Deadline Scheduling and Power Management for Speed Bounded Processors*

Xin Han[†] Tak-Wah Lam[‡] Lap-Kei Lee[§] Isaac K. K. To[¶] Prudence W. H. Wong[¶]

January 28, 2010

Abstract

In this paper we consider online deadline scheduling on a processor that can manage its energy usage by scaling the speed dynamically or entering a sleep state. A new online scheduling algorithm called SOA is presented. Assuming speed can be scaled arbitrarily high (the infinite speed model), SOA can complete all jobs with reduced energy usage, improving the competitive ratio for energy from $2^{2\alpha-2}\alpha^\alpha + 2^{\alpha-1} + 2$ [17] to $\alpha^\alpha + 2$, where α is the constant involved in the speed-to-power function, commonly believed to be 2 or 3. More importantly, SOA is the first algorithm that works well even if the processor has a fixed maximum speed and the system is overloaded. In this case, SOA is 4-competitive for throughput and $(\alpha^\alpha + \alpha^2 4^\alpha + 2)$-competitive for energy. Note that the throughput ratio cannot be better than 4 even if energy is not a concern.

Keywords: Power saving, speed scaling, sleep state, deadline scheduling, online algorithm, competitive analysis

1 Introduction

Speed scaling and deadline feasibility. Energy efficiency has become a major issue in the design of microprocessors, especially for battery-operated devices. A modern technology named *dynamic speed scaling* [11,15,23,24] enables a processor to vary the speed dynamically. Running a job slower reduces energy usage, but it takes longer and may affect performance. In the past few years, a lot of effort has been devoted to revisiting online job scheduling with speed scaling and energy usage taken into consideration. The challenge arises from the conflicting objectives of providing good "quality of service" (QoS) and conserving energy. In the model being considered, a processor, when running at speed s, consumes energy at the rate of s^α, where α is typically 2 [22] or 3 (the cube-root rule [11]). These studies first focused on the *infinite speed model* (e.g., [7, 8, 25]) where the processor speed can be scaled arbitrarily high, and have recently shifted to the more realistic *bounded speed model* (e.g., [4,12,19]) which imposes a bound T on the maximum allowable speed.

*A preliminary version of this paper has appeared in the Proceedings of the 9th Workshop on Models and Algorithms for Planning and Scheduling Problems

[†]Department of Mathematical Informatics, University of Tokyo, Japan. Email: hanxin.mail@gmail.com

[‡]Department of Computer Science, University of Hong Kong. Email: twlam@cs.hku.hk

[§]Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany. Email: lklee@mpi-inf.mpg.de

[¶]Department of Computer Science, University of Liverpool. Email: {isaacto, pwong}@liverpool.ac.uk

Deadline feasibility is a commonly used QoS measure for job scheduling. In the online setting, we assume that jobs with sizes and deadlines can arrive at unpredictable times. These jobs are to be run on a processor. Note that preemption is allowed and has no penalty. When scheduling jobs, the primary concern is the throughput, i.e., the total size of jobs completed entirely by their deadlines. When energy is also a concern, we want to achieve the maximum throughput while using the minimum amount of energy.

The theoretical study of energy efficient algorithms for deadline scheduling was pioneered by Yao, Demers and Shenker [25]. They considered the infinite speed model, which makes it feasible to complete all jobs by their deadlines. The only concern is the energy usage. Yao et al. [25] gave a simple online speed scaling algorithm called AVR that is $2^{\alpha-1}\alpha^{\alpha}$-competitive for energy, and proposed another algorithm called OA (Optimal Available). Bansal, Kimbrel and Pruhs [7] eventually showed that OA is α^{α}-competitive; they also gave a $2(\alpha/(\alpha-1))^{\alpha}e^{\alpha}$-competitive algorithm (which is called BKP and is better than OA if α is larger than 5). Recently, the result is improved by the algorithm qOA [6] with the competitive ratio $4^{\alpha}/(2\sqrt{e\alpha})$. Albers et al. [2] have also considered this problem in the multiprocessor setting.

When the maximum processor speed is bounded, it is not always possible to complete all jobs and it is no longer trivial how to select the jobs so as to strike a balance between throughput and energy. Chan et al. [12] were the first to study the bounded speed model. They proposed a job selection algorithm FSA which, when coupled with the speed scaling algorithm OAT (OA capped at T), is 14-competitive for throughput and $(\alpha^{\alpha}+\alpha^{2}4^{\alpha})$-competitive for energy. Later, Bansal et al. [4] proposed a more aggressive job selection algorithm Slow-D, improving the throughput ratio to 4, while maintaining the same energy ratio. Note that no algorithm can be better than 4-competitive for throughput even if energy is not a concern [9].

The study of online speed scaling and energy-efficient scheduling has been extended to other QoS measures. In particular, the problem of minimizing flow time and energy has attracted a lot of attention [1, 4, 5, 8, 13, 14, 20, 21].

Sleep management plus speed scaling. In older days when the speed scaling technology was not available, energy reduction was mainly achieved by allowing a processor to enter a low-power *sleep state*, yet waking up requires extra energy. In the (embedded system) literature, there are different energy-efficient strategies to bring a processor to sleep during a period of zero load [10]. This online problem is usually referred to as *dynamic power management*, in which the input comprises the start time and finish time of the zero-load period (the latter is known only when the period ends). There are some interesting results with competitive analysis (e.g., [3, 16, 18]). The problem assumes the processor is in either the *awake* state or the *sleep* state. The awake state always requires a static power $\sigma > 0$. Only when the processor is in the sleep state, the energy usage is zero, yet a wake-up back to the awake state requires $\omega > 0$ energy.

It is natural to extend the model of energy-efficient scheduling to allow a processor to exploit both speed scaling and sleep state. Precisely, we assume that, in the awake state, a processor running at speed $s \geq 0$ consumes energy at the rate $s^{\alpha} + \sigma$, where $\sigma > 0$ is the static power and s^{α} is the dynamic power[1]. In this case, job scheduling involves three components: (i) determine when to sleep and wake up; (ii) when not sleeping, determine which job to run; and (iii) determine at what speed to run a job. In contrast to the dynamic power management problem, here the length of the sleep period is part of the optimization (instead of the input). Adding a sleep state changes the nature of speed scaling. Without sleep state, running a job

[1]Static power is dissipated due to leakage current and is independent of processor speed, and dynamic power is due to dynamic switching loss and increases with the speed.

	infinite speed model	bounded speed model
Throughput	1	4 [this paper]
Energy usage	$2^{2\alpha-2}\alpha^\alpha + 2^{\alpha-1} + 2$ [17] $\max\{\alpha^\alpha + 2, 4\}$ [this paper]	$\alpha^\alpha + \alpha^2 4^\alpha + 2$ [this paper]

Table 1: Competitive ratios for throughput and energy when scheduling on a processor that exploits both speed scaling and sleep management.

slower is a natural way to save energy. With sleep state, one can also save energy by working faster to allow a longer sleep period. It is not trivial how to strike a balance. Notice that AVR, OA, BKP, and qOA no longer perform well, and their competitive ratios can be made arbitrarily large.

Irani et al. [17] were the first to study deadline scheduling that exploits both speed scaling and sleep management. Under the infinite speed model, they proposed a sleep management algorithm called PROCRASTINATOR to work with AVR or any speed scaling algorithms that are "additive" and "monotone" (see [17] for definitions). The resulting algorithm can complete all jobs and is $(2^{2\alpha-2}\alpha^\alpha + 2^{\alpha-1} + 2)$-competitive for energy.[2] Unfortunately, OA, BKP and qOA are not additive; otherwise PROCRASTINATOR can yield a better ratio. Furthermore, PROCRASTINATOR does not work for the bounded speed model as it requires extra speed to catch up with the optimal offline algorithm. It was indeed an open problem whether there exists an algorithm that can exploit sleep management and bounded speed scaling effectively and has constant competitive ratios for both throughput and energy [12]. On the other hand, the problem seems easier when one switches the context to flow time scheduling; recently, Lam et al. [19] has given a sleep management algorithm that fits well with existing speed scaling algorithms using a speed bounded processor, yielding a competitive algorithm for minimizing flow time plus energy.

Our contribution. In this paper, we present an online algorithm SOA that can exploit speed scaling and sleep state to support energy-efficient deadline scheduling. SOA can be considered as the sleep-aware version of the existing speed scaling algorithm OA [25]. SOA improves the previous result on the infinite speed model and is the first competitive algorithm for the bounded speed model (Table 1 gives a summary).

- In the infinite speed model, SOA completes all jobs and is $\max\{\alpha^\alpha + 2, 4\}$-competitive for energy, improving the ratio of Irani et al. [17]. Note that $\alpha^\alpha + 2 > 4$ if $\alpha \geq 2$.
- The major contribution of SOA is on the bounded speed model. We show that SOA capped at the maximum speed T can support the job selection strategy Slow-D [4] to give the first online algorithm that is 4-competitive for throughput and $(\alpha^\alpha + \alpha^2 4^\alpha + 2)$-competitive for energy.

An interesting feature of PROCRASTINATOR is that when jobs are released while the processor is in sleep state, execution of these jobs is delayed to prolong the sleep period until the amount of work accumulates to a certain level. To compensate for the delay, extra speed on top of the speed recommended by the speed scaling algorithm (e.g., AVR) is needed to execute these delayed jobs. This approach, however, is difficult to use in the bounded speed model. We cannot

[2]In general, using a c-competitive speed scaling algorithm, PROCRASTINATOR would yield a ratio of $2^{\alpha-1}c + 2^{\alpha-1} + 2$.

increase the processor speed above T to catch up with the delayed jobs. If we simply cap the speed of Procrastinator at T, it would not be fast enough to support any reasonable job selection strategy like FSA or Slow-D, and the throughput ratio is unbounded.

Our new algorithm SOA is based on OA. It does not avoid delaying job execution when the processor is idle. But it does not rely on extra speed to run the delayed jobs and can be used in the bounded speed model. An important observation is that we can treat the delayed jobs as if they were just released at the moment when the processor wakes up, and count on a more complicated analysis to prove that this is indeed enough to achieve a sufficiently large throughput. As we do not have to rely on extra speed to run the delayed jobs, the speed of SOA only increases modestly even during "busy" periods and SOA maintains $O(1)$-competitive for energy.

2 Preliminaries

The input is a job sequence arriving online. We denote the release time, work requirement (or size) and deadline of a job J as $r(J)$, $w(J)$ and $d(J)$, respectively. We schedule the jobs on a single processor. Preemption is allowed; a preempted job can resume at the point of preemption. The *throughput* is defined as the total work of the jobs completed by their deadlines. To ease our discussion, we assume that an algorithm will not process a job after missing its deadline, and whenever we say that a job is completed, it always means to be completed by the deadline.

Speed and power. At any time, a processor is in either *sleep* state or *awake* state. When a processor is in sleep state, the speed is zero and the power is zero. When a processor is in awake state, it can vary its speed in $[0, T]$, where T is the fixed maximum speed; the rate of energy consumption is modeled as $s^\alpha + \sigma$, where s is the current speed, and $\alpha > 1$ and $\sigma > 0$ are constants. We call s^α the *dynamic power* and σ the *static power*. State transition requires energy; without loss of generality, we assume a transition from the sleep state to the awake state requires an amount of $\omega > 0$ of energy, and the reverse takes zero energy. We assume state transition takes no time. It is useful to distinguish two types of awake state: *idle* state with zero speed, and *working* state with positive speed. Note that idle state consumes energy. Initially the processor is in sleep state.

Consider any schedule. The energy usage E is divided into three parts: W denotes the *wake-up energy* due to wake-up transitions (the total number of wake-ups multiplied by ω), E_i is the *idling energy* (static power consumption in the idle state), and E_w is the *working energy* (static and dynamic power consumption in the working state).

Deadlines, density and feasibility. Consider a sequence of jobs with deadlines. We want to maximize the throughput, i.e., the sum of $w(J)$ over all jobs J that can be completed by its deadline $d(J)$. At time t, let $w(t, t')$, for any $t' > t$, be the remaining work of jobs arriving at or before t and with deadlines in $(t, t']$. We define the *density* $\rho(t, t')$ to be $w(t, t')/(t' - t)$, and the *highest density* ρ to be $\max_{t'>t} \rho(t, t')$. Intuitively, ρ is a lower bound on the average speed required to complete all jobs by deadlines. The algorithm OA always uses the speed ρ of the current time to run the job with the earliest deadline (EDF strategy). For any schedule $\mathcal{S}$, we use $\mathcal{S}(t)$ to denote the speed used at time t.

In the bounded speed model, at any time t, we say that a set of jobs is *feasible* if at time t, all remaining work of these jobs can be completed by their deadlines by running the processor at speed T starting at t (and using, say, the EDF strategy to select the current job for running).

Critical speed. If a job J is run to completion using speed s, the energy usage is $P(s)w(J)/s$, where $P(s) = s^\alpha + \sigma$. When deadline is not a constraint, the energy usage is minimized at the speed s satisfying $P(s) = s \times P'(s)$, i.e., $s = (\sigma/(\alpha-1))^{1/\alpha}$. We call this speed the *critical speed* s_{crit}. In the following, we assume $s_{\text{crit}} \le T$. The case when $s_{\text{crit}} > T$ is not interesting as the competitive ratio shown in this paper can be easily achieved by running the processor at speed T whenever it is awake (this will minimize the working energy as defined above).

3 SOA Sleep-aware Optimal Available

In this section, we describe the online algorithm SOA, which dynamically determines the processor speed, as well as when to sleep and wake up. In the infinite speed model, SOA (coupled with EDF) can complete all jobs. In the bounded speed model, we cap the speed of SOA at the maximum speed T; this is sufficient to support the job selection algorithm Slow-D [4] to give a 4-competitive algorithm for throughput. The difficult part is on the analysis of the energy of SOA, especially in the bounded speed model, which is detailed in Section 4.

3.1 SOA in the infinite speed model

Without sleep state, a scheduler, to save energy, would run a job as slow as its deadline allows. When sleep state is allowed, we want to create longer idle periods and let the processor sleep more. One possible way is to postpone job execution during idle time, and work faster later. This idea is first used by Procrastinator [17], which relies on extra speed to catch up with the delayed jobs. To save energy, as well as to make the above idea work in the bounded speed model, we observe that the extra speed is indeed not necessary and SOA only mildly increases the maximum speed and energy usage even during the "busy" periods.

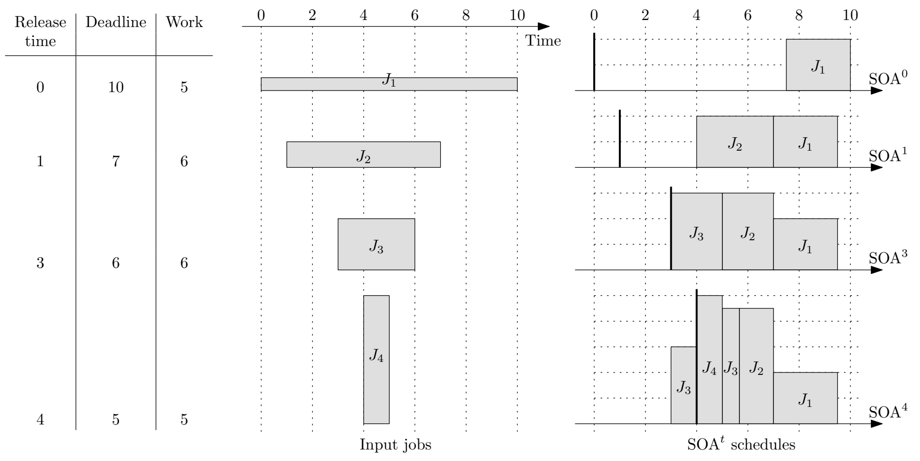
At any time t, SOA determines the processor speed by making reference to the currently highest density ρ (i.e., $\max_{t'>t} w(t,t')/(t'-t)$). SOA prefers to extend an idle (or sleep) period when the highest density ρ is small. If ρ is small after the processor has idled for ω/σ time units, then the processor switches to the sleep state. Later when ρ is big enough, SOA is forced to set a high speed to avoid missing any deadline. On the other hand, in the working state, it is simple to determine the next transition. The processor keeps on working as long as there are unfinished jobs; otherwise switches to the idle state. Note that in the infinite speed model, we target completing all jobs and we use EDF to pick the current job for execution. Details are given in Algorithm 1.

For simplicity, the algorithm is written assuming the scheduling algorithm is continuously running. In practice, we can rewrite the algorithm such that the execution is driven by discrete events like job release, job completion and wake-up. Recall that s_{crit} denotes the critical speed.

Algorithm 1 SOA, Sleep-aware Optimal Available

```
Consider any time $t$. Let $\rho$ be the highest density at time $t$.
In working state: If $\rho > 0$, keep working on the job with the earliest deadline (EDF) at speed $\max\{\rho, s_{\text{crit}}\}$; else (i.e., $\rho = 0$) switch to idle state.
In idle state: Let $t' \le t$ be the last time in the working state ($t' = 0$ if undefined). If $\rho \ge s_{\text{crit}}$, switch to working state; else if $(t-t')\sigma = \omega$, switch to sleep state.
In sleep state: If $\rho \ge s_{\text{crit}}$, switch to working state.
```

The following theorem is immediate from the definition of SOA.



Release time	Deadline	Work
0	10	5
1	7	6
3	6	6
4	5	5

Figure 1: An example showing how SOA^t changes as jobs are released. The jobs are shown on the left as rectangles. The release time and deadline of a job are the left and right edge of the rectangle, and the size is represented by its area. SOA^t are shown on the right, the thick vertical line is the current time. Here $s_{\text{crit}} = 2$.

Theorem 1. *In the infinite speed model,* SOA *(coupled with EDF) can complete all jobs.*

As jobs arrive over time, the speed determined by SOA, as a function of time, can be quite complex. Yet the structure is much simpler when no more jobs arrive. At any time t, let SOA^t denote the *planned schedule* of SOA, i.e., the schedule SOA would produce if no more jobs arrive after time t. An example is depicted in Figure 1. By definition, SOA^t satisfies the following properties.

Property 2. *Consider the period of time starting from t.*

(i) SOA^t *increases its speed at most once. This is possible only if* SOA^t *is in idle or sleep state at t, and increases the speed from 0 to s_{crit} at some time after t.*

(ii) SOA^t *decreases its speed only at deadlines of jobs, or when all jobs are completed.*

An implication of Property 2 (i) and (ii) is that after t, SOA^t is either a falling staircase-like function (see SOA^3 and SOA^4 in Figure 1) or a function with a single step of value s_{crit} (see SOA^0 and SOA^1 in Figure 1).

3.2 Slow-D(SOA) in the bounded speed model

In the bounded speed model, it may not be possible to finish all jobs. Two strategies are needed, one for selecting jobs and one for determining the speed. To determine the speed, we simply cap the speed at T. That is, at any time t, we keep a simulated SOA schedule and the processor runs at the speed $\min\{\text{SOA}(t), T\}$. We denote this speed function as SOAT(t). In [4], which assumes no sleep state, the job selection algorithm Slow-D together with the speed function OA (capped at T) is proven to be 4-competitive for throughput. With sleep state, we substitute OA by SOA and call the resulting algorithm Slow-D(SOA). We will describe the details and analyze the throughput of Slow-D(SOA) in Section 5. The proof that Slow-D(SOA) remains 4-competitive for throughput is similar to that in [4].

The analysis of the energy usage of Slow-D(SOA) is more challenging and we give the details in Section 4. Note that we will analyze the energy usage of the speed function SOAT(t), without knowing the details of Slow-D. To ease our analysis we define SOAT to be an imaginary schedule which at time t, processes the same job chosen by SOA (instead of Slow-D) at the speed SOAT(t). Jobs may not be scheduled to completion in the schedule SOAT.

4 Energy consumption of Slow-D(SOA)

In this section, we show that SOAT, and thus Slow-D(SOA), is $(\alpha^\alpha + \alpha^2 4^\alpha + 2)$-competitive for energy in the bounded speed model, against the optimal offline algorithm OPT that achieves the maximum throughput; and SOA is $\max\{\alpha^\alpha + 2, 4\}$-competitive for energy in the infinite speed model (Theorem 14 in Section 4.2.5).

Recall that the energy usage E is divided into three types: the *wake-up energy* W, the *idle energy* E_i due to the static power σ during idle state, and the *working energy* E_w due to static and dynamic power during working state (we use a superscript $*$ for OPT, e.g., E^*). In Section 4.1, we show that the idle energy and wake-up energy contribute only a small factor when compared with the energy used by the optimal offline algorithm. In Section 4.2, we give the detailed analysis of the working energy using potential functions.

4.1 Idle energy and wake-up energy

As observed by Irani et al. [17], the idle energy and wake-up energy of PROCRASTINATOR contributes only a small factor when compared with the energy used by OPT. Note that their proof assumes that $\omega = 1$. In this section, we show that SOA has a similar property for arbitrary ω (Lemma 4) and the corresponding factor is smaller than that of PROCRASTINATOR. We define an *idle interval* and a *sleep interval* as a maximal time period when the processor is in idle state and sleep state, respectively. We first show that in a schedule following the idle and sleep strategy as SOA, there cannot be many idle intervals that overlap with a sleep interval of OPT.

Lemma 3. *Consider a schedule using the same idle and sleep strategy as* SOA. **(i)** *Suppose $\mathcal{I}$ is an idle interval lying completely in a sleep interval $\mathcal{S}$ of* OPT*. Then there is no idle interval after $\mathcal{I}$ overlapping with $\mathcal{S}$.* **(ii)** *There is no idle interval of* SOA *overlapping with the first sleep interval of* OPT*.*

Proof. **(i)** Consider any two consecutive idle intervals $\mathcal{I}_1 = [x_1, y_1)$ and $\mathcal{I}_2 = [x_2, y_2)$ in SOA, and the first job J run after $\mathcal{I}_1$ (with earliest deadline). Since SOA switches to idle state only when $\rho = 0$, any job run after $\mathcal{I}_1$, including J, must arrive after x_1. At time y_1, the processor speed is set as ρ implying that the processor will keep busy at least until $d(J)$. In other words, $x_2 \geq d(J)$. If $\mathcal{I}_1$ lies completely in a sleep interval $\mathcal{S}$ of OPT, then OPT cannot be sleeping entirely during $[x_1, x_2)$; otherwise, we can improve OPT by completing J as well to achieve higher throughput, contradicting the optimality. Therefore, $\mathcal{I}_2$ does not overlap with $\mathcal{S}$.

(ii) Since the processor is in sleep state initially for both SOA and OPT, the statement follows for the same argument. □

With the above lemma, we now show that it suffices to focus on the working energy.

Lemma 4. *If the working energy of an algorithm using the same idle and sleep strategy as* SOA *is at most c times that of* OPT*, then its total energy is at most* $\max\{c+2, 4\}$ *times that of* OPT*.*

Proof. Suppose in OPT, there are m sleep intervals and a total of x time units of idle state, i.e., $E^* = E^*_{\mathrm{w}} + (m-1)\omega + x\sigma$ (note that the processor is in sleep state initially and finally). Consider the cost of SOA. To account for W, each idle interval with transition to sleep state pays for the wake-up cost ω of the previous wake-up. We bound the cost of an idle interval in two ways. Firstly, the cost is at most 2ω: ω to keep the processor awake for at most ω/σ time units, and ω for the wake-up (in case it transits to sleep state eventually). Secondly, the cost of any idle interval of length y_i is at most $2y_i\sigma$: $y_i\sigma$ if $y_i < \omega/\sigma$, and $y_i\sigma + \omega = 2y_i\sigma$ if $y_i = \omega/\sigma$.

To relate to OPT, we distinguish two types of idle intervals: intervals that overlap with some sleep interval in OPT and intervals that do not. The number of the first type of intervals is at most $2(m-1)$ (by Lemma 3) costing a total of at most $4(m-1)\omega$. Suppose y is the total length of the second type of intervals. During these intervals, OPT must have spent at least $y\sigma$ to keep the processor awake, i.e., $E^*_{\mathrm{w}} + x\sigma \geq y\sigma$, where x is the total length of idle periods in OPT, so the total cost of these intervals is at most $2(E^*_{\mathrm{w}} + x\sigma)$. Considering all idle intervals, $W + E_{\mathrm{i}} \leq 4(m-1)\omega + 2E^*_{\mathrm{w}} + 2x\sigma$.

Together with $E_{\mathrm{w}} \leq cE^*_{\mathrm{w}}$, we have $E \leq (c+2)E^*_{\mathrm{w}} + 4(m-1)\omega + 4x\sigma \leq \max\{c+2, 4\}E^*$. □

4.2 Working energy and potential analysis

We now compare the working energy E_{w} of SOAT and E^*_{w} of OPT (recall that SOAT is the imaginary schedule which at time t, processes the same job as SOA at the speed $\mathrm{SOAT}(t) = \min\{\mathrm{SOA}(t), T\}$). The analysis follows the framework of the amortization and potential analysis of OAT [12] (OA capped at T).

At any time t, let $E_{\mathrm{w}}(t)$ and $E^*_{\mathrm{w}}(t)$ be the corresponding value of E_{w} and E^*_{w} incurred up to t. We will define two potential functions $\phi(t)$ and $\beta(t)$, which are functions of time satisfying the following conditions: (i) $\phi(0) = \beta(0) = 0$ and at a time t_{e} after all job deadlines, $\phi(t_{\mathrm{e}}) = 0$ and $\beta(t_{\mathrm{e}})$ is small; and (ii) the following inequality holds at any time t:

$$E_{\mathrm{w}}(t) + \phi(t) - \beta(t) \leq \alpha^\alpha E^*_{\mathrm{w}}(t) \ . \tag{1}$$

We can then apply the inequality at t_{e} to obtain $E_{\mathrm{w}} \leq \alpha^\alpha E^*_{\mathrm{w}} + \beta(t_{\mathrm{e}})$.

The two potential functions $\phi(t)$ and $\beta(t)$ to be defined in Section 4.2.1 are continuous functions except at some discrete times. To prove Inequality (1), we follow the framework in [12] to consider how the potential functions change in two different scenarios, namely, at the arrival time of jobs and at any other time. For the former, we show that $\phi - \beta$ can never increase (Lemma 12 in Section 4.2.4). For the latter, we show that the rate of change of various functions satisfies the following bound (Lemma 7 in Section 4.2.2)

$$\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \frac{\mathrm{d}\beta(t)}{\mathrm{d}t} \leq \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} \ . \tag{2}$$

Then in Section 4.2.5, we consider the simple boundary cases and complete the proof of Inequality (1) and the overall energy competitive analysis of SOAT and Slow-D(SOA).

4.2.1 Potential functions

Below we show how to extend the analysis in [12] to the setting with sleep state. Roughly speaking, $\phi(t)$ denotes the difference of the unfinished work of SOAT and OPT "weighted" by a special function to make it compatible with energy, and $\beta(t)$ denotes the weighted amount of work that SOAT has processed for jobs that OPT does not complete (note that OPT may complete only a subset of jobs). The potential functions ϕ and β give different weights to work.

For β, work is weighted by a simple multiplier $\alpha^2 T^{\alpha-1}$. For ϕ, the weight is more complicated and not uniform; it is based on a notion called critical intervals, which is used in [12]. To handle the presence of idle and sleep periods, we use a new definition of critical intervals. To define ϕ and β, we first define two types of jobs depending on whether OPT completes the jobs.

Type-0 and type-1 job. Note that OPT does not aim to complete all jobs. For the sake of analysis, we say that a job is *type-1* if OPT completes the job, and *type-0* otherwise. We assume OPT does not run a type-0 job at any time. The work due to type-0 jobs is called type-0 work and similarly type-1 work for type-1 jobs. Note that the online algorithm does not know such a classification.

Neither SOAT nor OPT intend to complete all jobs. At any time t, the unfinished work no longer means the unfinished work of all jobs. For OPT, we confine the unfinished work to type-1 jobs only. For SOAT, the unfinished work of SOAT at t refers to the amount of work to be done after t in SOAT^t, the planned SOAT schedule calculated at t.

As mentioned earlier, $\phi(t)$ captures the difference in progress of SOAT and OPT while $\beta(t)$ captures the work SOAT would process for type-0 jobs. Since SOAT may schedule jobs including type-0 jobs, this makes it difficult to relate the energy and remaining work of SOAT and OPT. For example, when a type-0 job arrives, the unfinished work of OPT remains the same, yet this job will boost the unfinished work of SOAT. This problem is resolved by the potential function β to discount the effect of type-0 jobs in the amortization analysis.

Potential function $\boldsymbol{\beta(t)}$. We first define $\beta(t)$ to be $\alpha^2 T^{\alpha-1}$ times the type-0 work processed in SOAT^t (this includes the type-0 work that has already been processed and the unfinished type-0 work yet to process in the planned schedule). It has been shown in [12] that the amount of type-0 work that any algorithm (including OAT and SOAT) can process is at most 4 times of the total work OPT can complete (or equivalently, all type-1 work). This leads to the following upper bound on $\beta(t_\text{e})$ by the working energy of OPT (recall that t_e is the time after all job deadlines).

Lemma 5. [12] *$\beta(t_\text{e}) \le \alpha^2 4^\alpha E^*_\text{w}(t_\text{e})$.*

Potential function $\boldsymbol{\phi(t)}$. The value $\phi(t)$ is essentially a weighted sum of the amount of unfinished work of each job J in SOAT^t and in OPT. The weight is based on the notion of critical intervals (which was first introduced in [7] and refined in [12] for analyzing OA and OAT, respectively). Previously, critical intervals were defined based on the property that the planned schedule is a falling staircase-like function. Yet, to handle sleep state, the definition of critical intervals becomes not straightforward due to Property 2 (i). Below we give a new definition of critical intervals and unfinished work. Consider the current time t and the planned SOAT schedule calculated at t, SOAT^t.

- We define the function $\hat{\rho}(t') = \max\{\text{SOAT}^t(t'), s_\text{crit}\}$ for $t' > t$. Note that $\hat{\rho}(t')$ is exactly $\text{SOAT}^t(t')$ unless $\text{SOAT}^t(t') = 0$. By Property 2 (i) and (ii), after t, SOAT^t is either a falling staircase-like function or one with a single step of value s_crit. Taking the maximum with s_crit ensures that $\hat{\rho}$ is always a falling staircase-like function.

- Define a sequence of times as follows: Let $c_0 = t$. For $i \ge 1$, define c_i such that $(c_{i-1}, c_i]$ is the maximal time period where $\hat{\rho}$ does not change in value. Each interval $(c_{i-1}, c_i]$ is called a *critical interval*. We use $\hat{\rho}_i$ to denote the unchanged value of $\hat{\rho}$ in the interval $(c_{i-1}, c_i]$. Because of the staircase property, we have $\hat{\rho}_1 > \hat{\rho}_2 > \hat{\rho}_3 > \dots$

- By definition, SOAT schedules jobs in accordance with SOA, but using a speed capped at T. Unlike SOA, SOAT does not aim to complete every job. For any currently available job J, if the deadline $d(J)$ is in the critical interval $(c_{i-1}, c_i]$ and SOA plans to schedule J for x time units, then SOAT^t is to process $\hat{\rho}_i\, x$ units of work for J after time t. We define the *unfinished work* of J under SOAT to be $\hat{\rho}_i\, x$.

The potential function $\phi(t)$ weighs the unfinished work of the currently available jobs according to which critical intervals their deadlines fall into. At time t, with respect to the critical interval $(c_{i-1}, c_i]$,

- let $w_{\mathrm{a}}(i)$ be the amount of unfinished work under SOAT for currently available jobs with deadlines in $(c_{i-1}, c_i]$;
- let $w_{\mathrm{o}}(i)$ be the amount of unfinished type-1 work under OPT for currently available jobs with deadlines in $(c_{i-1}, c_i]$.

We apply the weight of $\alpha\hat{\rho}_i^{\alpha-1}$ to the unfinished work with deadlines in $(c_{i-1}, c_i]$ and define

$$\phi(t) = \alpha \textstyle\sum_{i \geq 1} \hat{\rho}_i^{\alpha-1}(w_{\mathrm{a}}(i) - \alpha w_{\mathrm{o}}(i)) \ .$$

The rest of this section is devoted to proving Inequality (1). As mentioned earlier, we consider the potential functions change between job arrivals and when jobs arrive.

4.2.2 Change of potential between job arrivals

Before we consider the change of the potential when no job is being released, we first state a fact about the critical speed s_{crit}.

Fact 6. [17] *For any speed s, we have $\frac{s_{\mathrm{crit}}^\alpha + \sigma}{s_{\mathrm{crit}}} \leq \frac{s^\alpha + \sigma}{s}$.*

Let t_0 be the current time. Assume that no job arrives at time t_0. Let s_{a} and s_{o} denote the speed of SOAT and OPT at t_0, respectively. We observe the changes of various components of Inequality (1) as follows.

- The rate of change of working energy $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t}$ depends on s_{a} and s_{o}, respectively. The change in working energy is $s^\alpha + \sigma$ when running at speed s, and 0 when idling.
- As no job arrives at t_0, the planned SOAT schedule does not change at t_0, implying that the amount of type-0 work does not change, so $\beta(t)$ does not change, i.e., $\frac{\mathrm{d}\beta(t)}{\mathrm{d}t} = 0$ at t_0.
- For $\phi(t)$, this also implies that the weight of a job does not change at t_0.

  If $s_{\mathrm{a}} > 0$ at time t_0, SOAT is going to process a job with deadline in the first critical interval (as defined at time t_0). In other words, $w_{\mathrm{a}}(1)$ is decreasing at the rate of s_{a}, $\hat{\rho}_1 = s_{\mathrm{a}}$, and the weight of the running job is $\alpha s_{\mathrm{a}}^{\alpha-1}$.

  If $s_{\mathrm{o}} > 0$ at time t_0, the deadline of the job to be processed by OPT is not necessarily in the first critical interval. Suppose it is in the k-th critical interval for some $k \geq 1$. Then $w_{\mathrm{o}}(k)$ is decreasing at the rate of s_{o} and the absolute value of the weight of the running job is $\alpha^2\hat{\rho}_k^{\alpha-1} \leq \alpha^2\hat{\rho}_1^{\alpha-1}$ because the function that determines the weights, $\hat{\rho}$, is a decreasing staircase-like function between job arrivals.

The following lemma shows that Inequality (1) cannot start being violated at t_0.

Lemma 7. *Let t_0 be a time when no job arrives. Then at t_0, $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \frac{\mathrm{d}\beta(t)}{\mathrm{d}t} \le \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t}$.*

Proof. As discussed before, $\frac{\mathrm{d}\beta(t)}{\mathrm{d}t} = 0$ at t_0. We thus need to show that $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} \le 0$. We distinguish 4 cases, depending on whether SOAT and OPT are working or not.

Case 1: $s_{\mathrm{a}} > 0$, $s_{\mathrm{o}} > 0$. We have $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} = s_{\mathrm{a}}^\alpha + \sigma$ and $\frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} = s_{\mathrm{o}}^\alpha + \sigma$. For $\phi(t)$, we have $\hat{\rho}_1 = s_{\mathrm{a}}$. The weight of the running job in SOAT is $\alpha s_{\mathrm{a}}^{\alpha-1}$, while the absolute value of the weight of the running job in OPT is no more than $\alpha^2 s_{\mathrm{a}}^{\alpha-1}$. So

$$\frac{\mathrm{d}\phi(t)}{\mathrm{d}t} \le -\alpha s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{a}} + \alpha^2 s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{o}},$$

and

$$\begin{aligned} \frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} &\le (s_{\mathrm{a}}^\alpha + \sigma) - \alpha s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{a}} + \alpha^2 s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{o}} - \alpha^\alpha (s_{\mathrm{o}}^\alpha + \sigma) \\ &\le s_{\mathrm{a}}^\alpha - \alpha s_{\mathrm{a}}^\alpha + \alpha^2 s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{o}} - \alpha^\alpha s_{\mathrm{o}}^\alpha, \end{aligned}$$

which can be written as $s_{\mathrm{o}}^\alpha f(z)$ where $f(z) = (1-\alpha)z^\alpha + \alpha^2 z^{\alpha-1} - \alpha^\alpha$, and $z = s_{\mathrm{a}}/s_{\mathrm{o}}$. By simple differentiation, $f(z)$ is maximized at $f(\alpha) = 0$, so the above expression is never positive.

Case 2: $s_{\mathrm{a}} > 0$, $s_{\mathrm{o}} = 0$. We have $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} = s_{\mathrm{a}}^\alpha + \sigma$ and $\frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} = 0$. For $\phi(t)$, the weight of the running job in SOAT is $\alpha s_{\mathrm{a}}^{\alpha-1}$, while there is no change in the terms involving OPT. So

$$\frac{\mathrm{d}\phi(t)}{\mathrm{d}t} = -\alpha s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{a}},$$

and

$$\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} = (s_{\mathrm{a}}^\alpha + \sigma) - \alpha s_{\mathrm{a}}^{\alpha-1} s_{\mathrm{a}} = \sigma - (\alpha-1)s_{\mathrm{a}}^\alpha.$$

Note that $s_{\mathrm{a}} \ge s_{\mathrm{crit}}$. Since $s_{\mathrm{crit}} = (\sigma/(\alpha-1))^{1/\alpha}$, $s_{\mathrm{a}}^\alpha \ge \sigma/(\alpha-1)$, and

$$\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} \le \sigma - \sigma = 0.$$

Case 3: $s_{\mathrm{a}} = 0$, $s_{\mathrm{o}} > 0$. We have $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} = 0$ and $\frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} = s_{\mathrm{o}}^\alpha + \sigma$. With Fact 6 and $s_{\mathrm{crit}} = (\sigma/(\alpha-1))^{1/\alpha}$, this implies $\frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} \ge s_{\mathrm{o}} \frac{s_{\mathrm{crit}}^\alpha + \sigma}{s_{\mathrm{crit}}} = \alpha s_{\mathrm{crit}}^{\alpha-1} s_{\mathrm{o}}$. For $\phi(t)$, we have $\hat{\rho}_1 = s_{\mathrm{crit}}$ and $w_{\mathrm{a}}(1)$ is not changing. There is no change in the terms involving SOAT, while the absolute value of the weight of the running job in OPT is at most $\alpha^2 s_{\mathrm{crit}}^{\alpha-1}$. So

$$\frac{\mathrm{d}\phi(t)}{\mathrm{d}t} \le \alpha^2 s_{\mathrm{crit}}^{\alpha-1} s_{\mathrm{o}},$$

and

$$\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} \le \alpha^2 s_{\mathrm{crit}}^{\alpha-1} s_{\mathrm{o}} - \alpha^\alpha \alpha s_{\mathrm{crit}}^{\alpha-1} s_{\mathrm{o}} \le 0.$$

Case 4: $s_{\mathrm{a}} = 0$, $s_{\mathrm{o}} = 0$. In this case $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t}$, $\frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t}$ and $\frac{\mathrm{d}\phi(t)}{\mathrm{d}t}$ are all 0, so $\frac{\mathrm{d}E_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^*_{\mathrm{w}}(t)}{\mathrm{d}t} = 0$. □

4.2.3 Change of potential when a job arrives - simple cases

Let t_0 be the current time. Assume that a job J arrives at time t_0. The arrival of J immediately changes the schedules of SOA and SOAT, the boundaries of critical intervals, unfinished work of SOAT and OPT, as well as β and ϕ. Yet, similar to [7,12], we observe that the resulting change would be the same if the single job J is replaced by multiple jobs with the same arrival time and deadline, and total work $w(J)$. Then we can simulate the changes due to J by a sequence of such jobs so that each of them leads to one of the following three possible changes to the planned SOAT schedule at t_0. In Section 4.2.4, we show how to simulate a job in general by a sequence of such jobs. In this section, we will show that the changes of these simple cases are simple enough to analyze.

Simple Case A. One interval I of uniform speed $\geq s_{\text{crit}}$ increases the speed uniformly by $w(J)/|I|$.

Simple Case B. The schedule does not change at all, in which case SOAT runs at speed T at $d(J)$.

Simple Case C. The interval of speed s_{crit} is lengthened by $w(J)/s_{\text{crit}}$ and possibly moved forward.

We now assume that the change due to job J is among one of the above simple cases.

Denote the change in β and ϕ due to the arrival of J at t_0 as $\Delta\beta$ and $\Delta\phi$, respectively. To show that Inequality (1) is not violated by such event, we show that $\Delta\phi - \Delta\beta \leq 0$ for each simple case. Just before J arrives, denote the critical intervals as $C_i = (c_{i-1}, c_i]$ and the unfinished work under SOAT and OPT with deadlines in C_i as $w_{\text{a}}(i)$ and $w_{\text{o}}(i)$, respectively. Suppose that $d(J)$ falls into C_x. Note that J can be a type-1 job (i.e., OPT schedules J) or a type-0 job (i.e., OPT does not schedule J).

Lemma 8. *Suppose a job J arrives at t_0 leading to change of Simple Case A.* **(i)** *If J is a type-1 job then $\Delta\phi \leq 0$ and $\Delta\beta = 0$.* **(ii)** *If J is a type-0 job then $\Delta\phi \leq \alpha^2 T^{\alpha-1} w(J)$ and $\Delta\beta \geq \alpha^2 T^{\alpha-1} w(J)$.*

Proof. In this case, one interval with uniform speed before adding the job increases in speed uniformly. Suppose this interval is $(d_1, d_2]$ and the value of the $\hat{\rho}$ function increases from $\hat{\rho}_{\text{a}}$ to $\hat{\rho}_{\text{b}}$. The interval may be a critical interval in the planned SOAT schedule or may possibly be split from some critical interval. We denote the remaining work with deadlines in $(d_1, d_2]$ under SOAT and OPT by w_{a} and w_{o}, respectively. Then, $\hat{\rho}_{\text{a}} = \frac{w_{\text{a}}}{d_2 - d_1}$ and $\hat{\rho}_{\text{b}} = \frac{w_{\text{a}} + w(J)}{d_2 - d_1}$. Note that $\hat{\rho}_{\text{b}} \leq T$. The scheduling of jobs other than J is not affected in SOAT.

(i) Suppose J is a type-1 job. The amount of type-0 work processed by SOAT does not change and thus $\Delta\beta = 0$. Consider $\Delta\phi$. J increases the unfinished work of OPT by exactly $w(J)$. Since $\hat{\rho}_{\text{b}} \leq T$, the increase of the unfinished work of SOAT (during $(d_1, d_2]$) is also $w(J)$. The term in ϕ for unfinished work of SOAT changes from $\alpha\hat{\rho}_{\text{a}}^{\alpha-1} w_{\text{a}}$ to $\alpha\hat{\rho}_{\text{b}}^{\alpha-1}(w_{\text{a}} + w(J))$; the term for unfinished work of OPT changes from $-\alpha^2\hat{\rho}_{\text{a}}^{\alpha-1} w_{\text{o}}$ to $-\alpha^2\hat{\rho}_{\text{b}}^{\alpha-1}(w_{\text{o}} + w(J))$. Hence,

$$\begin{aligned}
\Delta\phi &= \alpha\hat{\rho}_{\text{b}}^{\alpha-1}\Big((w_{\text{a}} + w(J)) - \alpha(w_{\text{o}} + w(J))\Big) - \alpha\hat{\rho}_{\text{a}}^{\alpha-1}\Big(w_{\text{a}} - \alpha w_{\text{o}}\Big) \\
&= \frac{\alpha}{(d_2 - d_1)^{\alpha-1}}\Big[(w_{\text{a}} + w(J))^{\alpha-1}\Big((w_{\text{a}} + w(J)) - \alpha(w_{\text{o}} + w(J))\Big) - w_{\text{a}}^{\alpha-1}\Big(w_{\text{a}} - \alpha w_{\text{o}}\Big)\Big].
\end{aligned}$$

Bansal et al. [7] has shown that a general form of the above expression is non-positive:

For any $q, r, z \geq 0$ and $\alpha \geq 1$, $(q+z)^{\alpha-1}(q + z - \alpha(r+z)) - q^{\alpha-1}(q - \alpha r) \leq 0$.

Thus, substituting $q = w_{\text{a}}$, $r = w_{\text{o}}$, and $z = w(J)$, we conclude that $\Delta\phi \leq 0$.

(ii) Suppose J is a type-0 job. J increases the unfinished type-0 work of SOAT, the increase is exactly $w(J)$ since $\hat{\rho} \leq T$, i.e., $\Delta\beta = \alpha^2 T^{\alpha-1} w(J)$. Consider $\Delta\phi$. Since J is a type-0 job, it does not increase the unfinished work of OPT. So the term in ϕ for OPT changes from $-\alpha^2\hat{\rho}_{\mathrm{a}}^{\alpha-1}w_{\mathrm{o}}$ to $-\alpha^2\hat{\rho}_{\mathrm{b}}^{\alpha-1}w_{\mathrm{o}}$.

$$\begin{aligned}\Delta\phi &= \alpha\hat{\rho}_{\mathrm{b}}^{\alpha-1}\Big(w_{\mathrm{a}} + w(J) - \alpha w_{\mathrm{o}}\Big) - \alpha\hat{\rho}_{\mathrm{a}}^{\alpha-1}\Big(w_{\mathrm{a}} - \alpha w_{\mathrm{o}}\Big)\\ &= \alpha\hat{\rho}_{\mathrm{b}}^{\alpha-1}\Big((w_{\mathrm{a}} + w(J)) - \alpha(w_{\mathrm{o}} + w(J))\Big) - \alpha\hat{\rho}_{\mathrm{a}}^{\alpha-1}\Big(w_{\mathrm{a}} - \alpha w_{\mathrm{o}}\Big) + \alpha^2\hat{\rho}_{\mathrm{b}}^{\alpha-1}w(J)\\ &\le \alpha^2\hat{\rho}_{\mathrm{b}}^{\alpha-1}w(J) \qquad \text{(see (i))}\\ &\le \alpha^2T^{\alpha-1}w(J). \qquad \square\end{aligned}$$

Lemma 9. *Suppose a job J arrives at t_0 leading to change of Simple Case B.* **(i)** *If J is a type-1 job then $\Delta\phi = -\alpha^2T^{\alpha-1}w(J)$ and $\Delta\beta \ge -\alpha^2T^{\alpha-1}w(J)$.* **(ii)** *If J is a type-0 job then $\Delta\phi = 0$ and $\Delta\beta \ge 0$.*

Proof. In this case, the planned SOAT schedule does not change at all at t_0, and the critical interval $C_x = (c_{x-1}, c_x]$ in which $d(J)$ lies has $\hat{\rho}_x = T$. We first consider $\Delta\phi$. There is no change in the values of $\hat{\rho}_x$, the weights in $\phi(t)$, nor the amount of work to be processed after t_0 under SOAT (i.e., $w_{\mathrm{a}}(x)$), thus the change is only due to $w_{\mathrm{o}}(x)$, which increases by $w(J)$ if J is type-1. So $\Delta\phi = -\alpha^2T^{\alpha-1}w(J)$ if J is type-1, and $\Delta\phi = 0$ if J is type-0.

Let us consider $\Delta\beta$. Recall that SOAT is an imaginary schedule which at time t, processes the same job as SOA at the speed $\mathrm{SOAT}(t) = \min\{\mathrm{SOA}(t), T\}$. When J arrives, SOA and hence SOAT will schedule J in C_x and must reduce the time for processing existing jobs in C_x so as to make room for J. Since SOAT cannot increase the speed beyond T during C_x, the work of some existing jobs to be processed by SOAT during C_x has to be reduced. Some of the work reduced may be type-0; yet the reduction of type-0 work is at most the work SOAT commits to J. If J is type-1, β may decrease by at most $\alpha^2T^{\alpha-1}w(J)$, i.e., $\Delta\beta \ge -\alpha^2T^{\alpha-1}w(J)$. If J is type-0, J itself contributes to the type-0 work to compensate for any possible decrease of the existing jobs, i.e., $\Delta\beta \ge 0$. $\square$

Lemma 10. *Suppose a job J arrives at t_0 leading to change of Simple Case C.* **(i)** *If J is a type-1 job then $\Delta\phi < 0$ and $\Delta\beta = 0$.* **(ii)** *If J is a type-0 job then $\Delta\phi \le \alpha^2T^{\alpha-1}w(J)$ and $\Delta\beta = \alpha^2T^{\alpha-1}w(J)$.*

Proof. In this case, the interval of speed s_{crit} in the planned SOAT schedule is lengthened by $w(J)/s_{\mathrm{crit}}$ and possibly moved forward. There is no change in the value of $\hat{\rho}$ but the unfinished work under SOAT increases by $w(J)$. The scheduling of other jobs are not affected. If J is a type-1 job, the amount of type-0 work under SOAT does not change and thus $\Delta\beta = 0$. The amount of unfinished work under both SOAT and OPT increases by $w(J)$, so $\Delta\phi = \alpha s_{\mathrm{crit}}^{\alpha-1}w(J) - \alpha^2 s_{\mathrm{crit}}^{\alpha-1}w(J) < 0$.

If J is a type-0 job, the amount of type-0 work under SOAT increases by $w(J)$ and $\Delta\beta = \alpha^2T^{\alpha-1}w(J)$. The amount of unfinished work under OPT does not change, so $\Delta\phi = \alpha s_{\mathrm{crit}}^{\alpha-1}w(J) \le \alpha^2T^{\alpha-1}w(J)$. $\square$

The corollary below follows immediately from Lemmas 8, 9, and 10.

Corollary 11. *When a job arrives at t_0 leading to a change of one of the Simple Cases, we have $\Delta\phi - \Delta\beta \le 0$.*

4.2.4 Change of potential when a job arrives - the general case

In general, when a job J is released at time t, the schedule of SOA and SOAT may change radically. Nevertheless, similar to an observation in [7], we can consider the change as a sequence of smaller changes. Roughly speaking, we can imagine the size of J as increasing from 0 to $w(J)$ while one of the previous simple cases holds up to a certain size, say u. Then we simulate the arrival of J by the arrival of two jobs J_1 and J_2 where J_1 has size u and J_2 has size $w(J) - u$. Details are given below. Consider SOAT^t, the planned SOAT schedule just before J arrives. Let $C_i = (c_{i-1}, c_i]$ denote the i-th critical interval with associated value $\hat{\rho}_i$. Note that J increases the density $\rho(t, t')$ for $t < d(J) \leq t'$ but not necessarily $\text{SOAT}^t(t')$ or $\hat{\rho}(t')$. We consider two cases depending on whether SOAT^t is working at t.

Case 1: SOAT^t is working at t. To define the value $u \leq w(J)$, we further consider 3 cases.

(a) If $\text{SOAT}^t(d(J)) = T$, which implies SOAT^t runs at speed T from t to $d(J)$, then J would not change the planned schedule SOAT^t and we let $u = w(J)$. (This leads to Simple Case B.)

(b) If $s_{\text{crit}} < \text{SOAT}^t(d(J)) < T$, suppose $d(J)$ lies in some critical interval $C_x = (c_{x-1}, c_x]$. By the definition of critical interval, $\hat{\rho}_{x-1} > \hat{\rho}_x$. Let $u \leq w(J)$ be the smallest size such that one of the following events occurs.

- The density $\rho(c_{x-1}, c_x)$ increases to $\hat{\rho}_{x-1}$, in which case, the density increases uniformly by $u/(c_x - c_{x-1})$.
- The density $\rho(c_{x-1}, d(J))$ increases to $\hat{\rho}_{x-1}$, in which case, the density increases uniformly by $u/(d(J) - c_{x-1})$.
- The density $\rho(c_{x-1}, d(J))$ increases from below $\rho(d(J), c_x)$ to exactly $\rho(d(J), c_x)$, in which case, the density of C_x increases uniformly by $u/(c_x - c_{x-1})$.

If none of the above events occur, this implies $w(J)$ is small and in this case, J itself increases the density of C_x by $w(J)/(c_x - c_{x-1})$ and we set $u = w(J)$. Taking the minimum value of u among these events ensures that the density increases in exactly one interval I and the amount of increase is $u/|I|$. (This leads to Simple Case A.)

(c) If $\text{SOAT}^t(d(J)) = s_{\text{crit}}$ or 0, suppose $I = (d_1, d_2]$ is the maximal interval where the speed of SOAT^t is s_{crit}. This means that the density of any interval starting from d_1 is at most s_{crit}. If there exists t_1 such that $d_1 < d(J) \leq t_1$ and $\rho(d_1, t_1) = s_{\text{crit}}$, without loss of generality, assume t_1 is the largest such value. In this case, we define the value u as in Case (b) with the interval I replaced by $(d_1, t_1]$. (This leads to Simple Case A.)

Otherwise, we let $u \leq w(J)$ be the largest size such that the density $\rho(d_1, t')$ remains at most s_{crit} for all $t' > d_1$. In this case, the interval I is lengthened by u/s_{crit}. (This leads to Simple Case C.)

Case 2: SOAT^t is idle at t. This means that SOAT^t contains a single interval $I = (t_1, t_2]$, where $t_1 > t$, with speed s_{crit} and at t, the density of any interval starting at t is less than s_{crit}, i.e., $\rho(t, t') < s_{\text{crit}}$ for all $t' > t$. Let $u \leq w(J)$ be the largest size such that the density $\rho(t, t')$ remains at most s_{crit} for all $t' > t$. Then the interval I is lengthened by u/s_{crit}. Furthermore, I may move forward to start earlier than t_1; if the new interval moves to start as early at t, it means SOAT will start working right after this job of size u arrives. (This leads to Simple Case C.)

Summary. Summarizing the two cases, we have defined a value $u \leq w(J)$ such that the arrival of J can be simulated by the arrival of two jobs J_1 and J_2, with $w(J_1) = u$ and $w(J_2) = w(J) - u$, both arriving at time t and having the same deadline as J. Furthermore, J_1 leads to Simple Case B for Case 1 (a), Simple Case A for Case 1 (b), Simple Case A or C for Case 1 (c), and Simple Case C for Case 2. We can repeat this process recursively for the job J_2, until we use up all $w(J)$ units of the work of J. Then, the change in the SOAT schedule due to J is equivalent to a sequence of smaller changes, where each of them corresponds to one of the simple cases. For each of the smaller change in the sequence, the change in the potential $\Delta\phi - \Delta\beta$ is non-positive. So the change in the potential $\Delta\phi - \Delta\beta$ due to J is non-positive.

Lemma 12. *When a job arrives at t_0, we have $\Delta\phi - \Delta\beta \leq 0$.*

4.2.5 Competitiveness of Slow-D(SOA)

Using the above results, we can prove Inequality (1) formally in Lemma 13.

Lemma 13. *At any time t, $E_{\mathrm{w}}(t) + \phi(t) - \beta(t) \leq \alpha^\alpha E^*_{\mathrm{w}}(t)$.*

Proof. We prove the lemma by induction on time. Let $t_0 = 0$ be the time before any job arrives. Obviously, $\phi(t_0)$, $\beta(t_0)$, $E_{\mathrm{w}}(t_0)$, and $E^*_{\mathrm{w}}(t_0)$ all equal 0, and the lemma is true for $t = t_0$. With respect to a job sequence, let $t_1, t_2, \ldots$ be the arrival time of the jobs. Consider any $i \geq 1$. Assume that the lemma is true at time t_{i-1}. Then by Lemma 7, the lemma remains true for all time before the job arrives at t_i. Furthermore, by Lemma 12, when the job arrives at t_i, $\phi(t) - \beta(t)$ cannot increase and the lemma continues to hold at t_i, completing the induction. □

Then we have the following theorem.

Theorem 14. (i) *In the bounded speed model,* Slow-D(SOA) *is $(\alpha^\alpha + \alpha^2 4^\alpha + 2)$-competitive for energy.* (ii) *In the infinite speed model,* SOA *completes all the jobs and is* $\max\{\alpha^\alpha + 2, 4\}$-*competitive for energy.*

Proof. (i) Recall that t_{e} denotes the first time when all deadlines have passed. Lemma 13 implies that $E_{\mathrm{w}}(t_{\mathrm{e}}) + \phi(t_{\mathrm{e}}) - \beta(t_{\mathrm{e}}) \leq \alpha^\alpha E^*_{\mathrm{w}}(t_{\mathrm{e}})$. As both OPT and SOAT have no more unfinished work at t_{e}, $\phi(t_{\mathrm{e}}) = 0$. By Lemma 5, $\beta(t_{\mathrm{e}}) \leq \alpha^2 4^\alpha E^*_{\mathrm{w}}(t_{\mathrm{e}})$. Hence, $E_{\mathrm{w}} \leq (\alpha^\alpha + \alpha^2 4^\alpha) E^*_{\mathrm{w}}$. With Lemma 4, Slow-D(SOA) is $\max\{\alpha^\alpha + \alpha^2 4^\alpha + 2, 4\}$-competitive, i.e., $(\alpha^\alpha + \alpha^2 4^\alpha + 2)$-competitive.

(ii) In the infinite speed model, both SOA and OPT complete all jobs, so $\beta(t) = 0$ for all time t. The proof in the bounded speed model would thus show that SOA is $\max\{\alpha^\alpha + 2, 4\}$-competitive in the infinite speed model. □

5 Throughput analysis of Slow-D(SOA)

For the sake of completeness, we describe the details of Slow-D(SOA) in Section 5.1 and then show how to extend the analysis in [4] to analyze the throughput in Section 5.2.

5.1 The algorithm Slow-D(SOA)

As mentioned in Section 3, for the bounded speed model, we simulate SOA running in infinite speed model and at any time t, Slow-D(SOA) works at the speed $\mathrm{SOAT}(t) = \min\{\mathrm{SOA}(t), T\}$. Slow-D(SOA) also follows the state transition in SOA. Note that unlike SOA, Slow-D(SOA) may not complete all the jobs, thus we need a careful job selection and execution strategy. The strategy relies on a notion called down-time(t).

The notion down-time(t). Consider a particular time t and SOA^t, the planned SOA schedule at t (assuming no more jobs arrive).

We define down-time(t) to be the latest time t' that SOA^t changes speed from above T to at most T (note that t' can be before, at or after t). If there is no such transition, we set down-time(t) $= -\infty$.

By the nature of SOA, down-time(t) is a monotonically non-decreasing function of t no matter how jobs arrive later.

At any time t, we classify all released jobs using down-time(t). A job J is said to be *t-urgent* if $d(J) \leq$ down-time(t), and *t-slack* otherwise. A t-slack job may turn into a t'-urgent job at a later time $t' > t$. On the other hand, a t-urgent job stays urgent until it completes or is discarded since down-time(t) is monotonically non-decreasing.

Before we describe how Slow-D(SOA) makes use of down-time(t), we observe the following properties of SOA.

Fact 15. *Consider the planned schedule* SOA^t*.*

(i) $\mathrm{SOA}(t) > T$ *iff* down-time(t) $> t$*.*

(ii) *If* $\mathrm{SOA}(t) > T$*, then during* $[t, \text{down-time}(t))$*,* SOA^t *processes only t-urgent jobs. (This implies that* SOA *processes t-slack jobs only if* $\mathrm{SOA}(t) \leq T$*.)*

Slow-D(SOA). We now describe how Slow-D(SOA) admits and selects jobs to run. Two job queues Q_{work} and Q_{wait} are kept. At any time t, the earliest deadline job in Q_{work} is processed at speed $\min\{T, \mathrm{SOA}(t)\}$. Q_{work} is always kept feasible at the current time, and each job J enters Q_{work} whenever Q_{work} remains feasible (i.e., can be completed using speed T). Otherwise, J enters Q_{wait}, and waits until $d(J) - w(J)/T$, which is the latest time at which J is feasible. We say an *LST* (Latest Start Time) interrupt occurs. A decision is then made to either discard J or move it to Q_{work}. A decision is also made to possibly expel some jobs from Q_{work} to keep it feasible. An *urgent period* is a maximal time period when there is some urgent job in Q_{work}. We recall the handling of LST interrupts in [4].

LST interrupt [4]. Whenever a job J reaches its latest start time, i.e., $t = d(J) - w(J)/T$, it raises an LST interrupt. At an LST interrupt, we either discard J or expel all t-urgent jobs in Q_{work} to make room for J as follows:

In the current urgent period[3], let J_0 be the last job admitted from Q_{wait} to Q_{work} (if no jobs have been admitted from Q_{wait} so far, let J_0 be a dummy job of size zero admitted just before the current period starts). Consider all the jobs ever admitted to Q_{work} that have become urgent after J_0 has been admitted to Q_{work}, and let W denote the total original size of these jobs. If $w(J) > 2(w(J_0) + W)$, all urgent jobs in Q_{work} are expelled and J is admitted to Q_{work}.

Note that whenever a job enters Q_{wait}, the current time lies in an urgent period.

5.2 Throughput analysis

We now analyze the throughput of Slow-D(SOA) and show that the competitive ratio is 4. Without loss of generality, we assume that the job size of any job J satisfies $w(J) \leq (d(J) - r(J))T$. Note that jobs with bigger size cannot be completed by any algorithm including the optimal offline algorithm OPT. We introduce the following notations to aid the analysis. We

[3]As we shall see LST interrupts occur only during urgent periods.

partition the job set $\mathcal{J}$ into three sets: $\mathcal{J}_{\rm s}$ contains jobs that are slack from release to completion, $\mathcal{J}_{\rm ua}$ contains jobs admitted to $Q_{\rm work}$ at release time and become urgent at some time (perhaps at release), and $\mathcal{J}_{\rm ur}$ contains jobs not admitted to $Q_{\rm work}$ at release and thus wait in $Q_{\rm wait}$.

We attempt to show that Slow-D(SOA) completes all jobs in $\mathcal{J}_{\rm s}$ (Corollary 18) and at least a quarter of the amount of jobs that the optimal offline algorithm can complete for $\mathcal{J}_{\rm ua} \cup \mathcal{J}_{\rm ur}$ (Lemma 20). Therefore, Slow-D(SOA) is 4-competitive for throughput.

Theorem 16. *In the bounded speed model,* Slow-D(SOA) *is* 4*-competitive for throughput.*

5.2.1 Throughput on $\mathcal{J}_{\rm s}$

To analyze the throughput of Slow-D(SOA) on $\mathcal{J}_{\rm s}$, we first show that as compared with SOA, Slow-D(SOA) does well for slack jobs (Lemma 17).

Lemma 17. *At any time t,* Slow-D(SOA) *does not lag behind* SOA *for any t-slack job.*

Proof. First, note that Slow-D(SOA) does not discard any t-slack job before time t; at any time t', only t'-urgent jobs can be discarded and these jobs will remain urgent after t'. Now, consider the execution of slack jobs under SOA. By Fact 15 (ii), SOA works on a t-slack job only if $\text{SOA}(t) \leq T$. By Fact 15 (i), at any time t when $\text{SOA}(t) \leq T$, all t-urgent jobs must have deadline passed, thus, both SOA and Slow-D(SOA) can only work on t-slack jobs. By definition, Slow-D(SOA) uses the same speed as SOA when $\text{SOA}(t) \leq T$. Since both use EDF, Slow-D(SOA) cannot lag behind SOA on any t-slack job. □

Then the following corollaries directly follows.

Corollary 18. Slow-D(SOA) *completes all jobs in $\mathcal{J}_{\rm s}$.*

Corollary 19. *Consider any time t. If no more job arrives after t, all t-slack jobs in $Q_{\rm work}$ can be scheduled to completion using only time after* $\max\{t, \text{down-time}(t)\}$.

Proof. By Fact 15 (ii), if no more job arrives after t, SOA can complete all t-slack jobs using only time after $\max\{t, \text{down-time}(t)\}$. This together with Lemma 17 leads to the corollary. □

5.2.2 Throughput on $\mathcal{J}_{\rm ur} \cup \mathcal{J}_{\rm ua}$

In this section, we show that Slow-D(SOA) completes enough jobs in $\mathcal{J}_{\rm ur} \cup \mathcal{J}_{\rm ua}$. We first introduce the following notations. For a job set L, let $w(L) = \sum_{J \in L} w(J)$ denote the total work of all jobs in L. The *span* of J, denoted $\text{span}(J)$, is the interval $[r(J), d(J)]$. Let $\text{span}(L)$ denote the union of the spans of all jobs in L, and let $|\text{span}(L)|$ be the total length of intervals in $\text{span}(L)$.

Consider each urgent period $U = [S, E)$. Let $\text{join}(U)$ be the total size of jobs in $\mathcal{J}_{\rm ua}$ that become urgent at some time in U, J^* be the latest-deadline job in $\mathcal{J}_{\rm ur}$ that is released during U, and E' be $\max\{d(J^*), E\}$. We denote $[S, E')$ as $\text{secured}(U)$. In Lemma 22 (Section 5.2.3), we will show that the total size of urgent jobs completed by Slow-D(SOA) during U is at least $(\text{join}(U) + |\text{secured}(U)|\ T)/4$. With Lemma 22, we can then bound the amount of work completed for $\mathcal{J}_{\rm ur} \cup \mathcal{J}_{\rm ua}$.

Lemma 20. Slow-D(SOA) *completes at least* $(w(\mathcal{J}_{\rm ua}) + |\text{span}(\mathcal{J}_{\rm ur})|\ T)/4$ *work for jobs in* $\mathcal{J}_{\rm ua} \cup \mathcal{J}_{\rm ur}$.

Proof. Let C be the collection of all urgent periods. Using Lemma 22 which we derive next, we see that the total size of urgent jobs completed by Slow-D over C is at least $\sum_{U\in C}(\text{join}(U) + |\text{secured}(U)|\, T)/4$. We reconcile these terms with $\mathcal{J}_{\text{ua}}$ and $\mathcal{J}_{\text{ur}}$. Since each job in $\mathcal{J}_{\text{ua}}$ becomes urgent at some time, $\sum_{U\in C}\text{join}(U) \geq w(\mathcal{J}_{\text{ua}})$. We claim that each job $J \in \mathcal{J}_{\text{ur}}$ is released during some urgent period U. The definition of secured(U) ensures that span$(J) \subseteq$ secured(U), thus $|\text{span}(\mathcal{J}_{\text{ur}})| \leq \sum_{U\in C}|\text{secured}(U)|$. The lemma then arrives by combining the two inequalities.

To prove the claim, suppose on the contrary that some job $J \in \mathcal{J}_{\text{ur}}$ is not released in an urgent period. Then at time $r(J)$, there is no $r(J)$-urgent job in Q_{work}. If J is $r(J)$-slack, then by Corollary 19, all ($r(J)$-slack) jobs in Q_{work} together with J is feasible. If J is $r(J)$-urgent, J has deadline no later than down-time(t). We can complete J using time $(t, \text{down-time}(t)]$ and all ($r(J)$-slack) jobs in Q_{work} using time after down-time(t) (Corollary 19). Therefore, in both cases, J would enter Q_{work} instead of Q_{wait}, contradicting to that $J \in \mathcal{J}_{\text{ur}}$. □

Combining Corollary 18 and Lemma 20, we are ready to prove Theorem 16.

Proof of Theorem 16. By Corollary 18 and Lemma 20, the throughput of Slow-D(SOA) is at least $w(\mathcal{J}_{\text{s}}) + (w(\mathcal{J}_{\text{ua}}) + |\mathit{span}(\mathcal{J}_{\text{ur}})|\, T)/4$. The throughput of OPT is $\text{OPT}(\mathcal{J}) \leq w(\mathcal{J}_{\text{ua}}) + w(\mathcal{J}_{\text{s}}) + |\text{span}(\mathcal{J}_{\text{ur}})|\, T$, i.e., no more than 4 times that of Slow-D(SOA). □

5.2.3 Throughput during an urgent period

It remains to analyze the throughput of Slow-D(SOA) during an urgent period. To analyze it, we need further investigation of the properties of Q_{wait} and Q_{work}.

Lemma 21. *Consider the scheduling of* Slow-D(SOA).

- **(i)** *Every job $J \in \mathcal{J}_{\text{ur}}$ that is released during an urgent period U must raise an LST interrupt in U (whether it is admitted to Q_{work} or not).*
- **(ii)** *At any time t, Q_{work} is feasible.*

Proof. **(i)** Suppose a job J enters Q_{wait}. Adding J thus makes Q_{work} infeasible. As shown in the proof of Lemma 20, J must be urgent. Since J is feasible by itself, Q_{work} contains some other urgent job at $r(J)$, and $r(J)$ is in an urgent period. There is more than $(d(J) - r(J))T - w(J)$ urgent work in Q_{work} at $r(J)$, otherwise J can complete after all other urgent jobs and cannot make Q_{work} infeasible. Some urgent jobs may be expelled later, but they are replaced by expelling jobs, which have total size larger than the replaced jobs according to Slow-D(SOA). So the total amount of admitted urgent work can only increase. Therefore, the urgent period cannot end before $r(J) + d(J) - r(J) - w(J)/T = d(J) - w(J)/T$, i.e., when the LST interrupt for J occurs.

(ii) By Corollary 19 and the fact that all t-urgent jobs have deadlines at most down-time(t), it suffices to show that at any time t, the set of t-urgent jobs in Q_{work} is feasible. We prove the statement by induction on t. Let t_0 be the time before any job arrives. The statement is trivial for $t = t_0$. Let $t_1, t_2, \ldots$ be all the times when a job is admitted to Q_{work} either at its release time or at its LST interrupt, and $t_1 < t_2 < \cdots$. Consider any $i \geq 1$. Assume the statement is true at t_{i-1}. At any time $t < t_i$, if there are t-urgent jobs in Q_{work}, Slow-D(SOA) always runs them at speed T and thus all t-urgent jobs in Q_{work} remain feasible. At time t_i, if a job is admitted to Q_{work} at its release time, the feasibility check of Slow-D(SOA) guarantees all t-urgent jobs in Q_{work} are feasible. Otherwise, a job J is admitted to Q_{work} at its LST interrupt. Note that J must be an urgent job; because by Corollary 19, we can conclude that the arrival of J makes the set of $r(J)$-urgent jobs in Q_{work} infeasible at $r(J)$ and hence J must be $r(J)$-urgent. Now, at time t_i, all other t_i-urgent jobs are expelled from Q_{work} and J becomes the only t_i-urgent job in Q_{work}. By the definition of LST interrupt, J is feasible at t_i, which completes the proof. □

With the above lemma, we can now analyze the throughput of Slow-D(SOA) during an urgent period.

Lemma 22. *The total size of urgent jobs completed by* Slow-D(SOA) *during an urgent period $U = [S, E)$ is at least* $(\text{join}(U) + |\text{secured}(U)|\, T)/4$.

Proof. Let J_1, J_2, $\ldots$, J_k be the k jobs in Q_{wait} admitted successfully to Q_{work} during U at times $L_1 \le L_2 \le \cdots \le L_k$, respectively. For notational convenience, we let J_0 and J_{k+1} be jobs of size zero, admitted at $L_0 = S$ and $L_{k+1} = E$ respectively. We refine the notation join as follows: For any $0 \le i \le k$, let join_i be the total size of jobs in $\mathcal{J}_{\text{ua}}$ that become urgent between the admittances of J_i and J_{i+1} to Q_{work}. Note that $\text{join}(U) = \sum_{i=0}^{k} \text{join}_i$.

We now show that the jobs admitted by LST interrupts are quite large.

Proposition J: For $i = 0, 1, \ldots, k$, $w(J_i) \ge \sum_{p=0}^{i-1} \text{join}_p + (L_i - S)\, T$.

We show this by induction. The base case $i = 0$ is trivial as both sides are zero. Assume Proposition J is true for some $i \le k-1$. For $i+1$,

$$\begin{aligned}\textstyle\sum_{p=0}^{i} \text{join}_p + (L_{i+1} - S)\, T &= \textstyle\sum_{p=0}^{i-1} \text{join}_p + \text{join}_i + ((L_{i+1} - L_i) + (L_i - S))\, T \\ &\le w(J_i) + \text{join}_i + (L_{i+1} - L_i)\, T \qquad \text{(by induction)} \\ &\le 2\,(w(J_i) + \text{join}_i) < w(J_{i+1})\end{aligned}$$

The second last step follows since during $[L_i, L_{i+1}]$, Slow-D(SOA) running at speed T has not exhausted all the $w(J_i) + \text{join}_i$ urgent work that it can work on. The final step is the condition for J_{i+1} to be admitted in Slow-D(SOA). So the proposition is true for $i = 0, 1, \ldots, k$.

Recall that J^* is the latest-deadline job in $\mathcal{J}_{\text{ur}}$ that is released during U. We observe the following bounds of $w(J^*)$:

$$(d(J^*) - E)\, T \le w(J^*) \le 2\,(w(J_k) + \text{join}_k)\ .$$

The first inequality follows from Lemma 21 (i) which says $d(J^*) - w(J^*)/T \le E$. For the second inequality, if J^* is admitted to Q_{work}, then $J^* = J_i$ for some $i \le k$; otherwise, $w(J^*) \le 2\,(w(J_i) + \text{join}_i)$ for some $i \le k$. In both cases, $w(J^*) \le 2\,(w(J_k) + \text{join}_k)$.

With this bound, we conclude the proof as follows:

$$\begin{aligned}\textstyle\sum_{p=0}^{k} \text{join}_p + |\text{secured}(U)|\, T &= \textstyle\sum_{p=0}^{k} \text{join}_p + (E - S)\, T + \max\{d(J^*) - E, 0\}\, T \\ &\le \textstyle\sum_{p=0}^{k} \text{join}_p + (L_k - S)\, T + (E - L_k)\, T + w(J^*) \\ &\le w(J_k) + \text{join}_k + (E - L_k)\, T + w(J^*) \qquad \text{(Proposition J)} \\ &\le w(J_k) + \text{join}_k + w(J_k) + \text{join}_k + w(J^*) \le 4\,(w(J_k) + \text{join}_k)\ ,\end{aligned}$$

where the fourth line $(E - L_k)T \le w(J_k) + \text{join}_k$ follows since during $[L_k, E)$, Slow-D(SOA) has just enough time to complete all the $w(J_k) + \text{join}_k$ urgent work that it can work on.

Now notice that by Lemma 21 (ii), Slow-D(SOA) completes J_k and all jobs in Q_{work} that become urgent after the admittance of J_k and before E. Therefore the total size of urgent jobs completed by Slow-D(SOA) during U is at least $w(J_k) + \text{join}_k$ and the lemma follows from the inequality in the last paragraph. □

References

[1] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3(4):49, 2007.

[2] S. Albers, F. Muller, and S. Schmelzer. Speed scaling on parallel processors. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 289–298, 2007.

[3] J. Augustine, S. Irani, and C. Swany. Optimal power-down strategies. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 530–539, 2004.

[4] N. Bansal, H. L. Chan, T. W. Lam, and L. K. Lee. Scheduling for speed bounded processors. In *Proceedings of International Colloquium on Automata, Lanaguages and Programming (ICALP)*, pages 409–420, 2008.

[5] N. Bansal, H. L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 693–701, 2009.

[6] N. Bansal, H. L. Chan, K. Pruhs, and D. Rogozhnikov-Katz. Improved bounds for speed scaling in devices obeying the cube-root rule. In *Proceedings of International Colloquium on Automata, Lanaguages and Programming (ICALP)*, pages 144–155, 2009.

[7] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):3, 2007.

[8] N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–813, 2007.

[9] S. K. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. E. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 100–110, 1991.

[10] L. Benini, A. Bogliolo, and G. de Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, 2000.

[11] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[12] H. L. Chan, W. T. Chan, T. W. Lam, L. K. Lee, K. S. Mak, and P. W. H. Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Transactions on Algorithms*, 6(1):10, 2009.

[13] H. L. Chan, J. Edmonds, T. W. Lam, L. K. Lee, A. Marchetti-Spaccamela, and K. Pruhs. Nonclairvoyant speed scaling for flow and energy. In *Proceedings of International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 255–264, 2009.

[14] G. Greiner, T. Nonner, and A. Souza. The bell is ringing in speed-scaled multiprocessor scheduling. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 11–18, 2009.

[15] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, 2000.

[16] S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Transactions on Embedded Computing Systems*, 2(3):325–346, 2003.

[17] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3(4):41, 2007.

[18] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 301–309, 1990.

[19] T. W. Lam, L. K. Lee, H. F. Ting, I. K. K. To, and P. W. H. Wong. Sleep with guilt and work faster to minimize flow plus energy. In *Proceedings of International Colloquium on Automata, Lanaguages and Programming (ICALP)*, pages 665–676, 2009.

[20] T. W. Lam, L. K. Lee, I. K. K. To, and P. W. H. Wong. Competitive non-migratory scheduling for flow time and energy. *Journal of Scheduling.* To appear. Preliminary version appeared in Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures, pages 256–264, 2008.

[21] T. W. Lam, L. K. Lee, I. K. K. To, and P. W. H. Wong. Speed scaling functions for flow time scheduling based on active job count. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 647–659, 2008.

[22] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[23] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, pages 89–102, 2001.

[24] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, 1994.

[25] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 374–382, 1995.