Sleep with Guilt and Work Faster to Minimize Flow plus Energy

Tak-Wah Lam[1,⋆], Lap-Kei Lee[1], Hing-Fung Ting[1], Isaac K. K. To[2,⋆⋆], and Prudence W. H. Wong[2,⋆⋆]

[1] Department of Computer Science, University of Hong Kong.
{twlam, lklee, hfting}@cs.hku.hk
[2] Department of Computer Science, University of Liverpool.
{isaacto, pwong}@liverpool.ac.uk

Abstract. In this paper we extend the study of flow-energy scheduling to a model that allows both sleep management and speed scaling. Our main result is a sleep management algorithm called IdleLonger, which works online for a processor with one or multiple levels of sleep states. The design of IdleLonger is interesting; among others, it may force the processor to idle or even sleep even though new jobs have already arrived. IdleLonger works in both clairvoyant and non-clairvoyant settings. We show how to adapt two existing speed scaling algorithms AJC [15] (clairvoyant) and LAPS [9] (non-clairvoyant) to the new model. The adapted algorithms, when coupled with IdleLonger, are shown to be $O(1)$-competitive clairvoyant and non-clairvoyant algorithms for minimizing flow plus energy on a processor that allows sleep management and speed scaling.
The above results are based on the traditional model with no limit on processor speed. If the processor has a maximum speed, the problem becomes more difficult as the processor, once overslept, cannot rely on unlimited extra speed to catch up the delay. Nevertheless, we are able to enhance IdleLonger and AJC so that they remain $O(1)$-competitive for flow plus energy under the bounded speed model. Non-clairvoyant scheduling in the bounded speed model is left as an open problem.

1 Introduction

Speed scaling, flow and energy. Energy consumption has become a major issue in the design of microprocessors, especially for battery-operated devices. Many modern processors support dynamic speed scaling to reduce energy usage. Recently there is a lot of theory research on online job scheduling taking speed scaling and energy usage into consideration (see [10] for a survey). The challenge arises from the conflicting objectives of providing good quality of service and conserving energy. Among others, the study of minimizing flow time plus energy has attracted much attention [1, 3–5, 9, 14, 15]. The results to date are

⋆ T.W. Lam is partially supported by HKU Grant 7176104.
⋆⋆ I.K.K. To & P.W.H. Wong are partially supported by EPSRC Grant EP/E028276/1.

based on a speed scaling model in which a processor, when running at speed s, consumes energy at the rate of s^α, where α is typically 3 (the cube-root rule [7]). Most research assumes the traditional *infinite speed model* [16] where any speed is allowed; some considers the more realistic *bounded speed model* [8], which imposes a maximum processor speed T.

Total flow time is a commonly used QoS measure for job scheduling. The flow time (or simply flow) of a job is the time elapsed since the job arrives until it is completed. In the online setting, jobs with arbitrary sizes arrive at unpredictable times. They are to be run on a processor which allows preemption without penalty. To understand the tradeoff between flow and energy, Albers and Fujiwara [1] initiated the study of minimizing a linear combination of total flow and total energy. The intuition is that, from an economic viewpoint, users are willing to pay a certain (say, ρ) units of energy to reduce one unit of time. By changing the units of time and energy, one can further assume that $\rho = 1$ and thus would like to optimize flow plus energy.

Under the infinite speed model, Albers and Fujiwara [1] considered jobs of unit size, and their work was extended to jobs of arbitrary sizes by Bansal, Pruhs and Stein [5]. The BPS algorithm scales the speed as a function of unfinished work and is $O((\frac{\alpha}{\ln \alpha})^2)$-competitive for minimizing flow plus energy. Bansal et al. [3] later adapted the BPS algorithm to the bounded speed model; the competitive ratio remains $O((\frac{\alpha}{\ln \alpha})^2)$ if the online algorithm is given extra speed. Recently, Lam et al. [15] gave a new algorithm AJC whose speed function depends on the number of unfinished jobs. AJC avoids the extra speed requirement and improves the competitive ratio to $O(\frac{\alpha}{\ln \alpha})$. Recall that α is typically equal to 3. Then the competitive ratios of BPS are estimated to be 7.9 and 11.9, and AJC 3.25 and 4 under the infinite and bounded speed model, respectively. More recently, Bansal et al. [4] further showed that AJC can be adapted to be 3-competitive, independent of α. All these results assume clairvoyance, i.e., the size of a job is known when the job arrives.

The non-clairvoyant setting, where job size is known only when the job is completed, is practically important. Chan et al. [9] have recently given a non-clairvoyant speed scaling algorithm LAPS that is $O(\alpha^3)$-competitive for total flow plus energy in the infinite speed model.

Sleep management. In earlier days, energy reduction was mostly achieved by allowing a processor to enter a low-power *sleep* state, yet waking up requires extra energy. In the (embedded systems) literature, there are different energy-efficient strategies to bring a processor to sleep during a period of zero load [6]. This is an online problem, usually referred to as *dynamic power management*. The input is the length of the period, known only when the period ends. There are several interesting results with competitive analysis (e.g., [2, 11, 13]). In its simplest form, the problem assumes the processor is in either the *awake* state or the *sleep* state. The awake state always requires a static power $\sigma > 0$. To have zero energy usage, the processor must enter the sleep state, but a wake-up back to the awake state requires $\omega > 0$ energy. In general, there can be multiple intermediate sleep states, which demand some static power but less wake-up energy.

It is natural to study job scheduling on a processor that allows both sleep states and speed scaling. More specifically, a processor in the awake state can run at any speed $s \geq 0$ and consumes energy at the rate $s^\alpha + \sigma$, where $\sigma > 0$ is static power and s^α is the dynamic power[3]. Here job scheduling requires two components: a *sleep management algorithm* to determine when to sleep or work, and a *speed scaling algorithm* to determine which job and at what speed to run. Notice that sleep management here is not the same as in dynamic power management; in particular, the length of a sleep or idle period is part of the optimization (rather than the input). Adding a sleep state actually changes the nature of speed scaling. Assume no sleep state, running a job slower is a natural way to save energy. Now one can also save energy by sleeping more and working faster later. It is even more complicated when flow is concerned. Prolonging a sleeping period by delaying job execution can save energy, yet it also incurs extra flow. Striking a balance is not trivial. In the theory literature, the only relevant work is by Irani et al. [12]; they studied deadline scheduling on a processor with one sleep state and infinite speed scaling. They showed an $O(1)$-competitive algorithm to minimize the energy for meeting the deadlines of all jobs.

Our contributions. This paper initiates the study of flow-energy scheduling that exploits both speed scaling and multiple sleep states. We give a sleep management algorithm called IdleLonger, which works for a processor with one or multiple levels of sleep states. IdleLonger works in both clairvoyant and non-clairvoyant settings. We adapt the clairvoyant speed scaling algorithm AJC [15] and the non-clairvoyant algorithm LAPS [9] to take the static power σ into consideration. Under the infinite speed model, these adapted algorithms together with IdleLonger are shown to be $O(1)$-competitive for minimizing flow plus energy in the clairvoyant and non-clairvoyant settings, respectively. More precisely, the ratios are $O(\frac{\alpha}{\ln \alpha})$ and $O(\alpha^3)$ (recall that α is a constant).

For the bounded speed model, the problem becomes more difficult since the processor, once overslept, cannot rely on unlimited extra speed to catch up the delay. Nevertheless, we are able to enhance IdleLonger and AJC to observe the maximum processor speed. They remain $O(1)$-competitive for flow plus energy under the bounded speed model.

Sleep management algorithm IdleLonger. When the processor is sleeping, it is natural to delay waking up until sufficient jobs have arrived. The non-trivial case is when the processor is idle (i.e., awake but at zero speed), IdleLonger has to determine when to start working again or go to sleep. At first glance, if some new jobs arrive while the processor is idle, the processor should run the jobs immediately so as to avoid extra flow. Yet this would allow the adversary to easily keep the processor awake, and it is difficult to achieve $O(1)$-competitiveness. In an idle period, IdleLonger considers the (static) energy and flow accumulated during the period as two competing quantities. Only if the flow exceeds the energy, IdleLonger would start to work. Otherwise, IdleLonger will remain idle

[3] 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.

until the energy reaches to a certain level; then the processor goes to sleep even in the presence of jobs.

Analysis framework. Apparently, a sleep management algorithm and a speed scaling algorithm would affect each other; analyzing their relationship and their total cost could be a complicated task. Interestingly, the results of this paper stem from the fact that we can isolate the analysis of these algorithms. We divide the total cost (flow plus energy) into two parts, *working cost* (incurred while working on jobs) and *inactive cost* (incurred at other times). We upper bound the inactive cost of IdleLonger independent of the speed scaling algorithm. For the working cost, although it does depend on both algorithms, our potential analysis of the speed scaling algorithms reveals that the dependency on the sleep management algorithm is limited to a simple quantity called *inactive flow*, which is the flow part of the inactive cost. Intuitively, large inactive flow means many jobs are delayed due to prolonged sleep, and hence the processor has to work faster later to catch up, incurring a higher working cost. It is easy to minimize inactive flow at the sacrifice of the energy part of the inactive cost. IdleLonger is designed to maintain a good balance between them. In conclusion, coupling IdleLonger with AJC and LAPS, we obtain competitive algorithms for flow plus energy.

Organization of the paper. Section 1.1 defines the model formally. Sections 2 and 3 focus on the infinite speed model and discuss the sleep management algorithm IdleLonger and two speed scaling algorithms. Finally, Section 4 presents our results on the bounded speed model.

1.1 Model and Notations

The input is a sequence of jobs arriving online. We denote the release time and work requirement (or size) of a job J as $r(J)$ and $w(J)$, respectively.

Speed and power. We first consider the setting with one sleep state. At any time, a processor is in either the *awake* state or the *sleep* state. In the former, the processor can run at any speed $s \geq 0$ and demands power in the form $s^\alpha + \sigma$, where $\alpha > 1$ and $\sigma > 0$ are constants. We call s^α the dynamic power and σ the static power. In the sleep state, the speed is zero and the power is zero. State transition requires energy; without loss of generality, we assume a transition from the sleep state to the awake state requires an amount ω of energy, and the reverse takes zero energy. To simplify our work, we assume state transition takes no time.

Next we consider the setting with $m > 1$ levels of sleep. A processor is in either the *awake* state or the *sleep-i* state, where $1 \leq i \leq m$. The awake state is the same as before, demanding static power σ and dynamic power s^α. For convenience, we let $\sigma_0 = \sigma$. The sleep-m state is the only "real" sleep state, which has static power $\sigma_m = 0$; other sleep-i states have decreasing positive static power σ_i such that $\sigma_0 > \sigma_1 > \sigma_2 > \cdots > \sigma_{m-1} > \sigma_m = 0$. We denote the wake-up energy from the sleep-i state to the awake state as ω_i. Note that $\omega_m > \omega_{m-1} > \cdots > \omega_1 > 0$.

It is useful to differentiate two types of awake state: with zero speed and with positive speed. The former is called *idle* state and the latter is *working* state.

Flow and energy. Consider any schedule of jobs. The flow $F(J)$ of a job J is the time elapsed since it arrives and until it is completed. The total flow is $F = \sum_J F(J)$. Note that $F = \int_0^\infty n(t)\ \mathrm{d}t$, where $n(t)$ is the number of unfinished jobs at time t. Based on this view, we divide F into two parts: F_w is the flow incurred during time intervals of working state, and F_i for idle or sleep state. The energy usage is also divided into three parts: W denotes the energy due to wake-up transitions, E_i is the idling energy (static power consumption in the idle or intermediate sleep state), and E_w is the working energy (static and dynamic power consumption in the working state). Our objective is to minimize the *total cost* $G = F_\mathrm{w} + F_\mathrm{i} + E_\mathrm{i} + E_\mathrm{w} + W$. We call $F_\mathrm{w} + E_\mathrm{w}$ the *working cost*, and $F_\mathrm{i} + E_\mathrm{i} + W$ the *inactive cost*.

2 Sleep Management Algorithm IdleLonger

This section presents a sleep management algorithm called IdleLonger that determines when the processor should sleep, idle, and work (with speed > 0). IdleLonger can be coupled with any *speed scaling algorithm*, which specifies which job and at what speed the processor should run when the processor is working. As a warm-up, we first consider the case with a single sleep state. Afterwards, we consider the general case of multiple sleep states.

In this section, we derive an upper bound of the inactive cost of IdleLonger independent of the choice of the speed scaling algorithm. Section 3 will present two speed scaling algorithms for the clairvoyant and non-clairvoyant settings, respectively, and analyze their working costs when they are coupled with IdleLonger. In conclusion, putting IdleLonger and each of these two speed scaling algorithms together, we can show that both the inactive cost and working cost are $O(1)$ times of the total cost of the optimal offline algorithm OPT.

2.1 Sleep Management Algorithm for a Single Sleep State

When the processor is in the working state and sleep state, it is relatively simple to determine the next transition. In the former, the processor keeps on working as long as there is an unfinished job; otherwise switch to the idle state. In the sleep state, we avoid waking up immediately after a new job arrives as this requires energy. It is natural to wait until the new jobs have accumulated enough flow, say, at least the wake-up energy ω, then we let the processor to switch to working state direct. Below we refer the flow accumulated due to new jobs over a period of idle or sleep state as the *inactive flow* of that period.

When the processor is in idle state, it is non-trivial when to switch to the sleep or working state. Intuitively, the processor should not stay in idle state too long, because it consumes energy (at the rate of σ) but does not get any work done. Yet to avoid frequent wake-up in future, the processor should not sleep immediately. Instead the processor should wait for possible job arrival and sleep only after the idling energy (i.e., σ times the length of idling interval) reaches the wake-up energy ω. When a new job arrives in the idle state, a naive idea is to let the processor switch to the working state to process the job immediately; this avoids accumulating inactive flow. Yet this turns out to be a bad strategy as it becomes too difficult to sleep; e.g., the adversary can use some tiny jobs sporadically, then the processor would never accumulate enough idling energy to sleep.

It is perhaps counter-intuitive that IdleLonger always prefers to idle a bit longer, and it can switch to the sleep state even in the presence of unfinished jobs. The idea is to consider the inactive flow and idling energy at the same time. Note that when an idling period gets longer, both the inactive flow and idling energy increase, but at different rates. We imagine that these two quantities are competing with each other.

> The processor switches from the idle state to the working state once the inactive flow catches up with the idling energy. If the idling energy has exceeded ω before the inactive flow catches up with the idling energy, the processor switches to the sleep state.

Below is a summary of the above discussion. For simplicity, IdleLonger is written in a way that it is being executed continuously. In practice, we can rewrite the algorithm such that the execution is driven by discrete events like job arrival, job completion and wake-up.

Algorithm 1 IdleLonger(A): A is any speed scaling algorithm

```
At any time $t$, let $n(t)$ be the number of unfinished jobs at $t$.
In working state: If $n(t) > 0$, keep working on jobs according to the algorithm $A$;
else (i.e., $n(t) = 0$), switch to idle state.
In idle state: Let $t' \leq t$ be the last time in working state ($t' = 0$ if undefined). If the
inactive flow over $[t', t]$ equals $(t - t')\sigma$, then switch to working state;
Else if $(t - t')\sigma = \omega$, switch to sleep state.
In sleep state: Let $t' \leq t$ be the last time in working state ($t' = 0$ if undefined). If
the inactive flow over $[t', t]$ equals $\omega$, switch to working state.
```

Below we upper bound the inactive cost of IdleLonger (the working cost will be dealt with in Section 3). It is useful to define three types of time intervals. An $\mathrm{I_w}$-interval is a maximal interval in idling state with a transition to the working state at the end, and similarly an $\mathrm{I_s}$-interval for that with a transition to the sleep state. Furthermore, an $\mathrm{IS_w}$-interval is a maximal interval comprising an $\mathrm{I_s}$-interval, a sleeping interval, and finally a wake-up transition. As the processor starts in the sleep state, we allow the first $\mathrm{IS_w}$-interval containing no $\mathrm{I_s}$-interval.

Consider a schedule of IdleLonger(A). Recall that the inactive cost is composed of W (wake-up energy), F_i (inactive flow), and E_i (idling energy). We further divide E_i into two types: $\boldsymbol{E_\mathrm{iw}}$ is the idling energy incurred in all $\mathrm{I_w}$-intervals, and $\boldsymbol{E_\mathrm{is}}$ for all $\mathrm{I_s}$-intervals.

By the definition of IdleLonger, we have the following property.

Property 1. (i) $F_\mathrm{i} \leq W + E_\mathrm{iw}$, and (ii) $E_\mathrm{is} = W$.

Therefore, the inactive cost of IdleLonger, defined as $W + F_{\mathrm{i}} + E_{\mathrm{iw}} + E_{\mathrm{is}}$, is at most $3W + 2E_{\mathrm{iw}}$. The non-trivial part is to upper bound W and E_{iw}. Our main result is stated below. For the optimal offline algorithm OPT, we divide its total cost G^* into two parts: W^* is the total wake-up energy, and $C^* = G^* - W^*$ (i.e., the total flow plus the working and idling energy).

Theorem 1. $W + E_{\mathrm{iw}} \leq C^* + 2W^*$.

Corollary 1. *The inactive cost of* IdleLonger *is at most* $3C^* + 6W^*$.

The rest of this section is devoted to proving Theorem 1. Note that W is the wake-up energy consumed at the end of all $\mathrm{IS_w}$-intervals, and E_{iw} is the idling energy of all $\mathrm{I_w}$-intervals. All these intervals are disjoint. Below we show a charging scheme such that, for each $\mathrm{IS_w}$-interval, we charge OPT a cost at least ω, and for each $\mathrm{I_w}$-interval, we charge OPT at least the idling energy of this interval. Thus, the total charge to OPT is at least $W + E_{\mathrm{iw}}$. On the other hand, we argue that the total charge is at most $C^* + 2W^*$. Therefore, $W + E_{\mathrm{iw}} \leq C^* + 2W^*$.

The charging scheme for an $\mathrm{IS_w}$-interval $[t_1, t_2]$ is as follows. The target is at least ω.

Case 1. If OPT switches from or to the sleep state in $[t_1, t_2]$, we charge OPT the cost ω of the first wake-up in $[t_1, t_2]$ (if it exists) or of the last wake-up before t_1.

Case 2. If OPT is awake throughout $[t_1, t_2]$, we charge OPT the static energy $(t_2 - t_1)\sigma$. Note that in an $\mathrm{IS_w}$-interval, IdleLonger has an idle-sleep transition, and hence $(t_2 - t_1)\sigma > \omega$.

Case 3. If OPT is sleeping throughout $[t_1, t_2]$, we charge OPT the inactive flow (i.e., the flow incurred by new jobs) over $[t_1, t_2]$. In this case, OPT and IdleLonger have the same amount of inactive flow during $[t_1, t_2]$, which equals ω (because IdleLonger wakes up at t_2).

For an $\mathrm{I_w}$-interval, we use the above charging scheme again. The definition of $\mathrm{I_w}$-interval allows the scheme to guarantee a charge of $(t_2 - t_1)\sigma$ instead of ω. Specifically, as an $\mathrm{I_w}$-interval ends with an idle-working transition, the inactive flow accumulated in $[t_1, t_2]$ is $(t_2 - t_1)\sigma$, and the latter cannot exceed ω. Therefore, the charge of Case 1, which equals ω, is at least $(t_2 - t_1)\sigma$. Case 2 charges exactly $(t_2 - t_1)\sigma$. For Case 3, we charge OPT the inactive flow during $[t_1, t_2]$. Note that OPT and IdleLonger accumulate the same inactive flow, which is $(t_2 - t_1)\sigma$.

Summing over all $\mathrm{I_w}$- and $\mathrm{IS_w}$-intervals, we have charged OPT at least $W + E_{\mathrm{iw}}$. On the other hand, since all these intervals are disjoint, in Cases 2 and 3, the charge comes from non-overlapping flow and energy of C^*. In Case 1, each OPT's wake-up from the sleep state is charged for ω at most twice, thus the total charge is at most $2W^*$. In conclusion, $W + E_{\mathrm{iw}} \leq C^* + 2W^*$.

2.2 Sleep Management Algorithm for $m \geq 2$ Levels of Sleep States

We extend the previous sleep management algorithm to allow intermediate sleep states, which demand less idling (static) energy than the idling state, and also less wake-up energy than the final sleep state (i.e., sleep-m state). We treat the sleep-m state as the only sleep state in the single-level setting, and adapt the transition rules of the idling state for the intermediate sleep states. The key idea is again to compare inactive flow against idling energy continuously. To ease our discussion, we treat the idle state as the sleep-0 state with wake-up energy $\omega_0 = 0$.

Algorithm 2 IdleLonger(A): A is any speed scaling algorithm

```
At any time $t$, let $n(t)$ be the number of unfinished jobs at $t$.
In working state: If $n(t) > 0$, keep working on the jobs according to the algorithm $A$; else if $n(t) = 0$, switch to idle state.
In sleep-$j$ state, where $0 \leq j \leq m-1$: Let $t' \leq t$ be the last time in the working state, and let $t''$, where $t' \leq t'' \leq t$, be the last time switching from sleep-$(j-1)$ state to sleep-$j$ state. If the inactive flow over $[t', t]$ equals $(t-t'')\sigma_j + \omega_j$, then wake up to the working state;
Else if $(t-t'')\sigma_j = (\omega_{j+1} - \omega_j)$, switch to sleep-$(j+1)$ state.
In sleep-$m$ state: Let $t' \leq t$ be the last time in the working state. If the inactive flow over $[t', t]$ equals $\omega_m$, then wake up to the working state.
```

When we analyze the multi-level algorithm, the definition of W (total wake-up cost) and F_i (total inactive flow) remain the same, but E_is and E_iw have to be generalized. Below we refer a maximal interval during which the processor is in a particular sleep-j state, where $0 \leq j \leq m$, as a *sleep interval* or more specifically, a sleep-j interval. Note that all sleep intervals, except sleep-m intervals, demand idling (static) energy. We denote $\boldsymbol{E_\text{iw}}$ as the idling energy for all sleep intervals that end with a wake-up transition, and $\boldsymbol{E_\text{is}}$ the idling energy of all sleep intervals ending with a (deeper) sleep transition.

IdleLonger imposes a rigid structure of sleep intervals. Define $\ell_j = (\omega_{j+1} - \omega_j)/\sigma_j$. A sleep-$j$ interval can appear only after a sequence of lower level sleep intervals, which starts with an sleep-0 interval of length ℓ_0, followed by a sleep-1 interval of length ℓ_1, ..., and finally a sleep-$(j-1)$ interval of length ℓ_{j-1}. Consider a maximum sequence of such sleep intervals that ends with a transition to the working state. We call the entire time interval enclosed by this sequence an $\text{IS}_\text{w}[j]$-interval for some $0 \leq j \leq m$ if the deepest (also the last) sleep subinterval is of level j. It is useful to observe the following lemma about an $\text{IS}_\text{w}[j]$-interval. Its proof is left in the full paper.

Lemma 1. *Consider any* $\text{IS}_\text{w}[j]$*-interval* $[t_1, t_2]$*, where* $0 \leq j \leq m$*. Assume that the last sleep-*j *(sub)interval is of length* ℓ*. Then,* $\omega_j + \ell\sigma_j \leq \omega_k + (t_2 - t_1)\sigma_k$ *for any* $0 \leq k \leq m$*.*

It is not hard to see that the rigid sleeping structure of IdleLonger allows us to maintain Property 1 as before. That is, (i) $F_\text{i} \leq W + E_\text{iw}$, and (ii) $E_\text{is} = W$. Thus, the inactive cost, which is equal to $F_\text{i} + E_\text{iw} + E_\text{is} + W$, is still at most $3W + 2E_\text{iw}$. In the rest of this section, we prove that W and E_iw have the same upper bound as before.

Theorem 2. *In the setting of* $m \geq 2$ *sleep states,* $W + E_\text{iw} \leq C^* + 2W^*$*.*

To account for W and E_{iw}, it suffices to look at all $\mathrm{IS}_{\mathrm{w}}[j]$-intervals, where $0 \leq j \leq m$. For each $\mathrm{IS}_{\mathrm{w}}[j]$-interval, we show how to charge OPT a cost $\omega_j + \ell\sigma_j$, where ℓ is length of the deepest sleep subinterval (it is useful to recall that $\omega_0 = 0$ and $\sigma_m = 0$). Then we argue that the total cost charged is at least $W + E_{\mathrm{iw}}$ and at most $C^* + 2W^*$.

Without loss of generality, we can assume that in a maximal interval $[r_1, r_2]$ that OPT is not working, if OPT has ever slept (in sleep-1 or deeper sleep state), then $[r_1, r_2]$ contains only one sleep transition, which occurs at r_1, and the processor remains in the same sleep state until r_2.

Charging scheme. Consider any $\mathrm{IS}_{\mathrm{w}}[j]$-interval $[t_1, t_2]$, where $0 \leq j \leq m$. Let ℓ be the length of the sleep-j (sub)interval in this interval.

Case 1. If OPT has ever switched from or to the sleep-1 or deeper sleep state in $[t_1, t_2]$, let $k \geq 1$ be the deepest sleep level involved in the entire interval. Note that OPT uses static energy at least $(t_2 - t_1)\sigma_k$ during $[t_1, t_2]$. We charge OPT the sum of $(t_2 - t_1)\sigma_k$ and ω_k (in view of a wake-up from sleep-k state inside $[t_1, t_2]$ or after t_2; if there is no-wake up after t_2, then we charge OPT the first wake-up). By Lemma 1, this charge is at least $\omega_j + \ell\sigma_j$.

Case 2. If OPT is working or idle throughout $[t_1, t_2]$, we charge OPT the static energy $(t_2 - t_1)\sigma_0$, which, by Lemma 1, is at least $\omega_j + \ell\sigma_j$.

Case 3. If OPT is sleeping (at any level except zero) throughout $[t_1, t_2]$, we charge OPT the inactive flow over $[t_1, t_2]$. Note that OPT has the same amount of inactive flow as IdleLonger. By definition of a wake-up transition in IdleLonger, the inactive flow equals $\omega_j + \ell\sigma_j$.

Since $\mathrm{IS}_{\mathrm{w}}[j]$-intervals are all disjoint, the flow and idling (static) energy charged to OPT by Cases 1, 2 and 3 come from different parts of C^*. For Case 1, each of OPT's wake-up from a sleep state is charged at most twice. Thus, $W + E_{\mathrm{iw}} \leq C^* + 2W^*$, completing the proof of Theorem 2.

3 Clairvoyant and Non-clairvoyant Speed Scaling

We consider both the clairvoyant and non-clairvoyant settings; in the latter, job size is only known when the job completes. IdleLonger can actually work in both settings as its decision does not depend on the job size. When there is no sleep state and the power function is in the form of s^α, [15] and [9] gave respectively a clairvoyant speed scaling algorithm AJC and a non-clairvoyant speed scaling algorithm LAPS that are $O(1)$-competitive for flow plus energy. These algorithms always run the jobs at a speed proportional to $n(t)^{1/\alpha}$, where $n(t)$ is the number of unfinished jobs at time t. In the sleep setting, when the processor is working, it requires at least the static power σ; if σ is large, running at a speed comparable to $n(t)^{1/\alpha}$ would be too slow to be cost effective as the dynamic power could be way smaller than σ. Indeed AJC and LAPS have unbounded competitive ratios no matter what sleep management algorithm is used. This section shows how to analyze the following simple adaptations of AJC and LAPS to the sleep setting, and upper bound their working costs in terms of OPT's total cost.

Clairvoyant algorithm SAJC. At any time t, SAJC runs the job with the shortest remaining work at the speed $(n(t) + \sigma)^{1/\alpha}$.

Non-clairvoyant algorithm SLS. At any time t, SLS runs at speed $(1 + \frac{3}{\alpha})(n(t) + \sigma)^{1/\alpha}$, and runs the $\lceil (\frac{1}{2\alpha})n(t) \rceil$ unfinished jobs with the latest release times by splitting the speed equally among these jobs.

The analysis of SAJC (resp. SLS) is valid no matter what (non-clairvoyant) sleep management algorithm Slp is being used together. Ideally we want to upper bound the working cost of SAJC and SLS solely in terms of the total cost of OPT, yet this is not possible as the working cost also depends on Slp. Below we give an analysis in which the dependency on Slp is bounded by the inactive flow incurred by Slp. More specifically, let G_{w} and F_{i} be respectively the working cost and the inactive flow of Slp(SAJC) (resp. Slp(SLS)). Again, we use C^* to denote the total cost of OPT minus the wake-up energy; the latter is denoted by W^*. We will show that $G_{\text{w}} = O(C^* + F_{\text{i}})$.

Let us look at a simple case. If Slp always switches to working state whenever there are unfinished jobs, then $F_{\text{i}} = 0$. In this case we can easily adapt the analysis of [15] (resp. [9]) to bound G_{w} in terms of C^* only. However, the inactive cost of Slp may be unbounded in this case. On the other hand, consider a sleep management algorithm that prefers to wait for more jobs before waking up to work (e.g., IdleLonger). Then SAJC (resp. SLS) would start at a higher speed and G_{w} can be much larger than C^*. Roughly speaking, the excess is due to the fact that the online algorithm is sleeping while OPT is working. Note that the cost to catch up the work lagged behind increases at a rate depending on $n(t)$. This motivates us to bound the excess in terms of F_{i}. Based on this idea, we can adapt the potential analyses of AJC [15] and LAPS [9] to show Theorem 3 below, where $\beta = 2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$ and $\gamma = (1 + (1 + \frac{3}{\alpha})^{\alpha}) \leq 1 + e^3$. Together with the results on inactive cost of IdleLonger (Property 1 and Corollary 1), it implies that the clairvoyant algorithm IdleLonger(SAJC) is $(2\beta+2)$-competitive for flow plus energy, and the non-clairvoyant algorithm IdleLonger(SLS) is $((4\alpha^3 + \alpha)\gamma + 3)$-competitive for flow plus energy. Detailed proofs will be given in the full paper.

Theorem 3. (i) *With respect to* Slp(SAJC)*,* $G_{\text{w}} \leq \beta C^* + (\beta - 2)F_{\text{i}}$. (ii) *With respect to* Slp(SLS)*,* $G_{\text{w}} \leq (4\alpha^3 + 1)\gamma C^* + (\alpha - 1)\gamma F_{\text{i}}$.

Corollary 2. *In the setting of single sleep state or multiple sleep states,*
(i) *the total cost of the clairvoyant algorithm* IdleLonger(SAJC) *is at most* $(2\beta + 2)$ *times of the total cost of OPT, and*
(ii) *the total cost of the non-clairvoyant algorithm* IdleLonger(SLS) *is at most* $((4\alpha^3 + \alpha)\gamma + 3)$ *times of the total cost of OPT.*

4 Bounded Speed Model

This section extends the sleep management algorithm IdleLonger and the clairvoyant speed scaling algorithm SAJC to the bounded speed model. We consider the setting where the processor speed is upper bounded by a constant $T > 0$, and there are $m \geq 1$ levels of sleep states. We show that the total cost (comprising inactive and working cost) of IdleLonger(SAJC) is $O(1)$ times of the optimal offline algorithm OPT.

Adaptation. In the bounded speed model, IdleLonger (see Section 2) still works and the inactive cost is $O(1)$ times of OPT's total cost. However, IdleLonger often allows a long sleep, then a speed scaling algorithm, without the capability to speed up arbitrarily, cannot always catch up the progress of OPT and may have unbounded working cost. Thus, we adapt IdleLonger to wake up earlier, especially when too many new jobs have arrived. To this end, we add one more wake-up condition to IdleLonger. Recall that $\sigma (= \sigma_0)$ is the static power in the working state.

> In the sleep-j state, where $0 \leq j \leq m$, if the number of unfinished jobs exceeds σ, the processor wakes up to the working state.

Recall that SAJC runs at the speed $(n(t) + \sigma)^{1/\alpha}$, where $n(t)$ is the number of unfinished jobs at time t. To adapt SAJC to the bounded speed model, we simply cap the speed at T. I.e., at any time t, the processor runs at the speed $\min\{(n(t) + \sigma)^{1/\alpha}, T\}$.

Inactive cost of IdleLonger. The rigid structure of sleep intervals remains the same as before, and the inactive cost is still at most $3W + 2E_{\text{iw}}$, where W is the wake-up energy and E_{iw} is the idling energy incurred in those idling or intermediate sleep intervals that end with a wake-up transition (see Section 2 for details). However, due to the additional wake-up rule, IdleLonger has a slightly worse bound on W plus E_{iw}. Our main result is stated in Theorem 4. Again, W^* denotes the wake-up energy of OPT, and C^* is the total cost of OPT minus W^*.

Theorem 4. (i) $W + E_{\text{iw}} \leq C^* + 3W^*$. (ii) *The inactive cost of* IdleLonger *is at most* $3C^* + 9W^*$.

To prove Theorem 4(i), we extend the charging scheme in Section 2.2 to show that for each $\text{IS}_\text{w}[j]$-interval, OPT can be charged with a cost at least $\omega_j + \ell\sigma_j$, where ℓ is the length of the deepest sleep subinterval (recall that $\omega_0 = 0$, $\sigma_0 = \sigma$ and $\sigma_m = 0$). The three cases of the old charging scheme remain the same, except that Case 3 is restricted to $\text{IS}_\text{w}[j]$-intervals where IdleLonger wakes up at the end due to excessive inactive flow. We supplement Case 3 with a new scheme (Case 4) to handle $\text{IS}_\text{w}[j]$-intervals with wake-ups due to more than σ unfinished jobs.

Charging scheme – Case 4. If OPT is sleeping (at any level except zero) throughout an $\text{IS}_\text{w}[j]$-interval $[t_1, t_2]$, and IdleLonger has accumulated more than σ unfinished jobs at t_2, we consider two scenarios to charge OPT, depending on $n_\text{o}(t_1)$, the number of unfinished jobs in OPT at t_1.
(a) Suppose $n_\text{o}(t_1) \geq \sigma_0$. We charge OPT the inactive flow of these jobs over $[t_1, t_2]$, which is at least $(t_2 - t_1)\sigma_0$. By Lemma 1, this charge is at least $\omega_j + \ell\sigma_j$.
(b) Suppose $n_\text{o}(t_1) < \sigma_0$. Note that OPT stays in a sleep-k state, for some $k \geq 1$, in the entire interval and uses static energy $(t_2 - t_1)\sigma_k$ during $[t_1, t_2]$. We charge OPT the sum of $(t_2 - t_1)\sigma_k$ and ω_k (in view of OPT's first wake-up after t_2, which must exist because new jobs have arrived within $[t_1, t_2]$). By Lemma 1, this charge is at least $\omega_j + \ell\sigma_j$.

In conclusion, we are able to charge OPT, for each $\mathrm{IS_w}[j]$-interval, a cost at least $\omega_j + \ell\sigma_j$. Therefore, the sum of the charges to all $\mathrm{IS_w}[j]$-intervals is at least $W + E_{\mathrm{iw}}$. On the other hand, recall that Case 1 has a total charge at most $2W^*$. Case 2, 3 and 4(a) have a total charge at most C^*. We can argue that OPT is charged by Case (4b) with a cost at most W^*; details are left in the full paper. Then we have $W + E_{\mathrm{iw}} \leq C^* + 3W^*$. And Theorem 4(ii) follows directly. In the full paper, we will adapt the potential analysis of AJC [15] and show that the working cost of SAJC is still $O(\frac{\alpha}{\ln \alpha})$ times OPT's total cost. Therefore, IdleLonger(SAJC) remains $O(\frac{\alpha}{\ln \alpha})$-competitive for flow plus energy.

References

1. S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *TALG*, 3(4), 2007.
2. J. Augustine, S. Irani, and C. Swany. Optimal power-down strategies. *FOCS*, pages 530–539, 2004.
3. N. Bansal, H. L. Chan, T. W. Lam, and L. K. Lee. Scheduling for speed bounded processors. *ICALP*, pages 409–420, 2008.
4. N. Bansal, H. L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. *SODA*, pages 693–701, 2009.
5. N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. *SODA*, pages 805–813, 2007.
6. 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.
7. 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.
8. H. L. Chan, W. T. Chan, T. W. Lam, L. K. Lee, K. S. Mak, and P. W. H. Wong. Energy efficient online deadline scheduling. *SODA*, pages 795–804, 2007.
9. H. L. Chan, J. Edmonds, T. W. Lam, L. K. Lee, A. Marchetti-Spaccamela, and K. Pruhs. Nonclairvoyant speed scaling for flow and energy. *STACS*, pages 255–264, 2009.
10. S. Irani and K. Pruhs. Algorithmic problems in power management. *SIGACT News*, 32(2):63–76, 2005.
11. S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Tran. Embeded Comp. Sys.*, 2(3):325–346, 2003.
12. S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *TALG*, 3(4), 2007.
13. A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. *SODA*, pages 301–309, 1990.
14. T. W. Lam, L. K. Lee, I. K. K. To, and P. W. H. Wong. Competitive non-migratory scheduling for flow time and energy. *SPAA*, pages 256–264, 2008.
15. 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. *ESA*, pages 647–659, 2008.
16. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. *FOCS*, 374–382, 1995.