Sleep Management on Multiple Machines for Energy and Flow Time

Sze-Hang Chan[1], Tak-Wah Lam[1], Lap-Kei Lee[2], Chi-Man Liu[1], and Hing-Fung Ting[1]

[1] Department of Computer Science, University of Hong Kong, Hong Kong
[2] MADALGO Center for Massive Data Algorithmics, Aarhus University, Denmark

Abstract. In large data centers, determining the right number of operating machines is often non-trivial, especially when the workload is unpredictable. Using too many machines would waste energy, while using too few would affect the performance. This paper extends the traditional study of online flow-time scheduling on multiple machines to take sleep management and energy into consideration. Specifically, we study online algorithms that can determine dynamically when and which subset of machines should wake up (or sleep), and how jobs are dispatched and scheduled. We consider schedules whose objective is to minimize the sum of flow time and energy, and obtain $O(1)$-competitive algorithms for two settings: one assumes machines running at a fixed speed, and the other allows dynamic speed scaling to further optimize energy usage.

Like the previous work on the tradeoff between flow time and energy, the analysis of our algorithms is based on potential functions. What is new here is that the online and offline algorithms would use different subsets of machines at different times, and we need a more general potential analysis that can consider different match-up of machines.

1 Introduction

Energy consumption is a major concern for large-scale data centers. It has been reported that the energy consumption of the data centers in US costs more than $4.5 billion a year and accounts for more than 1.5% of the total electricity usage in US [15]. When a machine (or server) is on, the power consumption is divided into *dynamic power* and *static power*; the former is consumed only when the machine is processing a job, while the latter is consumed constantly (due to leakage current) even when the machine is idle (e.g., an Intel Xeon E5320 server requires 150W when idling and 240W when working [7]). The static power consumption is cut off only when a machine is sleeping. From the energy viewpoint, a data center should let machines sleep whenever they are idle, yet waking up a machine later requires extra energy. It is energy inefficient to frequently switch a machine on and off. It is challenging to determine dynamically the appropriate number of working machines so as to strike a balance between energy usage and quality of service (QoS), especially when the workload is unpredictable. This paper initiates a theoretical study of online job scheduling on a pool of identical

L. Aceto, M. Henzinger, and J. Sgall (Eds.): ICALP 2011, Part I, LNCS 6755, pp. 219–231, 2011.

machines that takes sleep management, energy and QoS into consideration. We consider both models where machines are running at a fixed speed and machines can each scale their speed to control their power, respectively.

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 time elapsed since the job is released until it is completed. Without energy concern, there is already considerable amount of research on minimizing total flow time alone (see the survey [14]). It is well-known that the online algorithm SRPT (shortest remaining processing time) minimizes total flow time on a single machine. For scheduling on $m > 1$ identical machines, no online algorithm can be $O(1)$-competitive even if job migration is allowed; nevertheless, Chekuri et al. [6] showed that the non-migratory algorithm IMD [4] is $O(1 + \frac{1}{\epsilon})$-competitive for any $\epsilon > 0$ when using $(1 + \epsilon)$ times faster machines.

Sleep management and energy. The above results assume that all machines are always on. This paper extends the study of multi-machine scheduling to consider sleep management and the tradeoff between flow time and energy. When a job arrives, we need to determine whether to wake up a sleeping machine to process it or assign it to an awake machine. We also have to decide when to put machines to sleep to save static power. Waking up machines too aggressively wastes a lot of energy, while an over-conservative wake-up policy accumulates excessive flow time. The objective is to minimize the flow plus energy. Lam et al. [11] studied this problem in the single-machine setting and gave an $O(1)$-competitive algorithm for flow time plus energy. In the multi-machine setting, sleep management gets complicated and cannot be considered separately on each machine; for instance, a scheduler may overload some machines to put others to sleep earlier. No online algorithm has been known for the multi-machine setting. A relevant work is by Khuller et al. [10] who considered an offline problem of minimizing makespan subject to a wake-up budget for jobs all released at time 0.

Speed scaling. All the results above assume that machines run at a fixed speed. Another energy saving technology is *dynamic speed scaling*, which allows a processor to scale its speed dynamically. Running a job slower can reduce energy usage, but it leads to longer flow time. There are several online results that take speed scaling into consideration and attempts to minimize the total flow plus energy (see the survey [1]). Most results are based on a model in which the processor, when running at speed $s \in [0, \infty)$, consumes dynamic power s^α, where $\alpha > 1$ and is typically 3 [16]. Here a scheduler needs a speed scaling policy to determine the speed of each processor. When there is only one processor which is assumed to be always on, the best algorithm uses the job selection algorithm SRPT and the speed scaling algorithm AJC, which sets its speed based on the number of active jobs [13]. This algorithm is 2-competitive for flow plus energy [5,3]. Lam et al. [11] have also studied single-processor scheduling that exploits both sleep management and speed scaling, and gave an $O(\frac{\alpha}{\log \alpha})$-competitive algorithm for flow plus energy.

Existing online results on speed scaling for multiple machines do not consider sleep management [12,9,8]. The best result in the non-migratory model is by Gupta et al. [8]. They assume each machine runs the best single-machine speed scaling algorithm as mentioned above, and consider a job dispatching algorithm which assigns each new job to the machine that would result in the least increase in the future flow plus energy. The algorithm is $O(\alpha)$-competitive for flow plus energy. It remains open whether there exists a competitive algorithm when machines can exploit both speed scaling and sleep management.

Note that speed scaling and sleep management might work in opposite directions; the former prefers load balancing and working slowly, while the latter could overload some machines (and make them run faster) so as to let other machines sleep. Furthermore, speed scaling can be optimized separately within individual machine, but sleep management often requires considering all machines together.

Our contribution. This paper gives the first sleep management algorithm on multiple machines for minimizing total flow plus energy. Our results cover both the fixed speed and speed scaling models. We only consider non-migratory schedules since migration requires overheads in practice. In the fixed speed model, it is easy to show that any online algorithm is $\Omega(m)$-competitive even if all jobs have unit size. In view of the lower bound, we give the online algorithm $(1+\epsilon)$-speedup processors, which run $(1+\epsilon)$ times faster but does not consume more power. We show an online algorithm POOL that is $O(1+\frac{1}{\epsilon})$-competitive for total flow plus energy when using $(1+\epsilon)$-speedup processors. (Our result remains valid even if a machine when running at speed $(1+\epsilon)$ is charged for more power, say, $(1+\lambda)$ times of the original power. The competitive ratio would increase by a factor of $(1+\lambda)$.) We can adapt POOL to the speed scaling model using AJC; it is $O(\alpha)$-competitive for flow plus energy. Our new algorithm has comparable performance with the best speed scaling results on multiple machines that assume all machines are always on [12,9,8].

The core idea of our algorithm is to keep a pool of "dispatchable" machines, which are either all asleep or all awake. A new job is dispatched only to a machine currently in the pool. We separate the management of the pool from the job dispatching policy; the former depends on the history of the workload while the latter depends on the size of the current job. We exploit three simple ideas to manage the pool: (1) uses the total flow to trigger all machines in the pool to wake up; (2) uses the total working flow plus energy to decide when to include more machines; and (3) uses the total idling energy to decide when to remove a machine from the pool. Like many online algorithms, POOL is conservative in committing resources (i.e., waking up machines). Yet POOL can keep its flow under control even if there are many occasions when it uses too few machines.

We also face new challenge when analyzing the competitiveness. As in previous work, we need to discover the "right" potential function to account for the difference in the progress of the online algorithm and the optimal offline algoithm OPT. With different sleep management, POOL and OPT may indeed operate with a different subset of machines, and it is possible that POOL makes machine i heavily loaded while OPT puts machine i to sleep. It makes no sense

to compare their progress of machine i in POOL and OPT. We should match the machines of POOL and OPT with the same state and measure the progress of each pair. This paper gives a new potential analysis that allows us to dynamically change the matching of the machines so as to minimize the number of state mismatch. One might think that changing the matching might require us to "restart" the potential analysis. Yet we observe that such change cannot increase the potential. Another important observation is that we can restrict our attention to those offline algorithms that at any time have at most one machine that is sleeping and has unfinished jobs.

Definitions and notations. The input is a sequence of jobs arriving online, to be scheduled on $m \geq 1$ machines. The job's size is arbitrary and only known at release time. Jobs are sequential in nature (i.e., each job can be processed by one machine at a time). We consider only non-migratory schedules, in which each job is dispatched to one machine and is run entirely on that machine. Preemption is allowed, and a preempted job can resume at the point of preemption.

Sleep, awake, and static power. At any time, a machine is in either the *awake* state or the *sleep* state. It can process a job only when it is awake, and the energy is consumed at the rate $\mu > 0$, which includes both static and dynamic power. An awake machine can be idle (i.e., speed $= 0$) and only requires the static power σ ($0 < \sigma < \mu$). It can enter the sleep state to further reduce the power to zero. Initially, all machines 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 to the awake state requires an amount ω of energy, and the reverse takes zero energy. It is useful to differentiate two types of awake state, namely, with zero speed and with positive speed, which are referred to respectively as the *awake-idle* and *awake-working* state, or simply the *idle* and *working* state. Furthermore, we call a sleeping machine in the *procrastinating* state if the machine has jobs not yet finished.

Fixed-speed and speed scaling models. In the fixed-speed model, a working machine runs at speed 1 (i.e., processing x units of work in x units of time) and the power required is μ. In the speed scaling model, a working machine can run at any speed $s > 0$ at any time but at different power. We assume that the *power function* or the rate of energy usage is $P(s) = s^\alpha + \sigma$, where $\alpha > 1$ is a constant.

Flow and energy. Consider a schedule of jobs. At any time t, a job j is said to be *active* if j has been released but not yet completed. Its *flow time* or simply *flow* $F(j)$ is the time elapsed since it arrives and until it is completed. The total flow is $F = \sum_j F(j)$. Let n_t be the number of active jobs at time t. We can also view the total flow as a quantity incurring at a rate equal to the number of active jobs, i.e., $F = \int_0^\infty n_t \, \mathrm{d}t$. Energy is consumed by awake machines continuously over time and by discrete wake-up transitions. Denote the total energy as E. The cost G of a schedule is defined to be $F + E$. And the objective is to find a schedule that minimizes G. Furthermore, suppose that each job is dispatched to some machine immediately at release time. Then we can consider the flow and energy machine by machine. For each machine, we are particularly interested in the flow and energy incurred over the time when it is in the working state.

We define the *working cost* G_{W} of a schedule to be the sum over all machines of the flow and energy incurred when the machine is in the working state. Obviously, $G_{\mathrm{W}} < G$ as the latter includes the flow incurred when a machine is sleeping, the energy when idle, and the wake-up energy.

2 Sleep Management for Fixed-Speed Machines

This section focuses on fixed-speed machines and gives our algorithm POOL that handles sleep management, job dispatching and scheduling in an integrated way. POOL is using $m > 1$ machines that are $(1+\epsilon)$ times faster than the offline optimal algorithm OPT, while using the same power. (Recall that without extra speed, any online algorithm is $\Omega(m)$-competitive.) Following is our main result.

Theorem 1. *For any $\epsilon > 0$,* POOL *is $O(1+\frac{1}{\epsilon})$-competitive for flow plus energy when using $(1+\epsilon)$-speedup machines.*

Remaining working cost (rwc). As to be shown, POOL schedules jobs using SRPT. It is useful to compute the remaining working cost (rwc) required to serve the remaining jobs on any machine i that runs SRPT at a speed $s \geq 1$. For any time t and any $q \geq 0$, let $n_{i,t}(q)$ be the number of active jobs in machine i with remaining work at least q. Then at any time t, the *rwc* of machine i equals $\int_{q=0}^{\infty} \sum_{k=1}^{n_{i,t}(q)} (\frac{k+\mu}{s})\,\mathrm{d}q$. Moreover, if a job j of size $p(j)$ arrives at time t and is dispatched to machine i immediately, the increase in *rwc* due to j equals $\int_{q=0}^{p(j)} (\frac{n_{i,t}(q)+1+\mu}{s})\,\mathrm{d}q$ (note that $n_{i,t}(q)$ refers to the number before j arrives). Obviously, for a sequence of jobs J, the sum of the increases in *rwc* at their release times equals the total working cost of serving J.

2.1 Algorithm POOL

The core idea is to maintain a small pool P of *dispatchable machines*. P contains one sleeping machine initially and is always non-empty. At any time, machines in P are either all asleep or all awake, and P is said to be asleep and awake, respectively. Machines not in P are all asleep and do not have active jobs. POOL would gradually include more machines into P as jobs arrive, and they are put into the same state as P. POOL exploits three simple concepts to manage P: (1) total flow for triggering all machines in P to wake up; (2) total working cost for determining when to include more machines into P; and (3) total idling energy for determining when to put an idle machine to sleep and remove it from P.

To this end, POOL maintains three (real-value) counters B, C and D to keep track of the accumulated flow (when P is asleep), increase in working cost and idling energy, respectively. Initially, all counters equal 0. C only increases when a job arrives. When P is asleep, B increases (continuously) at rate of the number of active jobs. When P is awake, D increases at rate of σ times the number of idle machine, but once D reaches ω, its value is capped there. Intuitively, when D reaches ω, we could remove one idle machine from P. But this turns out to

be too aggressive. Let P_{idle} be the set of idle machines in P. Indeed POOL never sleeps an idle machine if it is the only idle machine in P (i.e., $|P_{\text{idle}}| = 1$) but $|P| \geq 2$; it does so only if there are two or more idle machines or $|P| = 1$.

When a job j arrives, POOL first assumes that j is dispatched to a machine with no active jobs and calculates the increase in *rwc*. Denote this amount of increase as $null\_Inc\_rwc(j)$. If $null\_Inc\_rwc(j)$ can compensate the wake-up energy, POOL will include one machine into P and dispatch j to this machine (even if P already has an idle machine). Otherwise, POOL dispatches j to a machine i in P that minimizes the increase in *rwc*; below we denote machine i as $\ell(j, P)$ and the amount of increase in *rwc* as $min\_Inc\_rwc(j, P)$. Note that $min\_Inc\_rwc(j, P) \geq null\_Inc\_rwc(j)$, and the total *rwc* also increases by the same amount in both cases. C keeps track of the increase in *rwc*; whenever it reaches a multiple of ω, we include one more machine into P.

Job dispatching (& expand P): When a job j arrives,
If $((|P| < m)$ & $(C + null\_Inc\_rwc(j) \geq \omega))$,
add a machine to P; dispatch j to this machine; $C = C + null\_Inc\_rwc(j) - \omega$;
else dispatch j to $\ell(j, P)$; $C = C + min\_Inc\_rwc(j, P)$;
While $((C \geq \omega)$ & $(|P| < m))$ **do** { add a machine to P; $C = C - \omega$. }
If $C > \omega$ **then** $C = \omega$.

Wake up P: If (P is asleep) & $(B = \omega)$, wake up all P's machines; reset $B = 0$.

Sleep a machine (& shrink P): When $D = \omega$,
- if $|P_{\text{idle}}| \geq 2$, remove one idle machine from P and put it to sleep; reset $D = 0$;
- if $|P_{\text{idle}}| = |P| = 1$, put P to sleep; reset $C = D = 0$.

Job scheduling in each machine of P: When awake, use SRPT policy.

Intuitively, scheduling for a small job (say, $null\_Inc\_rwc(j) < \omega$) is not obvious. It is too small to justify waking up a machine, yet dispatching it to an awake machine may preempt other jobs there (due to SRPT policy) and suddenly cause a huge increase of *rwc*. Interestingly, POOL can maintain a useful property that it always has at least one awake machine with a small workload and dispatching j to it cannot increase the *rwc* too much (say, $\leq 2\omega$).

Property 1. Every time after POOL executes the job dispatching procedure, it maintains the invariant that if $|P| < m$, there exists a machine with $rwc < \omega$.

POOL never lets an awake machine idle if that machine has active jobs. Thus, a machine can accumulate flow only when it is working or sleeping. Consider a schedule of POOL. We divide POOL's total flow F into two parts: the *working* flow F_{W} and the *sleeping* flow F_{S}, which refer to the total flow incurred by the machines when they are working and sleeping, respectively. We also divide POOL's energy usage E into three parts: E_{I} is the idling energy (static energy usage during the idle state), E_{W} the working energy, and U the wake-up energy. Note that POOL's working cost $G_{\text{W}} = F_{\text{W}} + E_{\text{W}}$, and its total cost $G = F_{\text{W}} + F_{\text{S}} + E_{\text{W}} + E_{\text{I}} + U$.

The analysis of POOL centers on a rather complicated potential analysis of F_{W}, which gives Lemma 1(i) below. Note that we will bound F_{W} by OPT's total cost G^* together with the sleeping flow F_{S}, because the sleep management

of POOL sometimes delays jobs and increases their flow. Such excess in flow is related to F_{S} but not OPT. To upper bound the other components of G, we show Lemma 1(ii). Lemmas 1(i) and (ii) would imply that POOL is $O(1)$-competitive.

Lemma 1. (i) $F_{\mathrm{W}} \leq (9 + \frac{10}{\epsilon})G^* + \frac{1}{\epsilon}F_{\mathrm{S}}$; (ii) $G \leq 4F_{\mathrm{W}} + 7G^*$.

Proof (Sketch of Lemma 1(ii)). We derive three properties of POOL: (a) $F_{\mathrm{S}} \leq G^*$; (b) $U \leq G_{\mathrm{W}} + G^*$; (c) $E_{\mathrm{I}} \leq U + E_w$. As POOL uses $(1+\epsilon)$-speedup machines, E_{W} is less than OPT's working energy and thus G^*. Thus, Lemma 1(ii) follows.

It is useful to partition the timeline into intervals called P-intervals, each of which consists of a maximal asleep period of P, followed by a maximal awake period of P. For a P-interval I, we can show that the total cost incurred by OPT within I (denoted by $G^*(I)$) is at least ω. For (a), the sleeping flow accumulated by POOL in the asleep period of I is $\omega \leq G^*(I)$. Summing over all I's gives $F_{\mathrm{S}} \leq G^*$. For (b), within I, except for the first machine added to P, a machine is added to P when the accumulated increase of *rwc* (i.e., counter C) reaches a multiple of ω. When I ends, this accumulated increase in *rwc* would be fully reflected in the working cost G^* incurred within I. Thus, within I, the total wake-up energy (including that for the first added machine) is at most the working cost plus ω (at most $G^*(I)$). Summing over all I's gives $U \leq G_{\mathrm{W}} + G^*$. For (c), E_{I} can be incurred when $D < \omega$ and when $D = \omega$. The first type increases at the same rate as D and is at most U. The second type is incurred when $|P_{\mathrm{idle}}| = 1$ but $|P| \geq 2$, i.e., a working machine exists in P, which is thus at most E_{W}. □

2.2 Potential Analysis of F_{W}

One might think that POOL is rather conservative in waking up machines and might sacrifice flow for energy. Indeed POOL can always catch up in time its number of machines and keep its flow under control. In particular, we can upper bound POOL's increase of flow even when it is using fewer machines than OPT. The analysis is complicated because POOL and OPT may use different subsets of (awake) machines. The rest of this section shows Lemma 1(i) using a potential function that allows different match-up between machines of POOL and OPT.

Restricting OPT. In analyzing F_{W}, we restrict OPT to be the optimal offline algorithm that always uses *SRPT* for job selection and has at most one procrastinating machine at any time. In Section 4, we show that such OPT incurs at most three times the total cost of an unrestricted one. Henceforth, we focus on the restricted OPT and show that POOL is $O(1)$-competitive against it.

Let $F_{\mathrm{W}}(t)$ denote the working flow F_{W} incurred up to time t by POOL. Similarly, define $G^*(t)$ for OPT's total cost G^*. Assume that machines are labeled with integers from 1 to n. At any time, we match each machine in POOL with a certain machine in OPT. Below we denote $x(i)$ as the machine in OPT currently matched with machine i in POOL. This matching is only for the purpose of analysis and not known to the algorithms. Initially $x(i) = i$ for all i. To show Lemma 1(i), we define a potential function $\Phi(t)$ that reflects POOL's remaining

working cost discounted in view of OPT's workload in the corresponding machines. Technically, we want $\Phi(t)$ to satisfy the following conditions: **(i)** *Boundary condition:* $\Phi = 0$ before any job is released and after all jobs are completed. **(ii)** *Job completion and state transition condition:* Φ does not increase when a job is completed or a machine changes its state in POOL or OPT. **(iii)** *Job arrival condition:* After a job arrives and gets dispatched, we re-match the machines. Φ may increase, yet the total increase due to all job arrivals is upped bounded by $O(G^*)$ (precisely, $(8+\frac{9}{\epsilon})\cdot G^*$). **(iv)** *Running condition:* At any other time t, $\frac{\mathrm{d}F_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} \le (1+\frac{1}{\epsilon})\cdot\frac{\mathrm{d}G^*(t)}{\mathrm{d}t} + \frac{1}{\epsilon}\cdot\frac{\mathrm{d}F_{\mathrm{s}}}{\mathrm{d}t}$. By integrating the above conditions over time, we have $F_{\mathrm{w}} \le (8+\frac{9}{\epsilon}+1+\frac{1}{\epsilon})G^* + \frac{1}{\epsilon}F_{\mathrm{s}}$. Then Lemma 1(i) follows.

Potential function Φ. For any machine i of POOL, for any $q \ge 0$, recall that $n_{i,t}(q)$ denotes the number of active jobs with remaining work at least q at time t. Define $n^*_{i,t}(q)$ similarly for OPT. We will drop the parameter t when t refers clearly to the current time. Let $(\cdot)_+ = \max(\cdot, 0)$. The potential function is

$$\Phi(t) = \textstyle\sum_{i=1}^{m} \Phi_i(t) \quad \text{where} \quad \Phi_i(t) = \frac{1}{\epsilon}\int_0^\infty \sum_{k=1}^{n_i(q)} (k - n^*_{x(i)}(q) + \mu)_+ \,\mathrm{d}q \ .$$

Machine re-matching. $x(1), \ldots, x(m)$ form a permutation of $1, 2, \cdots, m$. At any time once a new job has been dispatched to a machine by *ALG* and OPT, we keep swapping $x(i)$ and $x(j)$ as long as we find machines i and j satisfying:

- POOL has i not in P and OPT has $x(i)$ awake or procrastinating; and
- POOL has j in P and OPT has $x(j)$ sleeping.

Note that Φ_i, Φ_j and Φ may change after a swapping. Interestingly, we can verify that this change must be non-increasing.

Lemma 2. *After some $x(i)$ and $x(j)$ are swapped, Φ_i and Φ_j does not increase.*

We can easily show the boundary, job completion and state transition conditions. The running condition depends solely on the job scheduling policy (SRPT) and can be analyzed independently for each matched pair of machines using techniques for the single-machine analysis [5,11]; details will be given in full paper.

The core of the potential analysis is the arrival condition, which depends on both sleep management and job dispatching policies. Below we show that when a job arrives, after a special re-matching of machines, the increase of flow or technically Φ can be bounded in terms of some non-overlapping cost of OPT.

Lemma 3. *The sum over all jobs of the increase in Φ due to a job arrival (and machine re-matching) is at most $(8+\frac{9}{\epsilon})\cdot G^*$.*

At the point after a job j arrives and gets dispatched by POOL and OPT, we re-match their machines (i.e., compute a new matching function $x(i)$) before we re-calculate Φ. This re-matching process is for analysis sake and can make reference to any information in the POOL and OPT's schedule in the past or future. To formally define the inputs to the re-matching process, we need to first construct a list of "interesting" events ordered by time. There are 6 types of events: *JobArrive*(j)—a new job arrives and is dispatched by

POOL and OPT; POOL_In(j)—POOL adds a machine into the pool P due to job j; POOL_Out—removes a machine from P; OPT_Wake—OPT wakes up a machine; OPT_Sleep—OPT sleeps a machine; and *Rematch*—execute the re-matching procedure based on POOL and OPT's status as defined up to the previous event. When there are multiple events at the same time, they are arranged in the following order: POOL_Out, all OPT_Wake, *JobArrive*(j), *Rematch*, all POOL_In(j), *JobArrive*(j'), *Rematch*, all POOL_In(j'), $\cdots$, all OPT_Sleep. Let $A = (e_1, e_2, \ldots e_c)$ be the list of events. For each event e, we define h_e to be the number of machines in P immediately after e, and h^*_e the number of awake machines in OPT. We omit e when it is clear that we refer to the current event. Notice that before the first event of A, $h = 1$ and $h^* = 0$.

Lazy intervals. A *lazy interval* is a maximal sequence of events in A containing at least one *JobArrive*, in which all events e have $h_e \leq h^*_e$. By definition, a lazy interval must start with a POOL_Out or OPT_Wake event and end before a POOL_In or OPT_Sleep event. For any lazy interval ℓ, excluding the first event, let I_ℓ, O_ℓ, W^*_ℓ and S^*_ℓ be respectively the number of POOL_In, POOL_Out, OPT_Wake and OPT_Sleep events in ℓ. W.r.t. the first and the last event of a lazy interval, $h = h^*$. This implies the following useful property of a lazy interval.

Property 2. For a lazy interval ℓ, $I_\ell + S^*_\ell = W^*_\ell + O_\ell$.

Type-0, Type-1 and Type-2 jobs. Define Type-0 jobs to be jobs which POOL dispatches to a zero-*rwc* machine (i.e., no active jobs). For any other job, it is Type-1 if its *JobArrive* event e is in a lazy interval and $h_e < m$; otherwise, it is Type-2. Type-0 jobs are easy to analyze. Roughly speaking, Type-2 jobs arrive when POOL is using more machines than the OPT (or is using all m machines); after re-matching the machines, it is relatively easy to show that Φ has limited increase (see Lemma 4; proof will be given in the full paper). For Type-1 jobs, POOL might be using very few machines and POOL's increase in *rwc* can be way larger than OPT's. We analyze Type-1 jobs interval by interval (instead of job by job) and show that POOL's increase in *rwc* is bounded by the static energy and wakeup energy of OPT (see Lemma 5(b)). Then Lemma 3 follows.

Lemma 4. *The total increase in Φ due to Type-2 jobs is at most $\frac{1}{\epsilon} \cdot G^*$.*

For Type-0 and Type-1 jobs, it is relatively easy to see that the total increase in Φ is at most $(1 + \frac{1}{\epsilon})$ times the total increase in *rwc* of POOL. Thus, Lemma 5 implies that the increase in Φ due to Type-0 and Type-1 jobs is most $8(1+\frac{1}{\epsilon}) \cdot G^*$.

Lemma 5. (a) POOL*'s total increase in rwc due to Type-0 jobs is at most G^*.* (b) POOL*'s total increase in rwc due to Type-1 jobs is at most $7G^*$.*

Lemma 5(a) is obvious: when a Type-0 job arrives, POOL's increase in *rwc* cannot exceed that of OPT. To show Lemma 5(b), let $\Delta G'_\ell$ and $\Delta G'$ be the increase in *rwc* to POOL due to Type-1 jobs in a lazy interval ℓ and all Type-1 jobs, respectively. Let L be the set of all lazy intervals and let $|L|$ be the size of L. Define $I_L = \sum_{\ell \in L} I_\ell$ and similarly for O_L, W^*_L and S^*_L. It is useful to define

E^*_ℓ to be the total wake-up energy used by OPT during ℓ plus the static energy used by OPT during ℓ (precisely, during the time period enclosing all events in ℓ), and E^*_L to be the sum of E^*_ℓ over all $\ell \in L$. Obviously, $E^*_L \le G^*$. We can show Lemma 6 below; details will be given in the full paper. Roughly speaking, Lemma 6(i) and (ii) show respectively that POOL can wake up more machines in react to a large increase in working cost, and POOL does not put machines to sleep too frequently. Thus, POOL is not over-conservative when having fewer awake machines than OPT. Together with Property 2, Lemma 5(b) follows.

Lemma 6. *(i)* $\Delta G' < (I_L + 2|L|)\omega$*; (ii)* $(W^*_L + O_L)\omega \le E^*_L; |L|\omega \le 3E^*_L$.

3 Sleep Management and Speed Scaling

In this section, we consider the speed scaling model, where each processor can scale its speed s in $[0, \infty)$ and consumes energy at rate $P(s) = s^\alpha + \sigma$, where $\alpha > 1$. We adapt the algorithm POOL presented in Section 2 such that each awake processor in P scales its speed by AJC (active job count) [11], as follows. For machine i, define $n_{i,t}$ and $n_{i,t}(q)$ similarly as before.

> **Speed scaling in each machine of P:** When awake, if machine i has active jobs ($n_{i,t} > 0$), set its speed to $(n_{i,t} + \sigma)^{1/\alpha}$; else its speed is 0.

We can verify that the *rwc* of a machine i becomes $2\int_{q=0}^{\infty} \sum_{k=1}^{n_{i,t}(q)} (k+\sigma)^{1-1/\alpha}\, \mathrm{d}q$.

Theorem 2. *With speed scaling,* POOL *is* $O(\alpha)$*-competitive for flow plus energy.*

To prove Theorem 2, our main idea is similar to that in Section 2, as most properties of POOL do not depend on the power function and remains valid. In particular, we compare POOL with a restricted optimal algorithm OPT that keeps at most one machine procrastinating and follows SPRT and AJC. Such restriction allows us to calculate the *rwc* of OPT. In Section 4, we show that this only increases the competitive ratio by six times. We show Lemma 7 below which is analogous to Lemma 1 in Section 2. The sleeping flow F_{S}, idling energy E_{I} and wake-up energy U can be bounded using the same techniques, which gives Lemma 7 (ii). Yet in the speed scaling model, we can no longer bound E_{W} by E^*_{W} easily. Even though we restrict OPT to use AJC as the speed scaling policy, there is no simple relation between E_{W} and E^*_{W}. Thus, we will consider E_{W} and F_{W} together and analyze G_{W}, the working cost. Using a modified potential function, the potential analysis framework used in the fixed speed model is adaptable to the speed scaling model, allowing us to show Lemma 7(i).

Lemma 7. *In the speed scaling model,* (i) $G_{\mathrm{W}} \le 11\alpha \cdot G^* + (2\alpha - 2) \cdot F_{\mathrm{S}}$*;* (ii) $G \le 4G_{\mathrm{W}} + 4G^*$.

We now define the potential function Φ for proving Lemma 7(i). As shown in Section 2, we want Φ to capture POOL's remaining working cost in discounted in view of OPT's workload in the corresponding machines. We modify Φ in view of

the new *rwc* of POOL. At any time, let $x(i)$ to be the machine in OPT currently matched with machine i in POOL. Then we define

$$\Phi(t) = \sum_{i=1}^{m} \Phi_i(t) \quad \text{where} \quad \Phi_i(t) = 2\alpha \int_0^\infty \sum_{k=1}^{n_{i,t}(q)} (k - n^*_{x(i),t}(q) + \sigma)_+^{1-1/\alpha} \, \mathrm{d}q \ .$$

We will follow a similar framework in analyzing Φ. It is easy to show the boundary, job completion and state transition conditions. The arrival and running conditions can be proven using similar ideas but requires some modification mostly due to the new definition of *rwc*. We only state the arrival and running conditions and leave the detailed proofs in the full paper.

Lemma 8. (i) *The sum over all jobs of the increase in Φ due to a job arrival (and machine re-matching) is at most $9\alpha \cdot G^*$.* (ii) *Consider any time t without job arrival, completion, machine reordering and state transition in both* POOL *and* OPT. $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq 2\alpha \cdot \frac{\mathrm{d}G^*}{\mathrm{d}t} + (2\alpha - 2)\frac{\mathrm{d}F_{\mathrm{s}}}{\mathrm{d}t}$.

4 Transformation of Offline Schedule

This section describes the transformation for getting the suitable offline schedule.

Theorem 3. *Given any schedule S that uses speed scaling, we can transform S into another schedule S' that also uses speed scaling, and* (i) *S' has at most one procrastinating machine at any time,* (ii) *it uses SRPT for job selection and AJC for speed scaling, and* (iii) *the total cost of S' is at most 6 times that of S.*

We first transform S into a schedule S_0 with the following invariants: (1) If a job j is assigned to some machine i, then j is completed before i goes to sleep for the first time after the release of j (this invariant is mainly for simplifying analysis); and (2) it has at most one procrastinating machine at any time. The total cost of S_0 is at most 3 times that of S. Then, we show how to transform S_0 to S' that uses SRPT and AJC, and this will further blow up the cost by a factor of at most 2. Observe that we have a similar transformation for the fixed speed model, which blows up the total cost by a factor of 3 instead of 6: we transform S_0 to S' that uses SRPT, which does not further increase the cost.

We now give the essential ideas for proving Theorem 3; details will be given in the full paper. The construction of S_0 from S starts by copying the wakeup and sleeping times of each machine from S into S_0 (i.e., each machine has the same sequence of awake periods in both schedules). Then, we schedule each job into S_0 in order of release times; a job may be assigned to a different machine and different time slots than it is in S. Suppose we are considering job j, whose "execution profile" in S is $\langle i, (t_1, \ell_1, s_1), (t_2, \ell_2, s_2) \ldots, (t_m, \ell_m, s_m)\rangle$, meaning that j is assigned to i, running at time t_1 for ℓ_1 time units at speed s_1, and so on. Then, we copy this execution profile to S_0. If the resulting schedule violates an invariant, we do the following before moving on to schedule the next job.

Suppose Invariant (1) is violated, i.e., $[t_1, t_m + \ell_m]$ covers several awake periods of i. Then we "leftpack" the schedule as follows: Let u be the time machine i goes to sleep for the first time after the release of j. Then $u < t_m + \ell_m$.

We extend the awake period that ends at u as much as possible by executing j in this period. If j still cannot be completed before the next sleep time, we repeat this process. For example, suppose that j is released at time 8 and its execution profile in S is $\langle i, (12, 1, 1), (17, 6, 3), (30, 1, 1)\rangle$, and i's awake periods after 8 is $[10, 13], [17, 23], [29, 32]$. After leftpacking, the schedule for j in S_0 becomes $\langle i, (12, 1, 1), (13, 4, 3), (17, 2, 3), (23, 1, 1)\rangle$, and the awake periods of i after 8 is $[10, 24]$ and $[29, 32]$. The extra cost for executing j during $[13, 17]$ equals the cost saved by not executing j during $[19, 23]$, and during this period, i becomes idle and costs static energy; but this will at most double the original energy cost.

Now we briefly describe how to maintain Invariant (2). Before scheduling j, Invariant (2) ensures that S_0 has at most one procrastinating machine p at the release time u of job j. If $p = i$, the new schedule still satisfies Invariant (2). Suppose $p \neq i$, and after leftpacking, there is a period $[u, v]$ during which both p and i are procrastinating. If j's length (i.e. total execution time) is larger than $|[u, v]|$, we can wake up p earlier at u so that it becomes awake during $[u, v]$. Note that the extra static energy cost during this idle period is no greater than that for processing j and thus we double the energy at most. If the length of j is smaller than $|[u, v]|$, we re-assign j to p and execute it during $[v - \ell, v]$ where ℓ is the length of j. Now machine i no longer procrastinates during $[u, v]$ and hence Invariant (2) is satisfied. But there is a subtle problem here: it is possible that j completes before v in S, so in the new schedule j has an increased flow time. In such case, we employ a different strategy which involves moving multiple jobs from i to p, and the analysis becomes rather complicated (see the full paper).

Finally, we describe how to transform S_0 to S', which uses SRPT and AJC. Consider any machine i. Whenever i is awake in S_0, i is also awake in S', processing jobs using SRPT and AJC (or idles if no job remains). If i goes to sleep in S_0 at some time t, i also goes to sleep in S at time t only if it has no active jobs; otherwise it stays awake and processes the jobs using SRPT and AJC until there are no more active jobs, say at time t'. Then it copies the status of i at time t' in S_0, i.e., it goes to sleep in S' if and only if i is asleep in S_0 at time t'.

References

1. Albers, S.: Energy-efficient algorithms. CACM 53(5), 86–96 (2010)
2. Albers, S., Fujiwara, H.: Energy-efficient algorithms for flow time minimization. ACM Transactions on Algorithms 3(4), 49 (2007)
3. Andrew, L., Wierman, A., Tang, A.: Optimal speed scaling under arbitrary power functions. ACM SIGMETRICS Performance Evaluation Review 37(2), 39–41 (2009)
4. Avrahami, N., Azar, Y.: Minimizing total flow time and total completion time with immediate dispatching. In: Proc. SPAA, pp. 11–18 (2003)
5. Bansal, N., Chan, H.L., Pruhs, K.: Speed scaling with an arbitrary power function. In: Proc. SODA, pp. 693–701 (2009)
6. Chekuri, C., Goel, A., Khanna, S., Kumar, A.: Multi-processor scheduling to minimize flow time with ϵ resource augmentation. In: Proc. STOC, pp. 363–372 (2004)

7. Gandhi, A., Gupta, V., Harchol-Balter, M., Kozuch, M.: Optimality analysis of energy-performance trade-off for server farm management. Performance Evaluation 67(11), 1155–1171 (2010)
8. Gupta, A., Krishnaswamy, R., Pruhs, K.: Scalably scheduling power-heterogeneous processors. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 312–323. Springer, Heidelberg (2010)
9. Greiner, G., Nonner, T., Souza, A.: The bell is ringing in speed-scaled multiprocessor scheduling. In: Proc. SPAA, pp. 11–18 (2009)
10. Khuller, S., Li, J., Saha, B.: Energy efficient scheduling via partial shutdown. In: Proc. SODA, pp. 1360–1372 (2010)
11. Lam, T.W., Lee, L.K., Ting, H.F., To, I., Wong, P.: Sleep with guilt and work faster to minimize flow plus energy. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 665–676. Springer, Heidelberg (2009)
12. Lam, T.W., Lee, L.K., To, I., Wong, P.: Competitive non-migratory scheduling for flow time and energy. In: Proc. SPAA, pp. 256–264 (2008)
13. Lam, T.W., Lee, L.K., To, I., Wong, P.: Speed scaling functions for flow time scheduling based on active job count. In: Halperin, D., Mehlhorn, K. (eds.) Esa 2008. LNCS, vol. 5193, pp. 647–659. Springer, Heidelberg (2008)
14. Pruhs, K., Sgall, J., Torng, E.: Online scheduling. In: Handbook of Scheduling: Algorithms, Models and Performance Analysis, pp. 15-1–15-41. CRC Press, Boca Raton (2004)
15. U.S. Environmental Protection Agency. EPA Report on server and data center energy efficiency (2007)
16. Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced CPU energy. In: Proc. FOCS, pp. 374–382 (1995)