Nonclairvoyant Sleep Management and Flow-time Scheduling on Multiple Processors

Sze-Hang Chan
Department of Computer Science
University of Hong Kong
shchan@cs.hku.hk

Tak-Wah Lam*
Department of Computer Science
University of Hong Kong
twlam@cs.hku.hk

Lap-Kei Lee†
Department of Computer Science
University of Hong Kong
lklee@cs.hku.hk

Jianqiao Zhu
Department of Computer Sciences
University of Wisconsin-Madison
jianqiao@cs.wisc.edu

ABSTRACT

In large data centers, managing the availability of servers is often non-trivial, especially when the workload is unpredictable. Using too many servers would waste energy, while using too few would affect the performance. A recent theoretical study, which assumes the clairvoyant model where job size is known at arrival time, has successfully integrated sleep-and-wakeup management into multi-processor job scheduling and obtained a competitive tradeoff between flow time and energy [6]. This paper extends the study to the nonclairvoyant model where the size of a job is not known until the job is finished. We give a new online algorithm SATA which is, for any $\epsilon > 0$, $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for the objective of minimizing the sum of flow time and energy.

SATA also gives a new nonclairvoyant result for the classic setting where all processors are always on and the concern is flow time only. In this case, the previous work of Chekuri et al. [7] and Chadha et al. [8] has revealed that random dispatching can give a non-migratory algorithm that is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive, and any deterministic non-migratory algorithm is $\Omega(\frac{m}{s})$-competitive using s-speed processors [7], where m is the number of processors. SATA, which is a deterministic algorithm migrating each job at most four times on average, has a competitive ratio of $O(\frac{1}{\epsilon^2})$. The number of migrations used by SATA is optimal up to a constant factor as we can extend the above lower bound result.

*The research was supported by HKU-SPF 201109176197.

†Part of the work was done when working in MADALGO, Aarhus University, Denmark.

SPAA'13, July 23–25, 2013, Montréal, Québec, Canada.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance Attributes; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General

Keywords

Online scheduling; competitive analysis; sleep management; flow time; job migration

1. INTRODUCTION

Energy consumption is a major concern for large-scale data centers. Very often data centers are running more servers than necessary and wasting a lot of energy. When a processor is on, the power consumption is divided into *dynamic power* and *static power*; the former is consumed only when the processor is processing a job, while the latter is consumed constantly (due to leakage current) even when the machine is idle. For example, an Intel Xeon E5320 server requires 150W of power when idling and 240W when working [11]. The static power consumption is cut off only when a processor is put to sleep. From the energy viewpoint, a data center should let the servers sleep whenever they are idle; yet waking up the servers later requires extra energy, and it is energy inefficient to frequently switch servers on and off. It is nontrivial how to determine dynamically the appropriate number of working servers so as to strike a balance between energy usage and quality of service (QoS), especially when the workload is unpredictable. The past few years have witnessed a number of theoretical results on revisiting different scheduling problems to consider sleep management, QoS and energy consumption together (e.g., [6,14–16,18], see the survey [1]).

Multi-processor flow-time scheduling. A well-studied QoS measurement for job scheduling is the total flow time. The flow time (or simply the flow) of a job is the length of the duration from its arrival until its completion. We consider the online setting where jobs arrive at unpredictable times. There are $m > 1$ identical processors. Jobs are sequential (i.e., each can be executed on at most one processor at a time) and preemptive in nature. Flow-time scheduling on

multiple processors is hard; competitive online algorithms can exist only when extra resource is given [17].

- For the clairvoyant model where job size is known at release time, it is known that both migratory and non-migratory algorithms can be $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for flow time [3, 10].[1]
- From the viewpoint of operating systems, it is more natural to consider the nonclairvoyant model where the size of a job is not known until the job is finished. To minimize flow time on multi-processors, extra speed is not too useful if migration is not allowed; Chekuri et al. [7] showed that any deterministic non-migratory online algorithm that dispatches jobs at arrival times is $\Omega(\frac{m}{s})$-competitive when using s-speed processors. They also showed a randomized non-migratory algorithm that dispatches jobs randomly and is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^4}\log\frac{1}{\epsilon})$-competitive. Chadha et al. [8] improved the analysis of random dispatching and the competitive ratio to $O(\frac{1}{\epsilon^3})$. The recent work of Gupta et al. [13] on the other hand implies a deterministic migratory algorithm that is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive; this algorithm migrates jobs after every infinitesimal time interval and requires unbounded number of migrations. Such migratory algorithm makes sense for multi-processors on a single chip, but may not be practical for a cluster of servers.

Sleep management and energy. We assume that a processor has two possible states, *awake* or *sleep*. When a processor is awake, it can process a job with energy consumed at the rate $\nu+\sigma$, where $\nu>0$ is the dynamic power and $\sigma>0$ is the static power. An awake processor can be idle, requiring only the static power σ. A processor can enter a sleep state to reduce its power consumption to zero, but a wake-up operation requires extra energy $\omega>0$. A scheduler has to determine when and which processors should be put to sleep or waken up, and how the jobs are scheduled on the awake processors. Following the literature on optimizing flow time and energy, we consider the objective of minimizing the sum of flow time and energy (or in general, a linear combination of flow and energy)[2] [2]. Under the clairvoyant model, Chan et al. [6] have given a non-migratory algorithm that is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for flow time plus energy. They also showed that without faster processors, even a migratory algorithm is $\Omega(m)$-competitive.

Our contribution. Our main result is a nonclairvoyant algorithm called SATA (Scheduling with Arrival-Time-Alignment) for job scheduling with sleep management. SATA is a deterministic algorithm and $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for the objective of minimizing flow time plus energy.

SATA also gives a new nonclairvoyant result for the classic setting where all processors are always on and the objective is to minimize flow time only. In this case, SATA is $(1+\epsilon)$-speed $8(1+\frac{1}{\epsilon})^2$-competitive. It is interesting to compare SATA with existing nonclairvoyant algorithms. SATA is a deterministic migratory algorithm that guarantees each job being migrated at most 4 times on average. It has a better competitive ratio than the previous randomized non-migratory algorithm [8] and deterministic unbounded-migratory algorithm [13]. We also extend the lower bound result in [7] to show that if a job is allowed to migrate at most c times on average for any real $c<1$, any deterministic nonclairvoyant s-speed algorithm for $s\geq 1$ has a competitive ratio $\Omega(\min(\frac{m}{s}, \frac{1}{\sqrt{cs}}))$. This lower bound holds even if jobs can be dispatched at any time.

Remarks on speed scaling. Another model for studying energy saving is the speed scaling model, in which a processor can dynamically scale its speed, and the power increases with the speed s according to a given power function $P(s)$. Online algorithms for minimizing flow time plus energy under the speed scaling model (without sleep management) have been studied extensively (e.g., [2, 4, 5, 12, 13], see [1] for a survey). In particular, the nonclairvoyant speed-scaling algorithm by Gupta et al. [13] is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^5})$-competitive. This algorithm requires unbounded number of migrations. We can extend SATA to support speed scaling (instead of sleep management) using some standard techniques [5, 12]; the resulting algorithm is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for flow plus energy and migrates each job at most 4 times on average. It is worth-mentioning that SATA assumes each processor to have the same speed-to-power function, while Gupta et al.'s algorithm [13] allows processors to be all different.

It is natural to consider a more general energy-saving model that exploits both speed scaling and sleep management. It is open how to adapt SATA or find another nonclairvoyant algorithm for this model. If clairvoyance is allowed, Chan et al. [6] have already given a non-migratory competitive algorithm that can exploit both sleep management and speed scaling for energy saving.

A glimpse of SATA. SATA processes a portion of the latest-arrived jobs (an idea from the parallel-job-algorithm[3] LAPS [9]), and runs k of them on each processor. The parameter k changes over time. We consider sequential jobs which can be executed on at most one processor at a time, so we may need to redistribute the jobs via migrations whenever the set of active jobs changes. This may appear to require a lot of migrations as each processor has to run exactly k of the km latest arrived jobs. Nevertheless, we derived a mechanism called the arrival-time-alignment property to ensure SATA can maintain this even job distribution by migrating at most two jobs when a job arrives or finishes. We can then analyze the SATA's competitiveness by a rather standard technique of potential analysis on the total flow-time incurred by SATA and the optimal algorithm OPT.

When processors can sleep, we extend SATA using two simple concepts: (1) use total flow time to trigger the next processor to wake up; and (2) use total idling energy to determine when to put an idle processor to sleep. The non-trivial part is extending the potential analysis. In our analysis we need to take into account the number of processors available to OPT. Without sleep management, this number

[1] An online algorithm A is said to be s-speed c-competitive if A's performance is at most c times the optimal offline algorithm OPT's performance when A is given processors s times faster than OPT's processors.

[2] From an economic point of view, both energy and flow time can be measured in terms of money and it can be assumed that users are willing to spend a certain units of energy for one unit of flow time; thus it makes sense to consider a linear combination of flow time and energy. By scaling the units of energy and time, we can assume they have the same relative weighting.

[3] The workload of a parallel job can be shared by multi-processors in parallel.

is always m. Our first attempt is to consider the number of OPT's awake processors at each time, yet this idea soon proves to be over precise. Our major trick here is to divide the time into intervals, each represents a cycle of SATA in which jobs and processors keep increasing and then both decrease to the bottom. A perhaps counter-intuitive idea here is to use the *maximum* number of OPT's awake processors within the interval, which turns out to be a sufficiently good estimate of OPT's available processors. Note that the potential analysis of the clairvoyant sleep management given in [6] is processor-based; it exploits a potential function that allows the best match-up of processors, then it becomes feasible to account for the progress of each pair of processors. However, such an approach only makes sense for non-migratory schedules.

To minimize flow time and energy on processors with the sleep state, SATA is $(1+\epsilon)$-speed $32(1+\frac{1}{\epsilon})^2$-competitive for flow time plus energy. We can extend the arrival-time-alignment property such that a job is migrated at most $(\ln m + 6)$ times on average. We need more migrations because SATA is always conservative and wakes up processors gradually; thus SATA needs to redistribute jobs to the newly awaken processor and this requires nonconstant number of migrations.

Organization of paper. Section 2 presents the algorithm SATA for minimizing flow time, and the lower bound result is given in Appendix A. Section 3 considers the sleep management model and extends SATA to minimize flow time plus energy. Section 4 discusses the extension to the speed scaling model.

2. SATA FOR NONCLAIRVOYANT FLOW-TIME SCHEDULING

In this section, we assume that all $m \geq 2$ processors are always awake (and ready to work). Jobs are sequential in nature (each can be processed by one processor at a time); preemption and migration are allowed. The objective is to minimize the total flow F, which is the sum over all jobs j of the time elapsed since job j arrives and until it is completed. It is useful to view the total flow as a quantity incurring at a rate equal to the number of active jobs (jobs released but not yet completed); i.e., $F = \int_0^\infty n(t)\,\mathrm{d}t$ where $n(t)$ denotes the number of active jobs at time t. Below is the main result.

THEOREM 1. *For any $\epsilon > 0$,* SATA *is $(1+\epsilon)$-speed $8(1+\frac{1}{\epsilon})^2$-competitive for total flow time, and* SATA *migrates a job at most 4 times on average.*

At any time, let n_{a} be the number of active jobs of SATA. If $n_{\mathrm{a}} < m$, SATA processes each job on a different processor. If $n_{\mathrm{a}} \geq m$, SATA, using the idea of LAPS, processes at least $\lceil \beta n_{\mathrm{a}} \rceil$ jobs that have the latest arrival times, where $\beta < 0.5$ is a constant (defined to be $\frac{\epsilon}{4(1+\epsilon)}$). Unlike LAPS, SATA exercises a tight control to require each of the m processors to run exactly k jobs, where $k \geq 1$ is the smallest integer such that $km \geq \lceil \beta n_{\mathrm{a}} \rceil$ (note that $\beta < 0.5$ and thus $km \leq n_{\mathrm{a}}$). In other words, SATA needs a nonclairvoyant technique that can evenly distribute the km latest-arrival jobs among the processors without migrating the jobs too often. This technique is rooted at the following property of job distribution when $n_{\mathrm{a}} \geq m$.

Arrival-time-alignment property. Let $j_1, j_2, \ldots, j_{n_{\mathrm{a}}}$ be all the active jobs ordered in ascending order of arrival times. Define the *tail* jobs $J_{\mathrm{tail}} = \{j_\tau, \ldots, j_{n_{\mathrm{a}}}\}$ to be the $\lceil \beta n_{\mathrm{a}} \rceil$ latest-arrival jobs (i.e., $\tau = n_{\mathrm{a}} - \lceil \beta n_{\mathrm{a}} \rceil + 1$). The following definitions are sensitive to the alignment width, defined to be the number of available processors, which always equals m in this section (Section 3 will consider arbitrary width $m_a \leq m$ and we need to replace m with m_a in the definitions). Define the *aligned jobs*, denoted J_{aln}, to be the m jobs (if exist) that arrived just before the tail jobs, i.e., $J_{\mathrm{aln}} = \{j_{\max(1,\tau-m)}, \ldots, j_{\tau-1}\}$. Finally, let $x \in [0, m-1]$ be an integer such that $\lceil \beta n_{\mathrm{a}} \rceil + x$ is a multiple of m. We call $j_{\tau-x}, \ldots, j_{\tau-1}$ the *boundary* jobs, denoted J_{bd}, which must all exist (because $\beta < 0.5$ and $\lceil \beta n_{\mathrm{a}} \rceil + x \leq n_{\mathrm{a}}$). A boundary job is also an aligned job. We say that the current job distribution satisfies the *arrival-time-alignment property* if the following conditions hold. Recall that k is the smallest integer such that $km \geq \lceil \beta n_{\mathrm{a}} \rceil$.

- C1. Every aligned job is with a distinct processor.
- C2. Every processor hosting a boundary job must have exactly $k-1$ tail jobs.
- C3. Every other processor must have exactly k tail jobs.

Figure 1 gives an example of these concepts. With the arrival-time-alignment property, SATA can let each processor run the boundary job (if present) and all the tail jobs. Over all processors, SATA is running x boundary jobs and $\lceil \beta n_{\mathrm{a}} \rceil$ tail jobs, which together are the km latest-arrival jobs.

When $n_{\mathrm{a}} < m$, SATA processes each job on a different processor, and dispatches any new job to a processor without jobs, so no job migration is needed when a job arrives or is completed. Once $n_{\mathrm{a}} = m$, there should be exactly one job in each processor and the arrival-time-alignment property is immediately satisfied with $\lceil \beta n_{\mathrm{a}} \rceil$ tail jobs and $m - \lceil \beta n_{\mathrm{a}} \rceil$ boundary jobs. The arrival-time-alignment property will remain satisfied until a job arrives or is completed. Below we show that at most 2 migrations are needed to maintain the arrival-time-alignment property for each job arrival or completion. Suppose a job arrives or is completed at time t. Define $M_{<t}(j)$ or simply $M(j)$ to be the processor hosting job j just before time t (note that j may migrate to another processor at t). Below n_{a} and x also refer to the numbers just before t.

Dispatching and migration procedure due to job arrival: When a new job j_{NEW} arrives, the number of active jobs increases from n_{a} to $n_{\mathrm{a}}+1$. By definition, the number of tail jobs (i.e., $|J_{\mathrm{tail}}|$) becomes $\lceil \beta(n_{\mathrm{a}}+1) \rceil$, which is equal to $\lceil \beta n_{\mathrm{a}} \rceil + 1$ or $\lceil \beta n_{\mathrm{a}} \rceil$. To maintain the arrival-time-alignment property, we dispatch j_{NEW} and migrate at most two jobs as follows.

- Case 1: $|J_{\mathrm{tail}}|$ increases by 1 (i.e., $\lceil \beta(n_{\mathrm{a}}+1) \rceil = \lceil \beta n_{\mathrm{a}} \rceil + 1$). By definition, J_{aln} is unchanged (and C1 holds), and $|J_{\mathrm{bd}}|$ becomes $x-1$ (precisely, $(x-1) \mod m$). If $x \geq 1$, dispatch j_{NEW} to processor $M(j_{\tau-x})$; if $x = 0$, dispatch j_{NEW} to $M(j_{\tau-m})$. Then C2 and C3 hold.
- Case 2: $|J_{\mathrm{tail}}|$ does not increase (i.e., $\lceil \beta(n_{\mathrm{a}}+1) \rceil = \lceil \beta n_{\mathrm{a}} \rceil$). J_{aln} becomes $\{j_{\tau-m+1}, \ldots, j_\tau\}$, and j_τ replaces $j_{\tau-m}$ as an aligned job, and j_τ also replaces $j_{\tau-x}$ as a boundary job. To maintain C1, we migrate j_τ to processor $M(j_{\tau-m})$ as the new aligned job there. To maintain C2 and C3, we migrate an arbitrary tail job from $M(j_{\tau-m})$ to $M(j_\tau)$ (i.e., j_τ's hosting processor before j_{NEW} arrives), and dispatch j_{NEW} to $M(j_{\tau-x})$.

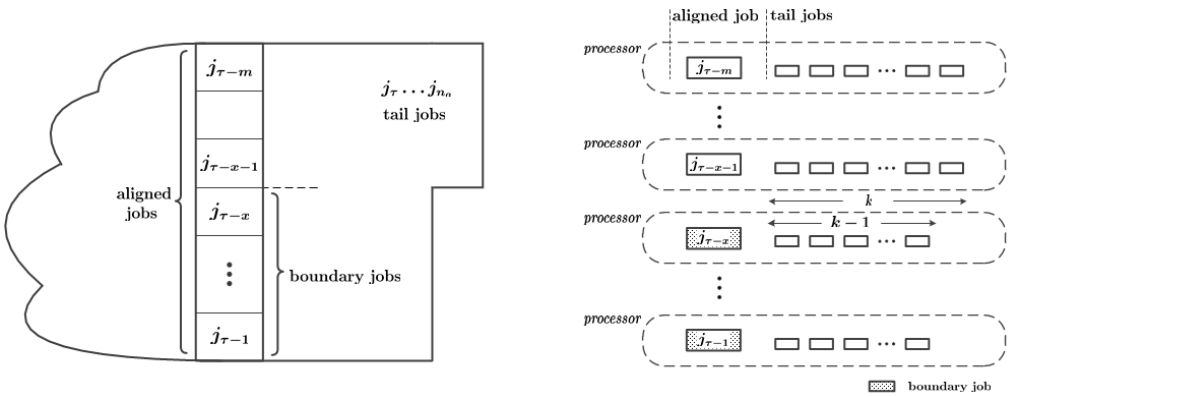

Figure 1: Arrival-time-alignment property

Migration procedure due to job completion: SATA only executes boundary or tail jobs. When a job j_{OLD} is completed, the number of active jobs becomes $n_a - 1$. We focus on the case $n_a - 1 \geq m$ (otherwise we stop maintaining the arrival-time-alignment property and each processor is currently hosting zero or one active job). By definition, $|J_{\text{tail}}|$ becomes $\lceil \beta(n_a - 1) \rceil$.

- Case 1: $|J_{\text{tail}}|$ decreases by 1. By definition, $|J_{\text{bd}}|$ becomes $x + 1$ (precisely, $(x + 1) \mod m$).
  (i) If j_{OLD} is a tail job, J_{aln} is unchanged (and C1 holds), and $j_{\tau-x-1}$ becomes a boundary job. To maintain C2 and C3, we migrate an arbitrary tail job from $M(j_{\tau-x-1})$ to $M(j_{\text{OLD}})$.
  (ii) If j_{OLD} is an aligned job, we make j_τ a new aligned job and migrate j_τ to $M(j_{\text{OLD}})$. Then C1 holds. To maintain C2 and C3, we migrate an arbitrary tail job from $M(j_{\tau-x-1})$ to $M(j_\tau)$.
- Case 2: $|J_{\text{tail}}|$ does not decrease. In this case, $j_{\tau-m-1}$ (if exists) should become a new aligned job, and we migrate $j_{\tau-m-1}$ to maintain C1, as follows. If j_{OLD} is a tail job, then $j_{\tau-1}$ becomes a tail job, and we migrate $j_{\tau-m-1}$ to processor $M(j_{\tau-1})$. Otherwise, j_{OLD} is an aligned job, and we migrate $j_{\tau-m-1}$ to processor $M(j_{\text{OLD}})$. To maintain C2 and C3, we migrate a tail job from $M(j_{\tau-x-1})$ to $M(j_{\text{OLD}})$.

LEMMA 2. *When a job arrives or is completed, it takes at most 2 migrations to maintain the arrival-time-alignment property.*

Algorithm SATA. We are ready to define SATA. Below n_a denotes the current number of active jobs (just before a job arrival or completion occurs). Whenever $n_a \geq m$, SATA maintains the arrival-time-alignment property.

Job arrival: When a job j arrives, if $n_a < m$, dispatch j to a processor without jobs; otherwise, dispatch j and migrate at most 2 jobs according to the job arrival procedure described above.

Job completion: When a job is completed, if $n_a \leq m$, no migration is needed; otherwise, migrate at most 2 jobs according to the job completion procedure described above.

Job scheduling:

- If $n_a < m$, each active job is processed on a different processor;
- If $n_a \geq m$, each processor shares its processing time equally among all its tail jobs and its boundary job (if present). Note that the m processors together are processing km latest-arrival jobs, where k is the smallest integer with $km \geq \lceil \beta n_a \rceil$.

Analysis of flow time. To analyze the flow time of SATA, we adapt the potential function analysis for LAPS [9]. The crux of the analysis is to define a suitable potential function $\Phi(t)$ that captures the difference of progress between SATA and OPT. Let $n_a(t)$ and $n_o(t)$ be the number of active jobs in SATA and OPT, respectively. At any time t, denote the active jobs in SATA as $j_1, j_2, \ldots, j_{n_a(t)}$, arranged in ascending order of release times. For each job j_i, let $q_a(j_i, t)$ and $q_o(j_i, t)$ be the remaining work of job j in SATA and OPT, respectively, and let $x_i = \max\{q_a(j_i, t) - q_o(j_i, t), 0\}$ which is the amount of work of j_i in SATA that is lagging behind OPT. We say a job j_i is *lagging* if $x_i > 0$. We define the potential function

$$\Phi(t) = \frac{2}{\epsilon} \cdot \sum_{i=1}^{n_a(t)} \max\left(1, \tfrac{i}{m}\right) \cdot x_i \ .$$

The definition of $\Phi(t)$ is similar to that of LAPS [9]; yet we give each job j_i the coefficient $\max(1, \frac{i}{m})$ instead of simply i as in [9]. This change is mainly because SATA is dealing with sequential jobs, while LAPS is dealing with parallel jobs that can be shared by multi-processors in parallel. Note that when $n_a < m$, SATA cannot fully utilize all the m processors as in LAPS. Furthermore, SATA is not running exactly $\lceil \beta n_a(t) \rceil$ jobs, the actual fraction varies over time which further complicates the analysis. When $t = 0$, $\Phi(t) = 0$. When a job arrives or is completed, Φ does not increase. Let $F(t)$ and $F^*(t)$ be the total flow incurred up to time t in SATA and OPT, respectively.

In the rest of this section we prove the following *running condition*, which relates the rate of increase of $F(t)$ to that of $F^*(t)$. Then Theorem 1 follows from integrating the running condition over time.

LEMMA 3. *At any time t when there is no job arrival or completion, $\frac{\mathrm{d}F(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} \leq 8(1 + \frac{1}{\epsilon})^2 \cdot \frac{\mathrm{d}F^*(t)}{\mathrm{d}t}$.*

PROOF. Consider a particular time t with no job arrival or completion by either SATA or OPT. For convenience, we drop t from all the notations. We divide the analysis into cases depending on n_a and m. In each case, $\frac{\mathrm{d}F}{\mathrm{d}t} = n_a$

and $\frac{\mathrm{d}F^*}{\mathrm{d}t} = n_\mathrm{o}$. To bound the rate of change of Φ, we will consider how Φ changes in an infinitesimal amount of time (from t to $t + \mathrm{d}t$). We may consider the rate of change of Φ due to SATA and OPT separately, which are denoted by $\frac{\mathrm{d}\Phi_\mathrm{a}}{\mathrm{d}t}$ and $\frac{\mathrm{d}\Phi_\mathrm{o}}{\mathrm{d}t}$, respectively. Note that $\frac{\mathrm{d}\Phi}{\mathrm{d}t} = \frac{\mathrm{d}\Phi_\mathrm{a}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi_\mathrm{o}}{\mathrm{d}t}$. We call the term $\max\left(1, \frac{i}{m}\right)$ in the potential Φ the *coefficient* of job j_i. We also define a ratio ϕ such that SATA is currently processing ϕn_a lagging jobs.

Case 1: $n_\mathrm{a} < m$. By definition, SATA is processing each active job on a different processor at speed $(1 + \epsilon)$, while OPT can process each job at speed at most 1. Therefore, for any lagging job j_i (i.e., $x_i > 0$), x_i is changing at rate at most $-(1 + \epsilon) + 1 = -\epsilon$; for any non-lagging job $j_{i'}$, $x_{i'}$ remains zero. We have $\frac{\mathrm{d}\Phi}{\mathrm{d}t} \le \frac{2}{\epsilon} \cdot \sum_{i:x_i>0} \max\left(1, \frac{i}{m}\right) \cdot (-\epsilon) = -2\sum_{i:x_i>0} 1 = -2\phi n_\mathrm{a}$. There are $n_\mathrm{a} - \phi n_\mathrm{a}$ non-lagging active jobs, which must also be active in OPT, so $n_\mathrm{o} \ge n_\mathrm{a} - \phi n_\mathrm{a}$. Then, $\frac{\mathrm{d}F}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \le n_\mathrm{a} - 2\phi n_\mathrm{a} \le n_\mathrm{o} = \frac{\mathrm{d}F^*}{\mathrm{d}t} \le 8(1 + \frac{1}{\epsilon})^2 \cdot \frac{\mathrm{d}F^*}{\mathrm{d}t}$.

Case 2: $n_\mathrm{a} \ge m$. In this case, SATA is processing km ($k \ge 1$) latest-arrival jobs. We use L to denote these jobs. Since there are $(km - \phi n_\mathrm{a})$ non-lagging jobs in L, which must be active in OPT, we have $n_\mathrm{o} \ge km - \phi n_\mathrm{a}$. Recall that $\beta = \frac{\epsilon}{4(1+\epsilon)}$. We further consider two subcases depending on whether $n_\mathrm{a} \le \frac{1}{2\beta} m$.

Case 2a: $n_\mathrm{a} \le \frac{1}{2\beta} m$. In this case, $k = 1$, and thus SATA is running each job in L on a different processor at speed $(1+\epsilon)$. Then, $\frac{\mathrm{d}\Phi_\mathrm{a}}{\mathrm{d}t} \le \frac{2}{\epsilon} \cdot \sum_{i\in L:x_i>0} \max(1, \frac{i}{m}) \cdot (-(1+\epsilon)) \le \frac{2}{\epsilon} \cdot \phi n_\mathrm{a} \cdot (-(1+\epsilon)) = -\frac{\phi}{2\beta} n_\mathrm{a}$. OPT is processing at most n_o jobs, each at speed 1. Since the coefficient of j_{n_a} is the largest among all j_i's, we have $\frac{\mathrm{d}\Phi_\mathrm{o}}{\mathrm{d}t} \le \frac{2}{\epsilon} \cdot \frac{n_\mathrm{a}}{m} \cdot n_\mathrm{o} \le \frac{2}{\epsilon} \cdot \frac{1}{2\beta} \cdot n_\mathrm{o} = \frac{1}{\epsilon\beta} n_\mathrm{o}$. Thus,

$$\begin{aligned}\frac{\mathrm{d}F}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} &\le n_\mathrm{a} - \tfrac{\phi}{2\beta} n_\mathrm{a} + \tfrac{1}{\epsilon\beta} n_\mathrm{o} \\ &\le \tfrac{1}{2\beta}(km - \phi n_\mathrm{a}) + \tfrac{1}{\epsilon\beta} n_\mathrm{o} \quad (\text{as } n_\mathrm{a} \le \tfrac{1}{2\beta} m \text{ and } k = 1) \\ &\le \tfrac{1}{2\beta} n_\mathrm{o} + \tfrac{1}{\epsilon\beta} n_\mathrm{o} \le 8(1 + \tfrac{1}{\epsilon})^2 \cdot \tfrac{\mathrm{d}F^*}{\mathrm{d}t} \ .\end{aligned}$$

Case 2b: $n_\mathrm{a} > \frac{1}{2\beta} m$. In this case, each processor in SATA is running k jobs, each at speed $\frac{1+\epsilon}{k}$. Thus,

$$\begin{aligned}\tfrac{\mathrm{d}\Phi_\mathrm{a}}{\mathrm{d}t} &\le \tfrac{2}{\epsilon} \cdot \textstyle\sum_{j_i\in L:x_i>0} \max(1, \tfrac{i}{m}) \cdot \tfrac{-(1+\epsilon)}{k} \\ &\le -2(1 + \tfrac{1}{\epsilon}) \cdot \tfrac{1}{k} \cdot \textstyle\sum_{j_i\in L:x_i>0} \tfrac{i}{m} \\ &\le -2(1 + \tfrac{1}{\epsilon}) \cdot \tfrac{1}{k} \cdot \phi n_\mathrm{a}(\tfrac{n_\mathrm{a} - km}{m}) \\ &\qquad (\text{since any } j_i \in L \text{ has } i > n_\mathrm{a} - km) \\ &= -2(1 + \tfrac{1}{\epsilon}) \cdot \phi n_\mathrm{a}(\tfrac{n_\mathrm{a}}{km} - 1) = -\tfrac{\phi}{2\beta} n_\mathrm{a}(\tfrac{n_\mathrm{a}}{km} - 1) \ .\end{aligned}$$

OPT has m processors of speed 1, and the coefficient of j_{n_a} is the largest among all j_i's. We have $\frac{\mathrm{d}\Phi_\mathrm{o}}{\mathrm{d}t} \le \frac{2}{\epsilon} \cdot \frac{n_\mathrm{a}}{m} \cdot m \le \frac{2}{\epsilon} n_\mathrm{a}$. Hence,

$$\begin{aligned}\frac{\mathrm{d}F}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} &\le n_\mathrm{a} - \tfrac{\phi}{2\beta} n_\mathrm{a}(\tfrac{n_\mathrm{a}}{km} - 1) + \tfrac{2}{\epsilon} n_\mathrm{a} \\ &\le n_\mathrm{a} + \tfrac{\phi}{2\beta} n_\mathrm{a} - \tfrac{\phi}{2\beta} n_\mathrm{a}(\tfrac{n_\mathrm{a}}{km}) + \tfrac{2}{\epsilon} n_\mathrm{a} \ .\end{aligned}$$

We now show that $\phi < 2\beta$ as follows. (1) If $n_\mathrm{a} \le \frac{1}{\beta} m$, then $k = 1$ and $\phi n_\mathrm{a} \le km = m$. Since $n_\mathrm{a} > \frac{1}{2\beta} m$, we have $\phi n_\mathrm{a} \le m < 2\beta n_\mathrm{a}$ and $\phi < 2\beta$. (2) If $n_\mathrm{a} > \frac{1}{\beta} m$, then $\phi n_\mathrm{a} \le km < \beta n_\mathrm{a} + m < 2\beta n_\mathrm{a}$ and hence $\phi < 2\beta$.

In conclusion,

$$\begin{aligned}\frac{\mathrm{d}F}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} &\le n_\mathrm{a} + n_\mathrm{a} - \tfrac{\phi}{2\beta} n_\mathrm{a}(\tfrac{n_\mathrm{a}}{km}) + \tfrac{2}{\epsilon} n_\mathrm{a} \qquad (\text{since } \phi < 2\beta) \\ &= \tfrac{1}{2\beta} n_\mathrm{a} - \tfrac{\phi}{2\beta} n_\mathrm{a}(\tfrac{n_\mathrm{a}}{km}) = \tfrac{1}{2\beta}(km - \phi n_\mathrm{a}) \cdot (\tfrac{n_\mathrm{a}}{km}) \\ &\le \tfrac{1}{2\beta^2}(km - \phi n_\mathrm{a}) \qquad (\text{as } km \ge \lceil \beta n_\mathrm{a} \rceil \Rightarrow \tfrac{n_\mathrm{a}}{km} \le \tfrac{1}{\beta}) \\ &\le 8(1 + \tfrac{1}{\epsilon})^2 \cdot n_\mathrm{o} = 8(1 + \tfrac{1}{\epsilon})^2 \cdot \tfrac{\mathrm{d}F^*}{\mathrm{d}t} \ . \qquad \square\end{aligned}$$

3. SATA WITH SLEEP MANAGEMENT

This section considers scheduling in the sleep management model. In this model, a processor is in either the *awake* state or the *sleep* state. Only when it is awake, it can process a job and the energy is consumed at the rate $\nu + \sigma$, where $\nu > 0$ is the dynamic power and $\sigma > 0$ is the static power. An awake processor can be idle (i.e., not processing any job) and only requires the static power σ. A processor can enter the sleep state to reduce the power to zero. Initially, all processors are in the sleep state. Following the literature, we assume that state transition is immediate but requires energy. A *wake-up* from the sleep state requires ω units of energy, and the reverse takes zero.

This section shows how to extend SATA to handle sleep management of $m \ge 2$ processors and job scheduling on a variable number of processors. The aim is to minimize the total flow time F plus energy E. Intuitively, SATA has to maintain an appropriate number of awake processors and strike a balance between flow time and energy. We assume that SATA is using $(1+\epsilon)$-speed processors for $\epsilon > 0$, which are $(1 + \epsilon)$ times faster than the optimal offline algorithm OPT while using the same power. Below is our main result.

THEOREM 4. *For any $\epsilon > 0$,* SATA *is $(1+\epsilon)$-speed $32(1+\frac{1}{\epsilon})^2$-competitive for flow plus energy, and* SATA *migrates a job at most $(\ln m + 6)$ times on average.*

SATA uses two simple ideas to determine the appropriate number of awake processors: (1) total flow for waking up processors; and (2) total idling energy for putting processors to sleep. More specifically, SATA maintains two (real-value) counters C_f and C_e to keep track of the accumulated flow and idling energy, respectively (see the algorithm below for precise definition). We view them as two competing quantities: If C_f reaches ω first, we wake up a sleeping processor; otherwise, when C_e reaches ω, we put one awake processor to sleep. In either case, both counters are reset to 0 and the process is restarted. The idea looks simple, yet the potential analysis (of flow time plus energy) is more complicated than before and triggers us to come up with new insight.

With proper sleep management, SATA can then handle job dispatching, migrations and scheduling in the same way as in the previous section, except that the decisions are made with reference to m_a, the current number of awake processors, instead of m, the maximum number of processors. Let n_a denote the current number of active jobs. When $n_\mathrm{a} < m_\mathrm{a}$, SATA processes each job on a different awake processor, and dispatches any new job to an awake processor without jobs. In this case, SATA never migrates a job even if it wakes up a processor or puts it to sleep.

When $n_\mathrm{a} \ge m_\mathrm{a}$, SATA schedules the km_a latest-arrival jobs evenly on the m_a awake processors, where k is the smallest integer such that $km_\mathrm{a} \ge \lceil \beta n_\mathrm{a} \rceil$. SATA maintains the arrival-time-alignment property with alignment width m_a. This ensures that in each processor, the tail jobs and the boundary job (if present) sum up to k. This property remains satisfied over a maximal period in which m_a

remains unchanged and there are at least m_a active jobs. Within this period, for each job arrival and completion, SATA uses the procedures in Section 2 to maintain the arrival-time-alignment property (with alignment width m_a) using at most 2 migrations.

Migration due to a wake-up: When SATA wakes up a processor, the number of awake processors increases from m_a to m_a+1. If n_a remains bigger than m_a+1, the arrival-time-alignment property must be restored with alignment width m_a+1. The wake-up does not change the active jobs and tail jobs. Denote the active jobs as $j_1, \cdots, j_{n_a}$, arranged in ascending order of arrival times. Since alignment width increases to m_a+1, we need one more aligned job, which is job $j_{\tau-m_a-1}$ (if exists) where $\tau = n_a - \lceil \beta n_a \rceil + 1$. To maintain C1, we send $j_{\tau-m_a-1}$ to the new processor; to maintain C2 and C3, we migrate $\lceil \lceil \beta n_a \rceil / (m_a+1) \rceil$ tail jobs from existing awake processors to the new processor.

The migration due to a wake-up can involve a lot of jobs. Nevertheless, a slightly tricky analysis shows that each job on average can be migrated $(\ln m + 2)$ times due to wake-up operations (see Lemma 14 in Appendix B). On the other hand, SATA is defined such that whenever $n_a \geq m_a$, SATA is not allowed to put an idle processor to sleep (see the algorithm below); thus, the number of awake processors and the alignment width will not decrease and cause any migration.

Algorithm Extended SATA. Below we give the details of extending SATA. Initially, all m processors are sleeping. Whenever $n_a \geq m_a$, SATA maintains the arrival-time-alignment property with alignment width m_a. Recall that σ and ω denote the static power and wake-up energy.

Wake up a processor: When $m_a < m$, increase C_f at rate of n_a; when $C_f = \omega$, wake up a sleeping processor, migrate jobs according to the wake-up procedure described above, and reset $C_f = C_e = 0$.

Put a processor to sleep: When $n_a < m_a$, increase C_e at rate of σ times the number of idle processors, i.e., $\sigma(m_a - n_a)$; when $C_e = \omega$, put an awake (idle) processor to sleep and reset $C_f = C_e = 0$.

Job arrival: When a job j arrives, if $n_a < m_a$, dispatch j to a processor without jobs; otherwise, dispatch j and migrate at most 2 jobs according to the job arrival procedure described above.

Job completion: When a job is completed, if $n_a \leq m_a$, no migration is needed; otherwise, migrate at most 2 jobs according to the job completion procedure described above.

Job scheduling:

- If $n_a < m_a$, each active job is processed on a different awake processor;
- If $n_a \geq m_a$, each awake processor shares its processing time equally among all its tail jobs and its boundary job (if present); note that the m_a processors together are processing the km_a latest-arrival jobs, where k is the smallest integer with $km_a \geq \lceil \beta n_a \rceil$.

It remains to show the competitive ratio of SATA stated in Theorem 4. Consider a schedule of SATA. We divide SATA's energy usage E into three parts: E_I is the *idling* energy (static energy incurred by processors when they are awake but idle), E_W the *working* energy (both dynamic and static energy incurred by processors when they are working on jobs), and U the wake-up energy. Denote F as SATA's flow, and define SATA's total cost $G = F + E_W + E_I + U$. These notations of cost and energy are used for OPT in the same way, but marked with an asterisk. By the definition of SATA, we can upper bound SATA's energy by SATA's flow F and OPT's working energy E_W^*.

LEMMA 5. *(i) $U \leq F$; (ii) $E_I \leq 2U$; (iii) $E_W \leq E_W^*$.*

PROOF. (i) By definition, SATA wakes up a processor when C_f accumulates ω units of total flow, so the number of wake-ups times ω is at most the total flow F, i.e., $U \leq F$.

(ii) By definition, when SATA puts a processor to sleep, C_e accumulates ω units of idling energy; when SATA wakes up a processor, C_e accumulates at most ω units of idling energy before being reset to 0. Any idling energy would have been counted in C_e, so E_I is at most ω times the number of wake-up and sleep events. Note that all processors are asleep initially and after all jobs are completed. Thus, the number of sleep events equals the number of wake-ups and hence $E_I \leq 2U$.

(iii) Any two algorithms using processors of the same speed incur the same amount of working energy for completing all jobs. As SATA is using $(1+\epsilon)$-speed processors, E_W is indeed less than OPT's working energy, i.e., $E_W \leq E_W^*$. □

The above lemma implies that SATA's total cost $G = F + E_W + E_I + U \leq 4F + E_W^*$. The rest of this section is devoted to analyzing the total flow F.

Intuitively, SATA's flow cannot be directly bounded by OPT's total cost. SATA often wakes up fewer processors than OPT and thus incurs more flow time. Such excess in flow is not related to OPT. To quantify it, our first attempt is to consider the flow incurred when SATA has fewer awake processors than OPT, yet this idea soon proves to be over precise. The number of awake processors in OPT can vary frequently. If SATA wakes up processors too slowly, then most of the excess in flow is incurred later when enough jobs have been accumulated. By that time, OPT may have less awake processors than SATA. Our major trick here is to give up an accurate accounting of OPT's number of processors. Instead we divide the time into some special intervals called S-interval. Roughly speaking, each S-interval represents a cycle of SATA in which jobs and processors keep increasing and then both decrease to the bottom. We use the maximum number of OPT's awake processors within the interval, which is more well-behaved and turns out to be a sufficiently good estimate of OPT's resource within the interval. We define *lazy flow* to be the flow incurred when SATA has fewer awake processors than such maximum number of OPT's processors (see precise definitions below).

***S*-intervals and lazy flow F_S.** Let $m_a(t)$ and $m_o(t)$ be the number of awake processors in SATA and OPT at time t (after all wake-up and sleep events at time t), respectively. We partition the timeline into intervals called S-intervals as follows. Consider the sequence of wake-up and sleep events in SATA. Each S-interval starts with the last sleep event of the previous S-interval (except that the first starts at time 0), followed by a sequence of wake-ups and and then a maximal sequence of sleep events. By definition, two consecutive S-intervals overlap at one sleep event. For any time t during an S-interval I, we define $m_o^*(t) = \max_{t' \in I} m_o(t')$, which is the maximum number of awake processors used by OPT during I. At time t when two S-intervals overlap, SATA puts exactly one awake processor to sleep (by definition, only one processor of SATA can sleep at a time), and

thus $n_{\rm a}(t) \le m_{\rm a}(t)$. We define *lazy flow* $F_{\rm s}$ to be the total flow incurred at the times t when $m_{\rm a}(t) < m_{\rm o}^*(t)$, i.e., $F_{\rm s} = \int_{t:m_{\rm a}(t)<m_{\rm o}^*(t)} n_{\rm a}(t)\,{\rm d}t$. Then, we can show Lemma 6 below, which involves two potential function analyses.

LEMMA 6. *(i)* $F \le 8(1+\frac{1}{\epsilon})^2 F^* + (1+\frac{2}{\epsilon})F_{\rm s}$*; (ii)* $F_{\rm s} \le 2(E_{\rm w}^* + E_{\rm i}^* + U^*)$.

Lemma 6 implies that $F \le 8(1+\frac{1}{\epsilon})^2 F^* + 2(1+\frac{2}{\epsilon})(E_{\rm w}^* + E_{\rm i}^* + U^*)$. Together with Lemma 5, $G \le 32(1+\frac{1}{\epsilon})^2 F^* + 8(1+\frac{2}{\epsilon})(E_{\rm w}^* + E_{\rm i}^* + U^*) + E_{\rm w}^* \le 32(1+\frac{1}{\epsilon})^2(F^* + E_{\rm w}^* + E_{\rm i}^* + U^*) = 32(1+\frac{1}{\epsilon})^2 G^*$. Theorem 4 follows.

3.1 Potential Analysis of Total Flow F

To show Lemma 6(i), we extend the potential analysis in Section 2 to allow SATA and OPT to use a variable number of processors. The potential function $\Phi(t)$ of Section 2 is modified so that the coefficient takes $m_{\rm a}(t)$ and $m_{\rm o}^*(t)$ into consideration. Recall that SATA's active jobs are arranged in ascending order of arrival times. For the i-th job j_i, we denote x_i as the amount of work of j_i in SATA that is lagging behind OPT (see Section 2 for the definitions). The new potential function is defined as follows:

$$\Phi(t) = \frac{2}{\epsilon} \cdot \sum_{i=1}^{n_{\rm a}(t)} \max\left(1, \frac{i}{\max(m_{\rm a}(t), m_{\rm o}^*(t))}\right) \cdot x_i \ ,$$

where $\max\left(1, \frac{i}{\max(m_{\rm a}(t), m_{\rm o}^*(t))}\right)$ is called the *coefficient* of job j_i.

Initially, $\Phi(0) = 0$. When a job arrives or is completed, Φ does not increase. The way we integrate $m_{\rm a}(t)$ and $m_{\rm o}^*(t)$ into $\Phi(t)$ guarantees that when $m_{\rm a}$ and $m_{\rm o}^*$ changes, Φ cannot increase: When $m_{\rm a}$ increases, the coefficient of any job and thus Φ do not increase. When $m_{\rm a}$ decreases to $m_{\rm a}-1$ or when $m_{\rm o}^*$ changes, SATA puts one awake processor to sleep, which implies $n_{\rm a} \le m_{\rm a} - 1$. The coefficients of all the $n_{\rm a}$ jobs remain exactly 1, so Φ does not change.

Let $F(t)$ be the total flow F incurred up to time t by SATA. Similarly, define $F^*(t)$ and $F_{\rm s}(t)$ for OPT's total flow F^* and SATA's lazy flow $F_{\rm s}$. It remains to show the following running condition. Lemma 6(i) then follows from integrating it over time.

LEMMA 7. *At any time t when there is no job arrival or completion or change on $m_{\rm a}(t)$ or $m_{\rm o}^*(t)$,* $\frac{{\rm d}F(t)}{{\rm d}t} + \frac{{\rm d}\Phi(t)}{{\rm d}t} \le 8(1+\frac{1}{\epsilon})^2 \cdot \frac{{\rm d}F^*(t)}{{\rm d}t} + (1+\frac{2}{\epsilon}) \cdot \frac{{\rm d}F_{\rm s}(t)}{{\rm d}t}$.

PROOF. At time t, let $n_{\rm a}(t)$ and $n_{\rm o}(t)$ be the number of jobs of SATA and OPT, respectively. For convenience, we drop t from all the notations. Note that $\frac{{\rm d}F}{{\rm d}t} = n_{\rm a}$ and $\frac{{\rm d}F^*}{{\rm d}t} = n_{\rm o}$. If $m_{\rm a} < m_{\rm o}^*$, $\frac{{\rm d}F_{\rm s}}{{\rm d}t} = n_{\rm a}$; otherwise, $\frac{{\rm d}F_{\rm s}}{{\rm d}t} = 0$. To bound the rate of change of Φ, we will consider how Φ changes in an infinitesimal amount of time (from t to $t+{\rm d}t$). If $n_{\rm a} < m_{\rm a}$, SATA processes each active job on a different awake processor. Similarly as in the proof of Lemma 3, we can show that $\frac{{\rm d}F(t)}{{\rm d}t} + \frac{{\rm d}\Phi(t)}{{\rm d}t} \le 8(1+\frac{1}{\epsilon})^2 \cdot \frac{{\rm d}F^*(t)}{{\rm d}t}$. If $n_{\rm a} \ge m_{\rm a} \ge m_{\rm o}^*$, $\max(m_{\rm a}, m_{\rm o}^*) = m_{\rm a}$. The number of awake processors in OPT is $m_{\rm o} \le m_{\rm o}^*$, which is at most $m_{\rm a}$. Since only the processing of OPT can increase Φ, the worst case is that OPT is also using $m_{\rm a}$ awake processors as SATA. Then, similarly as in the proof of Lemma 3, we can show that $\frac{{\rm d}F(t)}{{\rm d}t} + \frac{{\rm d}\Phi(t)}{{\rm d}t} \le 8(1+\frac{1}{\epsilon})^2 \cdot \frac{{\rm d}F^*(t)}{{\rm d}t}$.

It remains to consider $n_{\rm a} \ge m_{\rm a}$ and $m_{\rm o}^* > m_{\rm a}$. In this case, $\frac{{\rm d}F_{\rm s}}{{\rm d}t} = n_{\rm a}$. Since the processing of SATA can only decrease Φ, it suffices to consider the change of Φ due to the processing of OPT. (Case 1) $n_{\rm a} \ge m_{\rm o}^*$: Since OPT has $m_{\rm o}$ awake processors of speed 1, and the coefficient of $j_{n_{\rm a}}$ is the largest among all j_i's, we have $\frac{{\rm d}\Phi}{{\rm d}t} \le \frac{2}{\epsilon}\max(1, \frac{n_{\rm a}}{\max(m_{\rm a}, m_{\rm o}^*)}) \cdot m_{\rm o} = \frac{2}{\epsilon}\frac{n_{\rm a}}{m_{\rm o}^*} \cdot m_{\rm o} \le \frac{2}{\epsilon} n_{\rm a}$. (Case 2) $n_{\rm a} < m_{\rm o}^*$: For any j_i, its coefficient is $\max(1, \frac{i}{\max(m_{\rm a}, m_{\rm o}^*)}) = 1$. Since OPT can process at most $n_{\rm a}$ jobs which are also active in SATA, and the speed for each job is at most 1, we have $\frac{{\rm d}\Phi}{{\rm d}t} \le \frac{2}{\epsilon} n_{\rm a}$. In both cases, $\frac{{\rm d}F}{{\rm d}t} + \frac{{\rm d}\Phi}{{\rm d}t} \le n_{\rm a} + \frac{2}{\epsilon} n_{\rm a} = (1+\frac{2}{\epsilon}) \cdot \frac{{\rm d}F_{\rm s}}{{\rm d}t} \le 8(1+\frac{1}{\epsilon})^2 \cdot \frac{{\rm d}F^*}{{\rm d}t} + (1+\frac{2}{\epsilon}) \cdot \frac{{\rm d}F_{\rm s}}{{\rm d}t}$. □

3.2 Potential Analysis of Lazy Flow $F_{\rm s}$

To prove Lemma 6(ii), we need another potential function to analyze $F_{\rm s}$. Intuitively, if $m_{\rm o}^*$ is large in some S-interval, then OPT would have incurred more wake-up and static energy, and more lazy flow is incurred. We can relate the lazy flow against those OPT's energy. Yet OPT not necessarily wakes up the processors in the same S-interval in which a lot of lazy flow is incurred. Thus, we need the following potential function $\Phi(t)$ to account for the extra energy spent by OPT that has not yet been charged:

$$\Phi(t) = \omega \cdot \max(m_{\rm o}(t) - m_{\rm a}(t), 0) \ .$$

Let $E_{\rm o}^*$ be the total static energy incurred by processors in OPT, i.e., when they are working or idle. Then, $E_{\rm o}^* \le E_{\rm w}^* + E_{\rm i}^*$. We also assume that OPT pays ω units of energy when putting a processor to sleep, and the total amount is denoted by S^*. We have the same number of sleep and wake-up events, so $U^* = S^*$. We will compare SATA with such OPT, and show the following lemma.

LEMMA 8. *During each S-interval $I = [t_1, t_2]$, let $\Delta F_{\rm s} = F_{\rm s}(t_2) - F_{\rm s}(t_1)$, and let $\Delta\Phi$, ΔU^*, ΔS^* and $\Delta E_{\rm o}^*$ similarly for Φ, U^*, S^* and $E_{\rm o}^*$. Then, $\Delta F_{\rm s} + \Delta\Phi \le \Delta U^* + \Delta S^* + 2 \cdot \Delta E_{\rm o}^*$.*

Note that $m_{\rm a}$, $m_{\rm o}$ and hence Φ start and end at 0. With Lemma 8, we can use induction over the S-intervals to show that $F_{\rm s} \le U^* + S^* + 2E_{\rm o}^*$, which implies Lemma 6(ii) that $F_{\rm s} \le 2(E_{\rm w}^* + E_{\rm i}^* + U^*)$.

The rest of this section is devoted to proving Lemma 8. Recall that for any time $t \in I$, $m_{\rm o}^*(t)$ is the maximum number of awake processors in OPT during I. Since $m_{\rm o}^*(t)$ is fixed within I, we drop the parameter t. Similarly, we define $m_{\rm a}^*$ to be the maximum number of awake processor in SATA during I. By definition of S-intervals, I contains a sequence of wake-up events in SATA, followed by a sequence of sleep events in SATA. We split $I = [t_1, t_2]$ into two intervals: the *wake-up interval* I_w starts at t_1 and ends at the last wake-up of SATA in I; and the *sleep interval* I_s starts at the last wake-up of SATA in I and ends at t_2. Thus, $m_{\rm a}$ increases from $m_{\rm a}(t_1)$ to $m_{\rm a}^*$ within I_w, and then decreases from $m_{\rm a}^*$ to $m_{\rm a}(t_2)$ within I_s. We denote the lazy flow $F_{\rm s}$ incurred within I_w and I_s by $\Delta F_{\rm s}(I_w)$ and $\Delta F_{\rm s}(I_s)$, respectively. Note that $\Delta F_{\rm s} = \Delta F_{\rm s}(I_w) + \Delta F_{\rm s}(I_s)$.

Consider the schedule of OPT. Let $m_{\rm o}^-$ be the *minimum* number of awake processors in OPT during I. Within I, $m_{\rm o}$ starts from $m_{\rm o}(t_1)$, reaches the maximum $m_{\rm o}^*$ and the minimum $m_{\rm o}^-$ at different times, and finally ends up at $m_{\rm o}(t_2)$. Therefore,

$$\Delta U^* \ge (m_{\rm o}^* - m_{\rm o}(t_1)) \cdot \omega + (m_{\rm o}(t_2) - m_{\rm o}^-) \cdot \omega \quad \text{and}$$

$$\Delta S^* \ge (m_{\rm o}^* - m_{\rm o}^-) \cdot \omega \ .$$

To prove Lemma 8, it suffices to show that $\Delta F_{\mathrm{s}}(I_w)+\Delta F_{\mathrm{s}}(I_s)+\Phi(t_2) \le (m^*_{\mathrm{o}}-m_{\mathrm{o}}(t_1))\cdot\omega+(m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+(m^*_{\mathrm{o}}-m^-_{\mathrm{o}})\cdot\omega+2\cdot\Delta E^*_{\mathrm{o}}+\Phi(t_1)$.

We relate $\Delta F_{\mathrm{s}}(I_w)$ to SATA's wake-up energy incurred within I_w. By considering the wake-ups of SATA and OPT, the latter can be bounded by OPT's wake-up energy and all OPT's energy stored in Φ at t_1 (Lemma 9). We also relate $\Delta F_{\mathrm{s}}(I_s)$ to SATA's idling energy incurred within I_s. By considering the sleep events of SATA and OPT, the latter can be bounded by OPT's sleeping energy and static energy (Lemma 10). Finally, we show that OPT has enough wake-up energy and static energy remaining to store back to Φ (i.e., at least $\Phi(t_2)$) for the next S-interval (Lemma 11).

LEMMA 9. $\Delta F_{\mathrm{s}}(I_w) \le (\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1))\cdot\omega+\Phi(t_1)$.

PROOF. Recall that lazy flow is the flow incurred when $m_{\mathrm{a}}(t) < m^*_{\mathrm{o}}$. First, suppose $m_{\mathrm{a}}(t_1) < m^*_{\mathrm{o}}$. By definition of SATA, at time t_1 and at the end of I_w, both the counters C_e and C_f are zero. When C_f has accumulated ω units of flow, SATA wakes up a processor and resets C_f to zero. Therefore, $\Delta F_{\mathrm{s}}(I_w)$ equals to the wake-up energy incurred by SATA within I_w, i.e., $(\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{a}}(t_1))\cdot\omega$. The latter is equal to $(\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1))\cdot\omega+(m_{\mathrm{o}}(t_1)-m_{\mathrm{a}}(t_1))\cdot\omega \le (\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1))\cdot\omega+\max(m_{\mathrm{o}}(t_1)-m_{\mathrm{a}}(t_1),0)\cdot\omega$. Therefore, $\Delta F_{\mathrm{s}}(I_w) \le (\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1))\cdot\omega+\Phi(t_1)$.

Now, suppose $m_{\mathrm{a}}(t_1) \ge m^*_{\mathrm{o}}$. Since m_{a} increases from $m_{\mathrm{a}}(t_1)$ to m^*_{a} within I_w, which are always at least m^*_{o}, we have $\Delta F_{\mathrm{s}}(I_w)=0$. We consider two cases depending on whether $m^*_{\mathrm{a}} \ge m_{\mathrm{o}}(t_1)$.

Case 1: $m^*_{\mathrm{a}} \ge m_{\mathrm{o}}(t_1)$. Since $m^*_{\mathrm{o}} \ge m_{\mathrm{o}}(t_1)$ and Φ is always at least 0, we have

$$\Delta F_{\mathrm{s}}(I_w)=0 \le (\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1))\cdot\omega+\Phi(t_1)\ .$$

Case 2: $m^*_{\mathrm{a}} < m_{\mathrm{o}}(t_1)$. In this case, $m_{\mathrm{a}}(t_1) \le m^*_{\mathrm{a}} < m_{\mathrm{o}}(t_1) \le m^*_{\mathrm{o}}$. Thus,

$$\begin{aligned}\Delta F_{\mathrm{s}}(I_w)=0 &\le (\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{a}}(t_1))\cdot\omega\\ &=(\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1)+m_{\mathrm{o}}(t_1)-m_{\mathrm{a}}(t_1))\cdot\omega\\ &=(\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-m_{\mathrm{o}}(t_1))\cdot\omega+\Phi(t_1)\ .\end{aligned}$$ □

LEMMA 10. $\Delta F_{\mathrm{s}}(I_s) \le (m^*_{\mathrm{o}}-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}$.

PROOF. If $m_{\mathrm{a}}(t_2) \ge m^*_{\mathrm{o}}$, since m_{a} decreases from m^*_{a} to $m_{\mathrm{a}}(t_2)$ within I_s, we have $\Delta F_{\mathrm{s}}(I_s)=0 \le (m^*_{\mathrm{o}}-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}$. Now, suppose $m_{\mathrm{a}}(t_2) < m^*_{\mathrm{o}}$. By definition of SATA, at the beginning of I_s and at time t_2, both the counters C_e and C_f are zero. Within I_s, whenever SATA puts a processor to sleep, C_e has accumulated ω units of idling energy; C_f has accumulated less than ω units of flow; and both C_e and C_f are reset to zero. Note also that if C_e is accumulating idling energy when $m_{\mathrm{a}}(t) \le m^-_{\mathrm{o}}$, OPT would have accumulated at least the same amount of static energy. Therefore, we conclude that $\Delta F_{\mathrm{s}}(I_s) < (\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}})-\max(m_{\mathrm{a}}(t_2),m^-_{\mathrm{o}}))\cdot\omega+\Delta E^*_{\mathrm{o}} \le (m^*_{\mathrm{o}}-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}$. □

LEMMA 11. $\Phi(t_2) \le (m^*_{\mathrm{o}}-\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}}))\cdot\omega+(m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}$.

PROOF. If $m_{\mathrm{o}}(t_2) \le m_{\mathrm{a}}(t_2)$, it is trivial that $\Phi(t_2)=\max(m_{\mathrm{o}}(t_2)-m_{\mathrm{a}}(t_2),0)\cdot\omega=0 \le (m^*_{\mathrm{o}}-\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}}))\cdot\omega+(m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}$.

Now, consider the case that $m_{\mathrm{o}}(t_2) > m_{\mathrm{a}}(t_2)$ and we need to bound $\Phi(t_2)=(m_{\mathrm{o}}(t_2)-m_{\mathrm{a}}(t_2))\cdot\omega$. Note that within I_s, whenever SATA puts a processor to sleep, C_e has accumulated ω units of idling energy; C_f has accumulated less than ω units of flow; and both C_e and C_f are reset to zero. Note also that if C_e is accumulating idling energy when $m_{\mathrm{a}}(t) \le m^-_{\mathrm{o}}$, OPT would have accumulated at least the same amount of static energy. Therefore, if $m^*_{\mathrm{a}} \ge m_{\mathrm{o}}(t_2)$,

$$\begin{aligned}&(m_{\mathrm{o}}(t_2)-m_{\mathrm{a}}(t_2))\cdot\omega\\ &\le (m_{\mathrm{o}}(t_2)-\max(m_{\mathrm{a}}(t_2),m^-_{\mathrm{o}}))\cdot\omega+\Delta E^*_{\mathrm{o}}\\ &\le (m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}\\ &\le (m^*_{\mathrm{o}}-\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}}))\cdot\omega+(m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}\ .\end{aligned}$$

Similarly, if $m^*_{\mathrm{a}} < m_{\mathrm{o}}(t_2)$, we have $(m^*_{\mathrm{a}}-m_{\mathrm{a}}(t_2))\cdot\omega \le (m^*_{\mathrm{a}}-\max(m_{\mathrm{a}}(t_2),m^-_{\mathrm{o}}))\cdot\omega+\Delta E^*_{\mathrm{o}}$. Thus, $(m_{\mathrm{o}}(t_2)-m_{\mathrm{a}}(t_2))\cdot\omega=(m_{\mathrm{o}}(t_2)-m^*_{\mathrm{a}}+m^*_{\mathrm{a}}-m_{\mathrm{a}}(t_2))\cdot\omega \le (m^*_{\mathrm{o}}-m^*_{\mathrm{a}})\cdot\omega+(m^*_{\mathrm{a}}-\max(m_{\mathrm{a}}(t_2),m^-_{\mathrm{o}}))\cdot\omega+\Delta E^*_{\mathrm{o}}$, which, by $m^*_{\mathrm{a}} < m_{\mathrm{o}}(t_2)$, is less than $(m^*_{\mathrm{o}}-\min(m^*_{\mathrm{a}},m^*_{\mathrm{o}}))\cdot\omega+(m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+\Delta E^*_{\mathrm{o}}$. □

By Lemmas 9, 10 and 11, we obtain the desired inequality $\Delta F_{\mathrm{s}}(I_w)+\Delta F_{\mathrm{s}}(I_s)+\Phi(t_2) \le (m^*_{\mathrm{o}}-m_{\mathrm{o}}(t_1))\cdot\omega+(m_{\mathrm{o}}(t_2)-m^-_{\mathrm{o}})\cdot\omega+(m^*_{\mathrm{o}}-m^-_{\mathrm{o}})\cdot\omega+2\cdot\Delta E^*_{\mathrm{o}}+\Phi(t_1)$, which implies $\Delta F_{\mathrm{s}}+\Delta\Phi \le \Delta U^*+\Delta S^*+2\cdot\Delta E^*_{\mathrm{o}}$. Thus Lemma 8 is proven.

4. SATA WITH SPEED SCALING

This section considers scheduling in the speed scaling model. In this model, at any time t, each processor $i \in [1,m]$ can independently scale its speed $s_i(t) \in [0,T]$, where T is the maximum allowable speed, and the processor consumes power at rate $P(s_i(t))$. Without loss of generality, we can assume $P(0)=0$, and P is defined, strictly increasing, strictly convex, continuous and differentiable at all speeds in $[0,T]$; if $T=\infty$, the speed range is $[0,\infty)$ and for any speed x, there exists x' such that $P(x)/x < P(s)/s$ for all $s > x'$ (otherwise the optimal speed scaling policy is to always run at the infinite speed and an optimal schedule is not well-defined). We use $Q(x)$ to denote $\min\{P^{-1}(x),T\}$. Note that Q is monotonically increasing and concave. E.g., if $P(s)=s^\alpha$ for some $\alpha > 1$, then $Q(x)=\min\{x^{1/\alpha},T\}$.

This section shows how to extend SATA to the speed scaling model. The objective is to minimize the total flow time F plus energy E. Note that a $(1+\epsilon)$-speed processor for any $\epsilon > 0$ can run at speed $(1+\epsilon)s$ when given power $P(s)$. The following theorem shows that the extended SATA is $(1+\epsilon)^2$-speed $O(\frac{1}{\epsilon^3})$-competitive.

THEOREM 12. *For any $\epsilon > 0$,* SATA *is $(1+\epsilon)^2$-speed $80(1+\frac{1}{\epsilon})^3$-competitive for flow plus energy, and* SATA *migrates a job at most 4 times on average.*

SATA handles job dispatching, migrations and selection in the same way as in Section 2. Let n_{a} be the current number of active jobs. Whenever $n_{\mathrm{a}} \ge m$, SATA maintains the arrival-time alignment property with alignment width m. This ensures that each processor is processing exactly k jobs, where k is the smallest integer such that $km \ge \lceil\beta n_{\mathrm{a}}\rceil$ where $\beta=\frac{\epsilon}{4(1+\epsilon)}$. After each job arrival and completion, SATA migrates up to 2 jobs to maintain the arrival-time-alignment property. SATA handles speed scaling as follows: At any time, each of the processors with active jobs works at the same speed. The total power of the processors is set to the number of active jobs n_{a}, except that when the processor speed exceeds the maximum speed T, it is capped at T. Below we give the details of extending SATA.

Job arrival: When a job j arrives, if $n_a < m$, dispatch j to a processor without jobs; otherwise, dispatch j and migrate at most 2 jobs according to the job arrival procedure in Section 2.

Job completion: When a job is completed, if $n_a \leq m$, no migration is needed; otherwise, migrate at most 2 jobs according to the job completion procedure in Section 2.

Job scheduling:

- If $n_a < m$, each active job is processed on a different processor running at speed $(1+\epsilon)Q(1)$;
- If $n_a \geq m$, each processor runs at speed $(1+\epsilon)Q(\frac{n_a}{m})$ and shares its processing time equally among all its tail jobs and its boundary job (if present). Note that the m processors together are processing km latest-arrival jobs, where k is the smallest integer with $km \geq \lceil \beta n_a \rceil$.

Restricting offline algorithm. It remains to show the competitive ratio of SATA stated in Theorem 12. We need to compare SATA against the optimal offline algorithm. To ease the analysis, we assume that OPT is the algorithm GKP [12], which is $(1+\epsilon)$-speed $4(1+\frac{1}{\epsilon})$-competitive for flow plus energy. GKP is clairvoyant and non-migratory in nature, and it always dispatches a job to a processor at release time. At any time t, GKP's processor $i \in [1,m]$ runs at speed $s_i^*(t) = Q(n_i^*(t))$, where $n_i^*(t)$ is the current number of active jobs assigned to processor i. This implies that if GKP has $n_o(t)$ active jobs at time t, i.e., $n_o(t) = \sum_{i=1}^m n_i^*(t)$, then the total speed over all processors of GKP is at most $m \cdot Q(\frac{n_o(t)}{m})$ if $n_o(t) \geq m$, and $n_o(t) \cdot Q(1)$ otherwise.[4] To show Theorem 12, it suffices to show that SATA is $(1+\epsilon)$-speed $20(1+\frac{1}{\epsilon})^2$-competitive against this restricted OPT.

Potential analysis of total flow F. The SATA's energy usage E is at most its total flow F because, by the definition of SATA, at any time, if $n_a < m$, the total power of all processors is $n_a \cdot P(Q(1)) \leq n_a \cdot P(P^{-1}(1)) = n_a$; and if $n_a \geq m$, the total power of all processors is $m \cdot P(Q(\frac{n_a}{m})) \leq m \cdot P(P^{-1}(\frac{n_a}{m})) = n_a$. Thus, it suffices to analyze SATA's flow time F. To this end, we adapt the potential function in Section 2 by taking speed scaling into account. Recall that SATA's active jobs are arranged in ascending order of arrival times. For the i-th job j_i, we denote x_i as the amount of work of j_i in SATA that is lagging behind OPT (see Section 2 for the definitions). Recall that $n_a(t)$ and $n_o(t)$ are the number of active jobs in SATA and OPT, respectively. The new potential function is defined as follows:

$$\Phi(t) = \frac{2}{\epsilon} \cdot \sum_{i=1}^{n_a(t)} \frac{\max\left(1, \frac{i}{m}\right)}{Q(\max\left(1, \frac{i}{m}\right))} \cdot x_i \ ,$$

where $\max\left(1, \frac{i}{m}\right) / Q(\max\left(1, \frac{i}{m}\right))$ is called the *coefficient* of job j_i. In the full paper , we will prove the running condition $\frac{dF(t)}{dt} + \frac{d\Phi(t)}{dt} \leq 10(1+\frac{1}{\epsilon})^2 \cdot \frac{dF^*(t)}{dt}$. Then Theorem 12 follows.

[4] By the concavity of Q, if $n_o(t) \geq m$, $\sum_{i=1}^m s_i^*(t) = \sum_{i=1}^m Q(n_i^*(t)) \leq mQ(\sum_{i=1}^m n_i^*(t)/m) = mQ(n_o(t)/m)$. Otherwise, GKP can use at most $n_o(t)$ processors, and similarly, $\sum_{i=1}^m s_i^*(t) \leq n_o(t)Q(n_o(t)/n_o(t)) = n_o(t)Q(1)$.

5. REFERENCES

[1] S. Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

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

[3] N. Avrahami, and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. *Algorithmica*, 47(3):253–268, 2007.

[4] N. Bansal, H. L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. In *Proc. SODA*, pages 693–701, 2009.

[5] S. H. Chan, T. W. Lam, and L. K. Lee. Non-clairvoyant speed scaling for weighted flow time. In *Proc. ESA*, pages 23–35, 2010.

[6] S. H. Chan, T. W. Lam, L. K. Lee, C. M. Liu, and H. F. Ting. Sleep management on multiple processors for energy and flow time. In *Proc. ICALP*, pages 219–231, 2011.

[7] C. Chekuri, A. Goel, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize flow time with ϵ resource augmentation. In *Proc. STOC*, pages 363–372, 2004.

[8] J. Chadha, N. Garg, A. Kumar, and V. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelated processors with speed augmentation. In *Proc. STOC*, pages 679–684, 2009.

[9] J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *Proc. SODA*, pages 685–692, 2009.

[10] K. Fox and B. Moseley. Online scheduling on identical processors using SRPT. In *Proc. SODA*, pages 120–128, 2011.

[11] A. Gandhi, V. Gupta, M. Harchol-Balter, and M. Kozuch. Optimality analysis of energy-performance trade-off for server farm management. *Performance Evaluation*, 67(11):1155–1171, 2010.

[12] A. Gupta, R. Krishnaswamy, and K. Pruhs. Scalably scheduling power-heterogeneous processors. In *Proc. ICALP*, pages 312–323, 2010.

[13] A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *Proc. SODA*, pages 1242–1253, 2012.

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

[15] S. Khuller, J. Li, B. Saha. Energy efficient scheduling via partial shutdown. In *Proc. SODA*, pages 1360–1372, 2010.

[16] T. W. Lam, L. K. Lee, H. F. Ting, I. To, and P. Wong. Sleep with guilt and work faster to minimize flow plus energy. In *Proc. ICALP*, pages 665–676, 2009.

[17] S. Leonardi, and D. Raz. Approximating total flow time on parallel processors. *Journal of Computer and System Sciences*, 73(6):875–891, 2007.

[18] J. Li and S. Khuller. Generalized processor activation problems. In *Proc. SODA*, pages 80–94, 2011.

[19] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theor. Comput. Sci.*, 130(1):17–47, 1994.

[20] E. Torng and J. McCullough. SRPT optimally utilizes faster processors to minimize flow time. *ACM Transactions on Algorithms*, 5(1):1, 2008.

Appendix A: Lower Bound

In this appendix, we assume processors are always awake and consider nonclairvoyant scheduling that minimizes the total flow time. We give a lower bound result on deterministic nonclairvoyant algorithms that allows a job to be migrated $c < 1$ times on average. Below we do not assume that each job is dispatched at its arrival time.

THEOREM 13. *If at most $c < 1$ migration is allowed per job on average (or equivalently, the total number of migrations is at most cn where n is the number of jobs), the competitive ratio of any deterministic nonclairvoyant algorithm, when using s-speed processors for $s \geq 1$, is at least $\frac{1}{16}\min(\frac{m}{s}, \frac{1}{\sqrt{cs}})$ for minimizing total flow time.*

PROOF. Consider any nonclairvoyant online algorithm ONLINE that is using m s-speed processors, where $s \geq 1$. Let ADV be the offline adversary that is using m 1-speed processors. We will show that the competitive ratio of ONLINE is at least $\frac{m}{8sx}$, where $x = \lceil\sqrt{c}m\rceil$ if $c > 0$, and $x = 1$ if $c = 0$. Then Theorem 13 follows (when $c > 0$, $x = \lceil\sqrt{c}m\rceil \leq \max(2\sqrt{c}m, 1)$, and $\frac{m}{8sx} \geq \frac{m}{8s\max(2\sqrt{c}m,1)} \geq \frac{1}{16}\min(\frac{m}{s}, \frac{1}{\sqrt{cs}})$; when $c = 0$, $x = 1$ and $\frac{m}{8sx} \geq \frac{m}{16s} = \frac{1}{16}\min(\frac{m}{s}, \frac{1}{\sqrt{cs}})$).

At time 0, ADV releases $n = m^3$ jobs $\{j_1, j_2, \ldots, j_{m^3}\}$. Define $\epsilon = \frac{1}{n^2}$. ADV will determine the sizes of all jobs at time 1. Consider ONLINE's schedule at time 1. Suppose ONLINE has processed $q(j_i)$ units of work for each job j_i.

- Let U be the set of jobs j_i that have $q(j_i) = 0$, i.e., ONLINE has not yet processed or dispatched them by time 1.
- Let P be a set of x processors to which ONLINE has dispatched most jobs.

We divide the proof into two cases based on a threshold $2mx^2$, which can be smaller, equal to, or bigger than n depending on the value of c.

Case 1. The total number of jobs in the x processors of P is less than $2mx^2$. In this case, ADV sets each job j_i to have size $q(j_i) + \epsilon$. Since ONLINE does not complete any job before time 1, ONLINE has total flow at least $1 \cdot n = n = m^3$. Consider ADV. ADV can first process ϵ units of work for each job on a single processor, and all jobs in U will be completed by time $n \cdot \epsilon = \frac{1}{n}$. ADV then simulates ONLINE's schedule on the other jobs from time 0 to time 1 (but working s times slower). Thus ADV has total flow at most $\frac{1}{n} \cdot n + s \cdot (n - |U|)$. Since the x processors of P are hosting most jobs dispatched by ONLINE, the total number of jobs that have been dispatched by ONLINE is at most $\min(2mx^2 \cdot \frac{m}{x}, n) = \min(2m^2x, n) \leq 2m^2x$ and thus $n - |U| \leq 2m^2x$. ADV has total flow at most $\frac{1}{n} \cdot n + s \cdot 2m^2x \leq 8m^2sx$. The competitive ratio of ONLINE is at least $\frac{m^3}{8m^2sx} = \frac{m}{8sx}$.

Case 2. The total number of jobs in the x processors of P is at least $2mx^2$. In this case, ADV selects $2mx^2$ jobs that are currently on P and set their size to sn. Note that $s < sn$. At time 1, ONLINE has processed any of these jobs for at most s units of work, and none of them is completed. Since ONLINE can only migrate at most cn jobs in total, at least $2mx^2 - cn$ jobs of size sn must be entirely run on the x processors in P. To minimize the flow, a best way for ONLINE is to process these jobs evenly on the x processors. Note that $x = 1$ if $c = 0$, and $x = \lceil\sqrt{c}m\rceil$ if $c > 0$. Then, we have $cn \leq mx^2$ and $2mx^2 - cn \geq mx^2$. Therefore, the total flow incurred by ONLINE is at least $x(1 + 2 + \cdots + \frac{mx^2}{x}) \cdot \frac{sn}{s} \geq \frac{1}{2}m^2x^3n$. Consider ADV. ADV can first simulate ONLINE's schedule on all jobs except those with size sn, so these jobs would have been completed by time $s(1 + n \cdot \epsilon) = s(1 + \frac{1}{n})$. Then, ADV processes the $2mx^2$ jobs with size sn evenly on m processors. The total flow of ADV is at most $s(1 + \frac{1}{n}) \cdot n + m(1 + 2 + \cdots + \frac{2mx^2}{m}) \cdot sn \leq 2sn + m \cdot 3x^4sn \leq 4mx^4sn$ (since $m \geq 2$ and $x \geq 1$). Therefore, the competitive ratio of ONLINE is at least $\frac{\frac{1}{2}m^2x^3n}{4mx^4ns} = \frac{m}{8sx}$. □

Appendix B: Migration Upper Bound (Section 3)

LEMMA 14. *Consider any sequence of wake-ups together and all the migrations involved in maintaining the arrival-time-alignment property. On average (over all jobs), a job is migrated at most $(\ln m + 2)$ times.*

PROOF. We consider each maximal sequence of consecutive wake-ups in SATA. Let t_1 be the time right after the last sleep event before this wake-up sequence; if such sleep event does not exist, let $t_1 = 0$. Let t_2 be the time right after the last wake-ups in the wake-up sequence. We will show that within the interval $I = [t_1, t_2]$, on average (over all jobs arriving in I), each job migrates at most $\ln m + 2$ times. Considering all such intervals I together, the average number of migrations per job is still at most $\ln m + 2$, and the lemma follows.

Consider the interval $I = [t_1, t_2]$. Suppose there are m_0 awake processors at t_1 and ℓ wake-ups within I. When SATA makes the i-th wakeup, let n_i be the current number of active jobs. Note that on the i-th wake-up, the number of awake processors becomes $m_i = m_0 + i$. If $n_i < m_i$, each job is hosted by a different processor, and no job needs to be migrated. Otherwise, we need to maintain the arrival-time-alignment property. We migrate job $j_{\tau-m-1}$ (if exist) to the newly awaken processor as a new aligned job, so that property C1 holds. To maintain C2 and C3, we need to migrate $\lceil\lceil\beta n_i\rceil/m_i\rceil$ tail jobs from existing awake processors to the newly awaken processor. Let $k_i \geq 1$ be the smallest integer such that $k_im_i \geq \lceil\beta n_i\rceil$. In other words, we need to migrate k_i tail jobs to the newly awaken processor. Since $n_i \geq m_i$ and $\beta < 0.5$, we have $k_im_i \leq n_i$. Thus, the total number of migrations within I is at most

$$N = \sum_{1\leq i\leq \ell: n_i \geq m_i} (1 + k_i) \leq \sum_{1\leq i\leq \ell: n_i \geq m_i} (1 + \frac{n_i}{m_i}) .$$

We can lower bound the number of jobs arriving in I, as follows. At time t_1, the number of active jobs must be no more than m_0 since there is a sleep event right before t_1 or $t_1 = 0$. Define $n^* = \max\{n_1, \ldots, n_l\}$. Then, the number of jobs arriving within I is at least $n^* - m_0$.

Now, we relate N with $(n^* - m_0)$. Let $r \leq \ell$ be the largest integer such that $n_r \geq m_r$. If such r does not exist, there is no migration within I. Otherwise, $N \leq \sum_{i=1}^{r}(1 + n_i/m_i)$ and $n_r \geq m_r = m_0 + r$. Note that $r \leq n_r - m_0 \leq n^* - m_0$, and $r \leq m$ (the wake-up sequence in I contains consecutive wake-ups). Also note that $n_i/m_i \leq n^*/m_i \leq (n^* - m_0)/(m_i - m_0) = (n^* - m_0)/i$. Therefore,

$$N \leq \sum_{i=1}^{r}(1 + \frac{n_i}{m_i}) \leq (n^* - m_0) \cdot \left(1 + \sum_{i=1}^{r}\frac{1}{i}\right)$$
$$\leq (n^* - m_0) \cdot (2 + \ln r) \leq (n^* - m_0) \cdot (2 + \ln m) .$$

Therefore, the average number of migrations per job within I is at most $N/(n^* - m_0) \leq \ln m + 2$. □