACM (2007)

4. Angel, E., Bampis, E., Chau, V.: Throughput maximization in the speed-scaling setting, arXiv:1309.1732
5. Antoniadis, A., Huang, C.C.: Non-preemptive speed scaling. In: SWAT. LNCS, vol. 7357, pp. 249–260. Springer (2012)
6. Azar, Y., Epstein, L., Richter, Y., Woeginger, G.J.: All-norm approximation algorithms. J. Algorithms 52(2), 120–133 (2004)
7. Bampis, E., Kononov, A., Letsios, D., Lucarelli, G., Nemparis, I.: From preemptive to non-preemptive speed-scaling scheduling. In: COCOON. LNCS, vol. 7936, pp. 134–146. Springer (2013)
8. Bampis, E., Kononov, A., Letsios, D., Lucarelli, G., Sviridenko, M.: Energy efficient scheduling and routing via randomized rounding. In: FSTTCS. LIPIcs, vol. 24, pp. 449–460. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
9. Bansal, N., Chan, H.L., Lam, T.W., Lee, L.K.: Scheduling for speed bounded processors. In: ICALP (1). LNCS, vol. 5125, pp. 409–420. Springer (2008)
10. Bansal, N., Chan, H.L., Pruhs, K.: Speed scaling with an arbitrary power function. In: SODA. pp. 693–701. SIAM (2009)
11. Bansal, N., Pruhs, K., Stein, C.: Speed scaling for weighted flow time. In: SODA. pp. 805–813. SIAM (2007)
12. Bingham, B.D., Greenstreet, M.R.: Energy optimal scheduling on multiprocessors with migration. In: ISPA. pp. 153–161. IEEE (2008)
13. Brooks, D., Bose, P., Schuster, S., Jacobson, H.M., Kudva, P., Buyuktosunoglu, A., Wellman, J.D., Zyuban, V.V., Gupta, M., Cook, P.W.: Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. IEEE Micro 20(6), 26–44 (2000)
14. Chan, H.L., Chan, J.W.T., Lam, T.W., Lee, L.K., Mak, K.S., Wong, P.W.H.: Optimizing throughput and energy in online deadline scheduling. ACM Transactions on Algorithms 6(1) (2009)
15. Chen, J.J., Kuo, T.W., Lu, H.I.: Power-saving scheduling for weakly dynamic voltage scaling devices. In: WADS. LNCS, vol. 3608, pp. 338–349. Springer (2005)
16. Cohen-Addad, V., Li, Z., Mathieu, C., , Mills, I.: Energy-efficient algorithms for non-preemptive speed-scaling, arXiv:1402.4111v2
17. Han, X., Lam, T.W., Lee, L.K., To, I.K.K., Wong, P.W.H.: Deadline scheduling and power management for speed bounded processors. Theor. Comput. Sci. 411(40-42), 3587–3600 (2010)
18. Irani, S., Shukla, S.K., Gupta, R.K.: Algorithms for power savings. In: SODA. pp. 37–46. ACM/SIAM (2003)
19. Li, M., Liu, B.J., Yao, F.F.: Min-energy voltage allocation for tree-structured tasks. In: COCOON. LNCS, vol. 3595, pp. 283–296. Springer (2005)
20. Li, M., Yao, F.F.: An efficient algorithm for computing optimal discrete voltage schedules. In: MFCS. LNCS, vol. 3618, pp. 652–663. Springer (2005)
21. Muratore, G., Schwarz, U.M., Woeginger, G.J.: Parallel machine scheduling with nested job assignment restrictions. Oper. Res. Lett. 38(1), 47–50 (2010)
22. Negrean, M., Ernst, R.: Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In: SIES. pp. 191–200. IEEE (2012)
23. Wierman, A., Andrew, L.L.H., Tang, A.: Power-aware speed scaling in processor sharing systems: Optimality and robustness. Perform. Eval. 69(12), 601–622 (2012)
24. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced cpu energy. In: FOCS. pp. 374–382. IEEE Computer Society (1995)

A Proof of Proposition 1

Proposition 1. *Let S and S' be two feasible schedules that process j using uniform speeds s and $s' > s$, respectively. Then $E(S',j) = (s'/s)^{\alpha-1} \cdot E(S,j)$.*

Proof.

$$\begin{aligned} E(S',j) &= P(s')\frac{v_j}{s'} = (s')^{\alpha-1}v_j = (\frac{s'}{s})^{\alpha-1}s^{\alpha-1}v_j \\ &= (\frac{s'}{s})^{\alpha-1}P(s)\frac{v_j}{s} = (\frac{s'}{s})^{\alpha-1}E(S,j). \end{aligned}$$

□

B Proof of Lemma 1

Lemma 1. *There exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{approx}}$, such that $E(\mathcal{G}) \leq (1+\epsilon)^{\alpha-1}\text{OPT}$.*

Proof. Let $\mathcal{S}^*$ be an optimal schedule, that is $E(\mathcal{S}^*) = \text{OPT}$. We show how to modify $\mathcal{S}^*$ by shifting and compressing certain jobs, s.t. every execution interval starts and ends at a grid point. For the proof we focus on one particular zone (t_i, t_{i+1}), and the lemma follows by applying the transformation to each other zone individually.

Let us consider the jobs that $\mathcal{S}^*$ processes within the zone (t_i, t_{i+1}). If a job's execution interval overlaps partially with this zone, we consider only its fraction inside (t_i, t_{i+1}) and treat this fraction as if it were a job by itself. We denote the set of (complete and partial) jobs in zone (t_i, t_{i+1}) by J. If $J = \emptyset$, nothing needs to be done. Otherwise, we can assume that $\mathcal{S}^*$ uses the entire zone (t_i, t_{i+1}) without any idle periods to process the jobs in J. If this were not the case, we could slow down the processing of any job in J without violating a release time or deadline constraint, and thus obtain a feasible schedule with lower energy cost than $\mathcal{S}^*$, a contradiction. Consequently, the total time for processing J in $\mathcal{S}^*$ is $\gamma n^2 L_i$ (recall that $L_i = \frac{t_{i+1}-t_i}{n^2\gamma}$), and as $|J| \leq n$, there must exist a job $j \in J$ with execution time $T_j \geq \gamma n L_i$.

We now partition the jobs in $J \setminus j$ into J^+, the jobs processed after j, and J^-, the jobs processed before j. First, we restrict our attention to J^+. Let $q_1, \dots, q_{|J^+|}$ denote the jobs in J^+ in the order they are processed by $\mathcal{S}^*$. Starting with the last job $q_{|J^+|}$, and going down to q_1, we modify the schedule as follows. We keep the end of $q_{|J^+|}$'s execution interval fixed, and shift its start to the next earlier grid point, reducing its uniform execution speed accordingly. At the same time, to not produce any overlappings, we shift the execution intervals of all $q_k,\ k < |J^+|$ by the same amount, in the direction of earlier times (leaving their lengths unchanged). Eventually, we also move the execution end point of j by the same amount towards earlier times (leaving its start point fixed). This shortens the execution interval of j and "absorbs" the shifting of the jobs in J^+. The shortening of j's execution interval is compensated by an appropriate increase of speed. We then proceed with $q_{|J^+|-1}$, keeping its end (which now already resides at a grid point) fixed, and moving its start to the next earlier grid point. Again, the shift propagates to earlier jobs in J^+, which are moved by the same amount, and shortens j's execution interval once more. When all jobs in J^+ have been modified in this way, we turn to J^- and apply the same procedure there. This time, we keep the start times fixed and instead shift the right end points of the execution intervals towards later times. As before, j "absorbs" the propagated shifts, as we increase its start time accordingly. After this modification, the execution intervals of all jobs in J start and end at grid points only.

To complete the proof, we need to analyze the changes made in terms of energy consumption. Let $\mathcal{G}$ denote the schedule obtained by the above modification of $\mathcal{S}^*$. Obviously, for all $j' \in J \setminus j$, we have that $E(\mathcal{G}, j') \leq E(\mathcal{S}^*, j')$, as the execution intervals of those jobs are only prolonged during the transformation process, resulting in a less or equal execution speed. The only job whose processing time is possibly shortened, is j. Since $|J| \leq n$, it can be shortened at most n times, each time by a length of at most L_i. Remember that the execution time of j in $\mathcal{S}^*$ was $T_j \geq \gamma n L_i$. Therefore, in $\mathcal{G}$, its execution time is at least $T_j - nL_i \geq T_j - T_j/\gamma$. Thus the speedup factor of j in $\mathcal{G}$ compared to $\mathcal{S}^*$ is at most

$$\frac{T_j}{T_j - \frac{T_j}{\gamma}} = \frac{1}{1 - \frac{1}{\gamma}} \leq 1 + \epsilon,$$

where the last inequality follows from the definition of γ. Hence, Proposition 1 implies that $E(\mathcal{G}, j) \leq (1+\epsilon)^{\alpha-1} E(\mathcal{S}^*, j)$, and the lemma follows by summing up the energy consumptions of the individual jobs. □

C Proof of Lemma 2

Lemma 2. *If all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$, there exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{exact}}$, such that $E(\mathcal{G}) = \text{OPT}$.*

Proof. Let $\mathcal{S}^*$ be an optimal schedule. W.l.o.g., we can assume that $\mathcal{S}^*$ changes the processing speed only at events (recall that an event is either a release time or a deadline of some job), as a constant average speed between any two consecutive events minimizes the energy consumption (this follows from Jensen's Inequality) without violating release time or deadline constraints. Given this property, we will show that $\mathcal{S}^*$ is in fact a grid point schedule with respect to $\mathcal{P}_{\text{exact}}$. To this end, we partition the time horizon of $\mathcal{S}^*$ into *phases* of constant speed, that is time intervals of maximal length during which the processing speed is unchanged. As every job itself is processed using a uniform speed, no job is processed only partially within a phase. Each phase is therefore characterized by a pair of events $t_i \leq t_j$ indicating its beginning and end, and a number x of jobs that are processed completely between t_i and t_j at constant speed. It is clear that the grid points created for the pair (t_i, t_j) and $k := x$ in the definition of $\mathcal{P}_{\text{exact}}$ correspond exactly to the start and end times of the jobs in this phase. Since this is true for every phase, $\mathcal{S}^*$ is indeed a grid point schedule. □

D Proof of Lemma 3

Lemma 3. *Every feasible schedule S' for $\mathcal{I}'$ can be transformed into a feasible schedule S for $\mathcal{I}$ with $E(S) \leq (1+\epsilon)^\alpha E(S')$.*

Proof. The lemma easily follows by using the same execution intervals as S' and speeding up accordingly. As rounded and original volume of a job differ by at most a factor of $1+\epsilon$, we need to increase the speed at any time t by at most this factor. Therefore the energy consumption grows by at most a factor of $(1+\epsilon)^\alpha$. □

E Proof of Lemma 4

Lemma 4. *Every δ-omitted schedule S' for a set of jobs J can be transformed into a feasible schedule S for all jobs in J, such that $E(S) \leq (1+\epsilon)^\alpha E(S')$.*

Proof. Let R be the set of non-omitted jobs in S'. W.l.o.g., we can assume that S' executes each job in R at a uniform speed, as this minimizes the energy consumption. For every $j \in R$, define $\mathrm{SMALL}(j) := \{x \in J \setminus R : \mathrm{big}(x) = j\}$. Note that every omitted job occurs in exactly one of the sets $\mathrm{SMALL}(j),\ j \in R$. The schedule S is constructed as follows. For all $j \in R$, we process the jobs $\{j\} \cup \mathrm{SMALL}(j)$ using the execution interval of j in S' and a uniform speed. The processing order may be chosen arbitrarily. Clearly, the resulting schedule is feasible by the definition of $\mathrm{big}(x)$. In order to finish the total volume V_j of the jobs $\{j\} \cup \mathrm{SMALL}(j)$ within the interval of j in S', we need to raise the speed in this interval by the factor V_j/v_j. As $|\mathrm{SMALL}(j)| \leq n$, and $v_x \leq v_j(1+\epsilon)^{-\delta}$ for all $x \in \mathrm{SMALL}(j)$, we have that

$$V_j \leq v_j + nv_j(1+\epsilon)^{-\delta} \leq v_j + nv_j\frac{\epsilon}{n} \leq (1+\epsilon)v_j,$$

where the second inequality follows from the definition of δ. For the speedup factor, we therefore obtain $V_j/v_j \leq 1+\epsilon$. Hence, the energy consumption grows by at most the factor $(1+\epsilon)^\alpha$. □

F Proof of Lemma 5

Lemma 5. *The schedules $S(v, \overrightarrow{\lambda}, g_1, g_2)$ constructed by the above dynamic program are δ-omitted schedules for the jobs in the subtree of v plus the jobs $J(\overrightarrow{\lambda})$. Furthermore, they satisfy $E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big) \leq E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$.*

Proof. We prove the lemma by induction. In the base cases, that is at the leaves of T, we already argued that the schedules are feasible and optimal. Since no jobs are omitted at all, the lemma is obviously true at this level. We now perform the induction step. To this end, let us consider a fixed schedule $S(v, \overrightarrow{\lambda}, g_1, g_2)$, and let us assume the lemma is true for the children v_1 and v_2 of v. We first show that $S(v, \overrightarrow{\lambda}, g_1, g_2)$ is indeed a δ-omitted schedule. The only point where jobs are omitted in the recursive procedure is the call of VECTOR $(\tilde{J})$, where a vector-representation $\overrightarrow{\gamma}$ of $\tilde{J}$ is constructed. This vector $\overrightarrow{\gamma}$ only represents a subset of the jobs $\tilde{J}$, namely the jobs in the δ largest size classes of $\tilde{J}$. Let $j_{\max}$ denote a job in $\tilde{J}$ with maximum volume, i.e. a job belonging to the largest size class. Then every omitted job j_{om} has volume at most $v_{j_{\max}}(1+\epsilon)^{-\delta}$, and we can choose $\text{big}(j_{om}) := j_{\max}$ to satisfy the requirements of Definition 8, as long as $j_{\max}$ is indeed contained in one of the subschedules that we combine in the recursion step. If, however, $j_{\max}$ is omitted in the corresponding subschedule, then there exists a job $\text{big}(j_{\max})$ as required in Definition 8, by induction hypothesis. In this case we can choose $\text{big}(j_{om}) := \text{big}(j_{\max})$. This proves that $S(v, \overrightarrow{\lambda}, g_1, g_2)$ is indeed a δ-omitted schedule.

We now argue about the energy consumption. Let J_1 and J_2 denote the subsets of jobs that $G(v, \overrightarrow{\lambda}, g_1, g_2)$ processes entirely within I_{v_1} and I_{v_2}, respectively. If there exists a "crossing" job spanning from I_{v_1} into I_{v_2}, we denote this job by j_c. Now we look at the iteration that handles exactly this situation, i.e. when $j = j_c$ and $\tilde{g_1}, \tilde{g_2}$ mark the beginning and end of j_c's execution interval in $G(v, \overrightarrow{\lambda}, g_1, g_2)$, or the passage after the for-loop for the case without "crossing" job. As mentioned earlier, the procedure possibly omits certain jobs and only splits up a subset of $J_v \cup J(\overrightarrow{\lambda})$ between the children v_1 and v_2. Since all possible splits are tried, one option for the min-operation is to combine subschedules that process a subset of J_1 within I_{v_1}, and a subset of J_2 within I_{v_2}. By induction hypothesis, and since we only schedule subsets of J_1 and J_2, the energy consumption of these subschedules is at most the energy spent by $G(v, \overrightarrow{\lambda}, g_1, g_2)$ for executing J_1 and J_2, respectively. Furthermore, if there exists a "crossing" job j_c, then executing this job from $\tilde{g_1}$ to $\tilde{g_2}$ at uniform speed costs at most the energy that $G(v, \overrightarrow{\lambda}, g_1, g_2)$ pays for this job. Summing up the different parts, we get that the considered option has an energy consumption of at most $E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$. The lemma follows as we choose the minimum over all possible options. □

G Proof of Theorem 1

Theorem 1. *The non-preemptive speed scaling problem admits a QPTAS if the instance is laminar.*

Proof. Let r^* be the earliest release time, and d^* be the latest deadline of any job in $\mathcal{J}$. Furthermore, let r be the root of the tree T, and let $\overrightarrow{0}$ denote the (heritable) job vector representing the empty set, i.e. $\overrightarrow{0} := (\delta - 1, 0, \dots, 0)$. We consider the schedule $S(r, \overrightarrow{0}, r^*, d^*)$, which is a δ-omitted schedule for the rounded instance by Lemma 5, and turn it into a feasible schedule S_r for the whole set of (rounded) jobs, using Lemma 4. Finally, we apply Lemma 3 to turn

S_r into a feasible schedule S for the original instance $\mathcal{I}$, and obtain

$$\begin{aligned}E(S) &\le (1+\epsilon)^{\alpha} E(S_r) \le (1+\epsilon)^{2\alpha} E\big(S(r, \overrightarrow{0}, r^*, d^*)\big) \\ &\le (1+\epsilon)^{2\alpha} E\big(G(r, \overrightarrow{0}, r^*, d^*)\big) \le (1+\epsilon)^{3\alpha-1}\mathrm{OPT} = \big(1+\mathcal{O}(\epsilon)\big)\mathrm{OPT}.\end{aligned}$$

Here, the third inequality holds by Lemma 5, and the fourth inequality follows from Lemma 1 and the fact that $G(r, \overrightarrow{0}, r^*, d^*)$ is an optimal grid point schedule for the rounded instance (with smaller job volumes). The quasipolynomial running time of the algorithm is easily verified, as we have only a polynomial number of grid points, and at most a quasipolynomial number of job vectors that are heritable to any vertex of the tree. □

H Purely-Laminar Instances

In this section, we present an FPTAS for a purely-laminar instance $\mathcal{I}$. W.l.o.g., we assume that the jobs are ordered by inclusion of their time windows, that is $[r_1, d_1) \subseteq [r_2, d_2) \subseteq \cdots \subseteq [r_n, d_n)$. Furthermore, whenever we refer to grid points in this section, we refer to the set $\mathcal{P}_{\text{approx}}$. Our FPTAS uses dynamic programming to construct an optimal grid point schedule for $\mathcal{I}$, satisfying the following structural property:

Property 1. For any $k > 1$, jobs $j_1, \ldots, j_{k-1}$ are either all processed before j_k, or all processed after j_k.

This structure can easily be established in any schedule for $\mathcal{I}$ by performing a sequence of energy-preserving swaps. According to this, the following lemma is a straightforward extension of Lemma 1 to the purely-laminar case.

Lemma 6. *If the problem instance is purely-laminar, there exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{approx}}$ that satisfies Property 1 and has energy cost $E(\mathcal{G}) \le (1+\epsilon)^{\alpha-1}\mathrm{OPT}$.*

Proof. Consider an optimal schedule $\mathcal{S}^*$ for $\mathcal{I}$, and let J^- and J^+ be the jobs executed before and after j_1, respectively. Now rearrange the execution intervals (without changing their lengths) of the jobs in J^+ into *smallest index first* order (SIF), by repeatedly swapping two consecutively processed jobs j_a preceding j_b, with $a > b$. For the swap, we let the execution interval of j_b now start at j_a's original starting time, and directly append j_a's execution interval once j_b is finished. Note that each such swap maintains feasibility, as no release times occurs during the execution of the jobs in J^+, and $a > b$ implies $d_a \ge d_b$. Similarly, we rearrange the execution intervals of the jobs in J^- into *largest index first* order (LIF), and denote the resulting schedule by S'. Clearly, $E(S') = \mathrm{OPT}$, as the rearrangements preserve the energy cost of every individual job. Furthermore, S' satisfies Property 1. To see this, let us fix $k > 1$ and distinguish whether j_k is in J^- or in J^+. In the first case, when $j_k \in J^-$, all $j \in J^+$ are scheduled after j_k by definition of J^-/J^+, and all $j_i \in J^-, i < k$ are scheduled after j_k by the LIF-order. In the second case, when $j_k \in J^+$, all $j \in J^-$ are scheduled before j_k by definition of J^-/J^+, and all $j_i \in J^+, i < k$ are scheduled before j_k by the SIF-order. As a final step, we now apply the transformation from the proof of Lemma 1 to S'. Since this transformation does not change the order of any jobs, the resulting grid point schedule $\mathcal{G}$ still satisfies Property 1, and has energy cost $E(\mathcal{G}) \leq (1+\epsilon)^{\alpha-1}\text{OPT}$. □

Dynamic Program. For any $k \leq n$ and grid points $g_1 \leq g_2$, let $S(k, g_1, g_2)$ denote a minimum cost grid point schedule for $j_1, \dots, j_k$ that satisfies Property 1 and uses only the time interval between g_1 and g_2. The corresponding energy cost of $S(k, g_1, g_2)$ is denoted by $E(k, g_1, g_2)$, where $E(k, g_1, g_2) := \infty$ if no such schedule exists. For ease of exposition, we only show how to compute the energy consumption values $E(k, g_1, g_2)$, and omit the straightforward bookkeeping of the corresponding schedules. The base cases are given by $E(0, g_1, g_2) = 0$, for all $g_1 \leq g_2$. All remaining entries can be computed with the following recursion.

$$E(k+1, g_1, g_2) = \begin{cases} \infty \quad \text{if } (g_1 = g_2) \vee (g_1 \geq d_{k+1}) \vee (g_2 \leq r_{k+1}). \\[2ex] \min\left\{ \frac{{v_{k+1}}^{\alpha}}{(g_2' - g_1')^{\alpha-1}} + \min\{E(k, g_1, g_1'), E(k, g_2', g_2)\} : \right. \\ \qquad \left. (g_1 \leq g_1' < g_2' \leq g_2) \wedge (g_1' \geq r_{k+1}) \wedge (g_2' \leq d_{k+1}) \right\} \\ \hfill \text{otherwise.} \end{cases}$$

Intuitively, we minimize over all possible combinations of grid points g_1' and g_2' that could mark the beginning and end of j_{k+1}'s execution. For fixed g_1' and g_2', it is best to process j_{k+1} at uniform speed, resulting in the energy cost ${v_{k+1}}^{\alpha}/(g_2' - g_1')^{\alpha-1}$ for this job. The remaining jobs $j_1, \dots, j_k$ must then be scheduled either before or after j_{k+1}, to satisfy Property 1. This fact is captured in the second min-operation of the formula. The constraints on g_1' and g_2' ensure that j_{k+1} can be feasibly scheduled in the chosen interval.

Once we have computed all values $E(k, g_1, g_2)$ (and their corresponding schedules), we output the schedule $\tilde{S} := S(n, r^*, d^*)$, where r^* is the earliest release time and d^* the latest deadline of any job in $\mathcal{I}$. Note that $\tilde{S}$ is an optimal grid point schedule with Property 1 for $\mathcal{I}$. Hence, Lemma 6 implies that $E(\tilde{S}) \leq (1+\epsilon)^{\alpha-1}\text{OPT} = \big(1 + \mathcal{O}(\epsilon)\big)\text{OPT}$. Finally, it is easy to verify that the running time of the algorithm is polynomial in n and $1/\epsilon$, since the total number of grid points in $\mathcal{P}_{\text{approx}}$ is $\mathcal{O}\big(n^3(1 + \frac{1}{\epsilon})\big)$. We therefore obtain the following theorem.

Theorem 3. *The non-preemptive speed scaling problem admits an FPTAS if the instance is purely-laminar.*

I Bounded Number of Time Windows

Let us consider a problem instance $\mathcal{I}$, and group together jobs that share the same time window. We refer to the group of jobs with time window $[r, d)$ as the *type* T_{rd}.

Theorem 4. *The non-preemptive speed scaling problem admits an FPTAS if the total number of types is at most a constant c.*

Our FPTAS draws on ideas of Antoniadis and Huang [5], as we transform the problem into an instance $\mathcal{I}'$ of *unrelated machine scheduling* with ℓ_α-norm objective. In this problem, one is given a set of machines $\mathcal{M}$, a set of jobs $\mathcal{J}$, and numbers p_{ij} that specify the processing time of job j on machine i. The goal is to find an assignment A of the jobs to the machines that minimizes $\mathrm{Cost}(A) = (\sum_{i \in \mathcal{M}}(\sum_{j:A(j)=i} p_{ij})^\alpha)^{1/\alpha}$. In general, this problem is APX-hard [6]. Our instance, however, will have only a constant number of machines, and for this special case an FPTAS exists [6].

The transformation works as follows. Let $\mathcal{G}$ be an optimal grid point schedule with respect to $\mathcal{P}_{\text{approx}}$, and for each type T_{rd} let $b(T_{rd})$ and $e(T_{rd})$ denote the grid points at which $\mathcal{G}$ starts to process the first job of T_{rd} and finishes the last job of T_{rd}, respectively. Our first step is to "guess" the entire set of grid points $b(\cdot)$ and $e(\cdot)$, by trying out all possible options with $r \le b(T_{rd}) < e(T_{rd}) \le d$ for every type T_{rd}. Note that the total number of choices that we have to make is at most $\mathcal{O}\big(n^{6c}(1+\frac{1}{\epsilon})^{2c}\big)$, and thus polynomial in both n and $1/\epsilon$. For one particular guess, let $g_1 < g_2 < \ldots < g_k$ be the ordered set of distinct grid points $b(\cdot)$ and $e(\cdot)$. The instance $\mathcal{I}'$ has a machine i for every interval $[g_i, g_{i+1})$, and a job j for every job of the original instance. The processing times p_{ij} are given as $p_{ij} := \frac{v_j}{(g_{i+1}-g_i)^{1-1/\alpha}}$ if $[g_i, g_{i+1}) \subseteq [r_j, d_j)$, and $p_{ij} := \infty$ otherwise.

Note that the total number of machines in $\mathcal{I}'$ is $k-1 < 2c$. Hence, the FPTAS of [6] can be applied to obtain an assignment A with $\mathrm{Cost}(A) \le (1+\epsilon)\mathrm{OPT}'$, where OPT' denotes the cost of an optimal assignment for $\mathcal{I}'$. The following two lemmas imply Theorem 4.

Lemma 7. *Every finite-cost assignment A for $\mathcal{I}'$ can be transformed into a schedule S for $\mathcal{I}$, such that $E(S) = \big(Cost(A)\big)^\alpha$.*

Proof. For any $i \in \mathcal{M}$, let A_i denote the set of jobs that A assigns to machine i. In order to create the schedule S, we iterate through all $i \in \mathcal{M}$ and process the jobs in A_i within the interval $[g_i, g_{i+1})$, using the uniform speed $(\sum_{j \in A_i} v_j)/(g_{i+1} - g_i)$. The resulting schedule is clearly feasible, as A has finite cost and every $j \in A_i$ thus satisfies $[g_i, g_{i+1}) \subseteq [r_j, d_j)$. For the energy consumption of S we get

$$E(S) = \sum_{i \in \mathcal{M}} \left(\frac{\sum_{j \in A_i} v_j}{g_{i+1} - g_i} \right)^\alpha (g_{i+1} - g_i) = \sum_{i \in \mathcal{M}} \left(\frac{\sum_{j \in A_i} v_j}{(g_{i+1} - g_i)^{1-1/\alpha}} \right)^\alpha$$

$$= \sum_{i \in \mathcal{M}} \left(\sum_{j \in A_i} p_{ij} \right)^\alpha = \Big(\mathrm{Cost}(A)\Big)^\alpha .$$

□

Lemma 8. *If the grid points $b(\cdot)$ and $e(\cdot)$ are guessed correctly, there exists an assignment A for $\mathcal{I}'$ with $Cost(A) \le \big((1+\epsilon)^{\alpha-1}\mathrm{OPT}\big)^{1/\alpha}$.*

Proof. Remember that $\mathcal{G}$ is an optimal grid point schedule for $\mathcal{I}$, and that the grid points $b(T_{rd})$ and $e(T_{rd})$ mark the time points at which $\mathcal{G}$ starts to process the first job of type T_{rd} and finishes the last job of T_{rd}, respectively. Now observe that in $\mathcal{G}$, every job j is processed entirely within some interval $[g_i, g_{i+1})$, satisfying $[g_i, g_{i+1}) \subseteq [r_j, d_j)$. This is true because $r_j \leq b(T_{r_j d_j}) < e(T_{r_j d_j}) \leq d_j$, and no job can stretch from an interval $[g_{x-1}, g_x)$ into $[g_x, g_{x+1})$ since g_x indeed marks the beginning or end of some job. Let A_i denote the set of jobs which are entirely processed within $[g_i, g_{i+1})$, and let A be the assignment that maps all jobs from A_i to machine i. The cost of A is given as

$$\begin{aligned}
\mathrm{Cost}(A) &= \Big(\sum_{i \in \mathcal{M}} \Big(\sum_{j \in A_i} p_{ij} \Big)^{\alpha} \Big)^{1/\alpha} = \Big(\sum_{i \in \mathcal{M}} \Big(\frac{\sum_{j \in A_i} v_j}{(g_{i+1} - g_i)^{1-1/\alpha}} \Big)^{\alpha} \Big)^{1/\alpha} \\
&= \Big(\sum_{i \in \mathcal{M}} \Big(\frac{\sum_{j \in A_i} v_j}{g_{i+1} - g_i} \Big)^{\alpha} (g_{i+1} - g_i) \Big)^{1/\alpha} \\
&\leq \Big(E(\mathcal{G}) \Big)^{1/\alpha} \leq \Big((1+\epsilon)^{\alpha-1} \mathrm{OPT} \Big)^{1/\alpha}.
\end{aligned}$$

Here the last two inequalities follow from the convexity of the power function and Lemma 1, respectively. $\square$