A Tutorial on Amortized Local Competitiveness in Online Scheduling

Sungjin Im [*] *Benjamin Moseley* [†]
Computer Science Department
University of Illinois Urbana-Champaign

Kirk Pruhs [‡]
Computer Science Department
University of Pittsburgh

May 3, 2011

1 Introduction

Recently the use of potential functions to analyze online scheduling algorithms has become popular [19, 7, 29, 13, 31, 4, 30, 3, 21, 15, 14, 28, 12, 2, 5, 6, 9, 11, 23, 33, 24, 8, 17, 16, 25, 1, 20, 26, 22, 18]. These potential functions are used to show that a particular online algorithm is locally competitive in an amortized sense. Algorithm analyses using potential functions are sometimes criticized as seeming to be black magic as the formal proofs do not require, and commonly do not contain, any discussion of the intuition behind the design of the potential function. Sometimes, as in the case for the first couple uses of potential functions in the online scheduling literature, this is because the authors arrived at the potential function by trial and error, and there was not a cohesive underlying intuition guiding the development. However, now that tens of online scheduling papers have used potential functions, one can see that a "standard" potential function has emerged that seems to be applicable to a wide range of problems. The use of this standard potential function to prove amortized local competitiveness can no longer be considered to be magical, and is a learnable technique. Our main goal here is to give a tutorial teaching this technique to readers with some modest prior knowledge of scheduling, online problems, and the concept of worst-case performance ratios.

Online Scheduling: We consider online scheduling problems where jobs/tasks arrive at a server (e.g. a web server, a database server, an operating system, etc.) over time. Throughout this paper N will denote the total number of jobs and jobs are indexed $J_1, J_2, \ldots, J_N$. If there is more than one unfinished task at a particular time, then the online scheduler A must determine which task to process. In the schedule $A(I)$ that results on input I, each job (or more strictly speaking the client that submitted the job) has a resulting quality of service. For example, the most common quality of service measure for a job is the flow/response time F_i which is the difference between time C_i, when job J_i is completed, and time r_i, when job J_i arrived in the system. One can then measure the quality of the schedule by combining the quality of service measures of the jobs in some way. For example, the most common quality of service measure for a schedule is the average flow time, $\sum_{i\in[N]} F_i/N$.

Competitiveness: An online scheduler A does not have knowledge of the future and, due to this, in most settings it is not possible for the online scheduler to guarantee that the resulting schedule is optimal for the quality of service measure under consideration. Thus one generally seeks an algorithm guaranteeing that

[*]Partially supported by NSF grants CCF-0728782 and CCF-1016684.
[†]Partially supported by NSF grants CCF-0728782 and CCF-1016684.
[‡]Supported in part by NSF grant CCF-0830558 and an IBM Faculty Award.

the degradation in the quality of service measure relative to some benchmark is modest/minimal/bounded. The most obvious benchmark is the optimal schedule $OPT(I)$. For a minimization problem, an algorithm is said to be c-competitive, or have competitive ratio c, if

$$\max_I \frac{G_a(I)}{G_o(I)} \leq c$$

where in this setting $G_a(I)$ and $G_o(I)$ refer to the value of the scheduling objective in the algorithm's schedule and the optimal solution's, respectively. The paper that is perhaps most responsible for popularizing this line of online scheduling research is [34]. For surveys see [38, 37].

In many settings, it is not possible for the scheduler to have bounded competitiveness relative to the optimal schedule. In such cases, a commonly used weaker benchmark than the optimal schedule is the optimal schedule on a slightly slower processor. An algorithm A is s-speed c-competitive if the algorithm is c-competitive given a processor s times faster than the optimal solution's processor. This is called resource augmentation analysis, and was introduced in [27], and the standard terminology was later coined in [36]. To understand the motivation for resource augmentation analysis, note that it is common for systems to possess the following (informally defined) *threshold property*: The input or input distributions can be parameterized by a load λ, and the system is parameterized by a capacity μ. The system then has the property that its QoS would be very good when the load λ is at most 90% of the system capacity μ, but degrades unacceptably if λ exceeds 110% of μ. Figure 1 gives such an example of the QoS curve for a system that has this kind of threshold property. Figure 1 also shows the performance of an online algorithm A and an optimal algorithm is similar in the sense that they have close thresholds. Notice however that the competitive ratio of A relative to the optimal is very large when the load is near capacity μ. Another natural way of comparison would be to compare the performance of A to the optimal with the load that is s times the load that A is given. Notice that multiplying the load by a factor of s is equivalent to slowing the system down by a factor of s. Hence we would like to compare A with an s times faster processor to the optimal schedule.

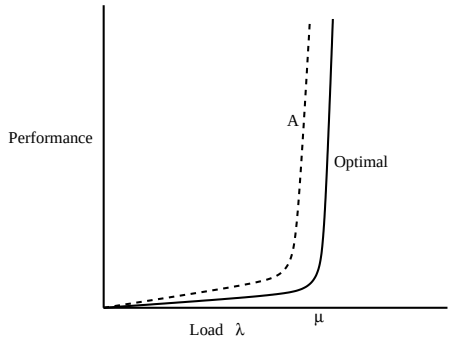

Figure 1: QoS curves of an scalable online algorithm A and the optimal algorithm for a system with the threshold property.

The informal notion of an online scheduling algorithm A being "reasonable" is then generally formalized as A having bounded competitiveness for some small constant speed augmentation s. Usually the ultimate goal is to find an algorithm where the speed augmentation required to achieve $O(1)$-competitiveness is arbitrary close to one. Such an algorithm is called *scalable*. Intuitively, a scalable algorithm performs very close to the optimal schedule, since it would guarantee a system capacity arbitrarily close to the optimal capacity, while also ensuring that the QoS remains comparable (within a constant factor).

It is instructive to note the similarities and differences between results obtained from competitive analysis and analogous results in the queuing theory literature. For example, we will show later that the algorithm

Shortest Job First (SJF) is scalable, more precisely $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive, for the objective of average flow time. A standard result from the queuing theory literature is that for an $M|M|1$ queue the average flow time for SJF (or any scheduler) with a unit speed processor is at most $\frac{1}{\epsilon}$ if the load is at most $1-\epsilon$ [32]. Both results can be interpreted as stating that if the system is not too heavily loaded then the performance of the scheduler should be reasonable. In the case of competitive analysis the measure of reasonable is relative and, in the queuing theory analysis, the measure of reasonable is absolute.

1.1 Notation

Consider an objective and a problem instance. Fix an algorithm for this problem and an optimal solution. Throughout this paper A will denote the algorithm. Let $A(t)$ and $O(t)$ denote the unsatisfied jobs at time t in the algorithm's and optimal solution's queue, respectively. The total number of jobs is denoted as N. The completion time of job J_i is C_i^a and C_i^o in the algorithm's schedule and optimal schedule, respectively. The arrival time of job J_i is r_i. The work/size of job J_i is p_i. If the jobs have weights then w_i denotes the weight of job J_i and $d_i = \frac{w_i}{p_i}$ is the density of J_i. Let $G_a(I)$ and $G_o(I)$ denote the algorithm's objective and the optimal objective. The (I) will be dropped when the instance is clear. The value of $p_i^a(t)$ and $p_i^o(t)$ denote the remaining work of job J_i in the algorithm's schedule at time t and the optimal schedule, respectively.

1.2 Objective Functions

As mentioned, the goal of the scheduler is to optimize the quality of service that the jobs receive. Recall that the flow time of a job J_i is $C_i^a - r_i$, the total time the job waits to be satisfied in the system. One of the most popular objectives considered is minimizing the total (equivalently average) flow time $\sum_{i\in[N]}(C_i^a - r_i)$. This objective is equivalent to minimizing $\int_{t=0}^{\infty} |A(t)|\mathtt{dt}$. Another popular objective is minimizing the total weighted flow time. In this setting each job J_i has a weight w_i and the goal is to minimize $\sum_{i\in[N]} w_i(C_i^a - r_i)$. This objective is equivalent to minimizing $\int_{t=0}^{\infty} w_a(t)\mathtt{dt}$, where $w_a(t)$ is the sum of the weights of the unsatisfied jobs at time t in the schedule.

1.3 Fractional Objectives

A popular technique used in obtaining a resource augmented competitive online algorithm for an (integral) objective is to first obtain an algorithm that is competitive for a fractional objective. The fractional objective for an algorithm A is defined to be $\sum_{i\in[N]} \int_{t=0}^{\infty} \frac{p_i^a(t)}{p_i} \frac{\mathtt{d}G_a(t,J_i)}{\mathtt{dt}}\mathtt{dt}$ where $G(t,J_i)$ is the total cost of job J_i up to time t. The fractional objective is usually considered when $G(t,J_i)$ depends only on the flow time of job J_i. As an example. consider the objective of weighted flow. The fractional weighted flow time of a job J_i is $\int_{r_i}^{C_i} w_i \frac{p_i^a(t)}{p_i}\mathtt{dt}$ where $p_i^a(t)$ is the remaining work of job J_i by algorithm A at time t. We call $\frac{p_i^a(t)}{p_i}$ the remaining fraction of J_i at time t. The total weighted fractional flow time objective is $\int_{t=0}^{\infty} \sum_{J_i\in A(t)} w_i \frac{p_i^a(t)}{p_i}\mathtt{dt}$. An interpretation of the fractional weighted flow time objective is that a job contributes to the objective in proportion to the amount of remaining work the job has. Notice that the total fractional weighted flow time of any schedule is at most the integral weighted flow time of the schedule.

The concept of fractional objectives has proved to be useful for analyzing online scheduling algorithms. Generally it is easier for an online algorithm to be competitive for fractional objectives. Further, fractional objectives are often easier to reason about. To the best of our knowledge, the use of fractional objectives to aid in the analysis of online scheduling algorithms originates from [10]. It is generally possible to convert a scheduler A that is good for a fractional objective into an algorithm A' that is good for an integer objective

with minimal speed augmentation in the following way: The algorithm A' always schedules the exact same jobs as A at any time, except a $(1+\epsilon)$ factor faster in rate of speed, unless the job has been completed in A''s schedule. If A is s-speed c-competitive for a fractional objective then generally A' is $(1+\epsilon)s$-speed $O(c/\epsilon)$-competitive for the corresponding integer objective.

1.4 Popular Algorithms

We give a quick overview of some of the most popular scheduling algorithms.

- Shortest Remaining Processing Time First (**SRPT**): Always processes the job with the least remaining work.
- Shortest Job First (**SJF**): Always schedules the job with the least initial work.
- Highest Density First (**HDF**): Always schedules the job J_i such that $d_i = \frac{w_i}{p_i}$ is maximized.
- Round Robin (**RR**): At each instantaneous time processes all alive jobs equally.
- Shortest Elapsed Time First (**SETF**): Works on the job that has been processed the least. If there are ties the algorithm round robins the jobs that have been processed the least.
- First In First Out (**FIFO**): Always schedules the oldest job.
- Late Arrival Processor Sharing (**LAPS**): Shares the processing equally among the latest arriving constant fraction of the jobs.

2 Local Competitiveness

In this section we discuss an analysis technique known as local competitiveness. Until relatively recently, this has been the most popular technique used for worst case analysis of scheduling algorithms. Let G denote some objective and let $G_a(t)$ be the cumulative objective in the schedule for algorithm A up to time t. So $\int_0^\infty \frac{\mathsf{d}G_a(t)}{\mathsf{dt}}\mathsf{dt} = G_a$ is the final objective of A. For example, when G is total flow then $\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} = |A(t)|$ and $G_a(\tau) = \int_0^\tau |A(t)|\mathsf{dt}$. When G is the total fractional flow then $\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} = \sum_{J_i \in A(t)} \frac{p_i^a(t)}{p_i}$ and $G_a(\tau) = \int_0^\tau \sum_{J_i \in A(t)} \frac{p_i^a(t)}{p_i}\mathsf{dt}$. The algorithm A is said to be *locally c-competitive* if for all times t,

$$\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} \leq c \cdot \frac{\mathsf{d}G_o(t)}{\mathsf{dt}} \tag{1}$$

Most of the early competitive analyses of online scheduling algorithms used local competitiveness. For instance the following theorems were shown using local competitiveness. Generally, a proof of local competitiveness uses one of the following techniques: (1) Show by induction on time an invariant concerning the algorithm's queue and the optimal solution's queue, or (2) Fix an arbitrary time t and, by examining the history, show that optimal does not have have enough processing capability to prevent the online algorithm from being locally competitive at time t.

Theorem 1. *SRPT is optimal for total/average flow time.*

Sketch. One can prove this induction on time that the following invariant holds: for all k, the aggregate remaining work of the k jobs with the most remaining work time is always greater for SRPT than any other algorithm. □

Theorem 2. *[35] SRPT is 2-approximate for average stretch.*

Sketch. The stretch objective focuses on minimizing $\sum_{i\in[N]} \frac{1}{p_i}(C_i - r_i)$. This is the same as weighted flow time when $w_i = \frac{1}{p_i}$. One can prove by induction on time that the following invariant holds: the sum of the weights (which are reciprocals of the work) of the jobs which are not being scheduled in SRPT's queue is bounded by the total weight of the optimal solution's queue. □

Theorem 3. *[10] HDF is scalable for weighted flow time.*

Sketch. To show this theorem, first it is shown that HDF is locally optimal for fractional weighted flow time. The conversion between fractional and integral flow time can be used to get an algorithm that is scalable for integral weighted flow time. □

3 Examples Where Local Competitiveness Arguments Won't Work

Unfortunately, local competitiveness cannot be used for many online scheduling problems. This is illustrated by the following examples.

Broadcast Scheduling: Consider the broadcast scheduling problem with the objective of average flow. In the broadcast problem there is a set of pages stored at the sever. Requests for pages arrive over time. In a unit time step the server can broadcast a page and all requests for the page that have arrived to the server are satisfied by the single broadcast because clients are assumed to be connected to the sever via a multicast channel. To see why local competitiveness will not work in this setting even when the algorithm is given $(1+\epsilon)$ resource augmentation, consider the following adversarial strategy. There are n requests that arrive at time 0, each for a distinct page. At time $n/2$, a new request arrives for each of the $n(1+\epsilon)/2$ pages that the online algorithm A has previously broadcasted. Thus by broadcasting the pages in the opposite order of A, it is possible to have satisfied all requests by time $(1+\epsilon)n$, but A still has about $n/2$ unsatisfied requests at this time. Therefore, at time $t = (1+\epsilon)n$ we have $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} = \Omega(n)$ and $\frac{\mathtt{d}G_o(t)}{\mathtt{dt}} = 0$.

Speed Scaling: In the speed scaling scheduling setting, the processor speed can be dynamically scaled over time. The power, energy used per unit time, of the processor is a convex function of the speed. In this setting, the scheduler needs to set the speed of the processor. Consider the objective of total flow time plus the energy. Intuitively, the optimal scheduler will spend one unit of energy if it results in a decrease in at least one unit of flow time. In this case, $\frac{\mathtt{d}G(t)}{\mathtt{dt}}$ is power used at time t plus the number of unfinished jobs. It is obviously not possible for a local competitiveness argument to work in this setting as one has to deal with the possibility that the optimal solution has previously finished all the jobs, by using a lot of power in the past, and is currently idling, so that the local cost for the benchmark schedule is zero.

Arbitrary Speed Up Curves: This is a popular parallel scheduling model. Due to space constraints, we give only a simplified version of the model; for the general model, see [21, 23]. Say we have M processors/cores, and each job J_i is either parallel or sequential. If job J_i is parallel, it is processed at rate M' when assigned M' cores. A job is sequential if it is processed at a rate of one regardless of how many cores it is assigned to. It is assumed that the algorithm does not know the parallelizability of each job (parallel or sequential) nor its work. Consider the objective of total flow time and the following instance. At time

0, suppose $M-1$ unit sized sequential jobs and one parallel job with size M are released. Clearly, there is a schedule that completes all jobs by time 1 by allocating all cores to the parallel job. Now consider any scheduling algorithm with a $(1+\epsilon)$ resource augmentation. Since the algorithm does not know which job is parallel, it must have wasted most of its processing capabilities working on sequential jobs. So at time 1, the algorithm has the parallel job left. Recall that the optimal schedule does not have any job at time 1. Thus the algorithm's queue size cannot be bounded by the optimal solution's queue size at time 1.

4 Amortized Local Competitiveness

For problems where local competitiveness is not possible, one alternative form of analysis is amortized local competitiveness. To prove that an online scheduling algorithm A is c-competitive using an amortized local competitiveness argument, it suffices to give a potential function $\Phi(t)$ such that the following conditions hold.

Boundary condition: Φ is zero before any job is released and Φ is non-negative after all jobs are finished.

Completion condition: Summing over all job completions by the optimal solution and the algorithm, Φ does not increase by more than $\beta \cdot G_o$ for some $\beta \geq 0$. Most commonly $\beta = 0$.

Arrival condition: Summing over all job arrivals, Φ does not increase by more than $\alpha \cdot G_{OPT}$ for some $\alpha \geq 0$. Most commonly $\alpha = 0$.

Running condition: At any time t when no job arrives or is completed,

$$\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} + \frac{\mathsf{d}\Phi(t)}{\mathsf{dt}} \leq c \cdot \frac{\mathsf{d}G_o(t)}{\mathsf{dt}} \tag{2}$$

Integrating these conditions over time one gets that $G_a - \Phi(0) + \Phi(\infty) \leq (\alpha+\beta+c) \cdot G_o$ by the boundary, arrival and completion conditions. Note that when Φ is identically 0, equation (2) is equivalent to the local competitiveness equation (1). Generally the value of the potential $\Phi(t)$ depends only on the state (generally how much work is left on each of the jobs) of the online algorithm and the optimal schedule at time t.

The value of the potential function can be thought of as a bank account. If the increase in the online algorithm's objective, $\frac{\mathsf{d}G_a(t)}{\mathsf{dt}}$, is less than c times the increase in the benchmark's objective, $\frac{\mathsf{d}G_o(t)}{\mathsf{dt}}$, then the algorithm can save some money in the bank. Otherwise, it withdraws some money from the bank to pay for its cost, $\frac{\mathsf{d}G_a(t)}{\mathsf{dt}}$. Because of the boundary condition that guarantees a non-negative deposit at the end, the total amount of money that the algorithm withdraws never exceeds its total deposit.

The concept of using a potential function to prove competitiveness goes back at least to the seminal papers by Sleator and Tarjan [39, 40]. The first use of a potential function to use a amortized local competitive argument was in [7]; although, the origination of the idea traces back to [19]. [19] contains a "potential function" but the potential at time t depends not only on the states of the online algorithm and the optimal schedule at time t, but also on the future job arrivals and future schedules. So arguably the amortization argument in [19] is probably closer to a charging argument than to a potential function argument.

Before introducing the standard potential function, it is instructive to consider an intuitive candidate potential function that does not work: the potential is the online algorithm's future cost minus c times the optimal solution's future cost, assuming no more jobs arrive. Further consider this potential within the context of the problem of scheduling unit jobs with restricted assignment on uniform parallel machines with the objective of total fractional flow time. Here there are a set of machines and each job is restricted to

be scheduled on some subset of the machines. We will use the notation x and y to refer to machines. Let $m_{a,x}$ denote the total remaining work of jobs that are assigned to machine x in the algorithm A's schedule. Likewise define $m_{o,x}$ for the optimal schedule. At any time t, $m_{a,x}(t)^2$ is roughly the online algorithm's future cost at time t for jobs assigned to machine x assuming that no more jobs arrive and the algorithm is given 1 speed. Usually, the estimate of the future cost used in the potential function assumes that the algorithm is given no resource augmentation. Thus the candidate potential function in this setting would be

$$\sum_x (m_{a,x}^2(t) - cm_{o,x}^2(t)) \tag{3}$$

where c is some constant. Although this potential function is based on the right intuition, this potential will not satisfy the arrival condition. To see this consider when job J_i arrives and is assigned to machine x by the algorithm and y by the optimal schedule. The change in the potential is $(m_{a,x}(t)+1)^2 - m_{a,x}^2(t) - c(m_{o,y}(t)+1)^2 + cm_{o,y}^2(t)$. The term $m_{a,x}(t)$ can be much larger than $m_{o,y}(t)$, making the increase in the potential not bounded by OPT.

We are now ready to introduce, the standard potential function. The first use of a potential function that clearly was of this standard form was in [9].

Semi-Formal Definition of the Standard Potential Function: *The potential is the future online cost assuming that (1) no more jobs arrive and (2) that each job size is the lag for that job, that is, how far the online algorithm is behind the optimal solution schedule in the work processed.*

Let us return to the problem of scheduling unit jobs with restriction on uniform machines with the objective of total fractional flow time. The standard potential function in this case is then:

$$c\sum_x (m_{a,x}(t) - m_{o,x}(t))^2 \tag{4}$$

Comparing the standard potential function in (4) to the the potential function in (3) one can see that in some sense the standard potential function is the "cost of the difference", instead of "the difference in the costs". We consider the algorithm $Greedy$ that assigns an arriving job to the machine x such that $m_{a,x}$ is minimized, and that schedules jobs in a FIFO fashion on each machine. In the following theorem we show how this potential function can be used to prove the scalability of $Greedy$. This is a special case of a more general result for scheduling on unrelated processors in [11].

Theorem 4. *The algorithm $Greedy$ is scalable, more specifically $(1+\epsilon)$-speed $(2+\frac{2}{\epsilon})$-competitive, for scheduling unit jobs with restricted assignment constraints on uniform machines with the objective of fractional average flow time.*

Proof. For the analysis we will compare $Greedy$ against a fixed optimal schedule that schedules each job on exactly one machine. We will further assume without loss of generality that the optimal solution runs the jobs assigned to a machine in FIFO order. Consider the potential function (4). We now consider various cases.

Boundary Conditions: The boundary conditions are trivially satisfied as the initial potential is clearly zero, and the potential is never negative.

Completion Condition: Consider when a job J_j is completed by the algorithm at time t and say that this job is assigned to machine x in the algorithm's schedule. When this job is completed, its remaining work is 0. Therefore this job is no longer contributing to $m_{a,x}(t)$. Thus there is no change in $\Phi(t)$. The case when the optimal solution completes a job is similar.

Arrival Condition: Now consider when a job J_i arrives and the algorithm assigns this job to machine x and the optimal solution assigns the job to machine y. The change in the potential due to the algorithm's assignment is $\frac{1}{\epsilon}(m_{a,x}(t) - m_{o,x}(t) + 1)^2 - \frac{1}{\epsilon}(m_{a,x}(t) - m_{o,x}(t))^2 = \frac{1}{\epsilon}(2m_{a,x}(t) - 2m_{o,x}(t) + 1)$. The change in the potential due to the optimal solution's assignment is $\frac{1}{\epsilon}(m_{a,y}(t) - m_{o,y}(t) - 1)^2 - \frac{1}{\epsilon}(m_{a,y}(t) - m_{o,y}(t))^2 = -\frac{1}{\epsilon}(2m_{a,y}(t) - 2m_{o,y}(t) - 1)$. We know that $m_{a,x}(t) \leq m_{a,y}(t)$ because the algorithm assigns a job to the machine which has the least load on it and the job could have been scheduled on either machine x or machine y. Hence, the increase in potential is at most $\frac{2}{\epsilon}(m_{o,y}(t) + 1)$. This is exactly $\frac{2}{\epsilon}$ times the flow time of job J_i in the optimal solution schedule, since the optimal solution is assumed to be running FIFO and it assigned J_i to machine y. Thus, by summing over all job arrivals, the increase is at most $\frac{2}{\epsilon}$ times the optimal cost.

Running Condition: Consider a time interval $[t, t + \mathtt{dt}]$. For a machine x, the change in the potential due to the algorithm's processing is $-\frac{2}{\epsilon}(1 + \epsilon)(m_{a,x}(t) - m_{o,x}(t))\mathtt{dt}$ since the algorithm processes a job on machine x at a speed of $(1 + \epsilon)$. The change due the optimal solution's processing on machine x is $\frac{2}{\epsilon}(m_{a,x}(t) - m_{o,x}(t))\mathtt{dt}$ since the optimal solution processes a job at a speed of 1 on machine x. Thus the total change in potential for the jobs due to the algorithm's and optimal solution's processing on machine x is $-2(m_{a,x}(t) - m_{o,x}(t))\mathtt{dt}$. The value of $m_{a,x}(t)\mathtt{dt}$ is exactly the fractional increase in the algorithm's objective during $[t, t + \mathtt{dt}]$ for jobs assigned to machine x in its schedule; likewise, $m_{o,x}(t)\mathtt{dt}$ is the fractional increase in the optimal solution's objective for jobs assigned to machine x in the optimal schedule. By summing over all machines, we have $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} + \frac{\mathtt{d}\Phi(t)}{\mathtt{dt}} = \sum_x (m_{a,x}(t) - 2(m_{a,x}(t) - m_{o,x}(t))) \leq \sum_x 2m_{o,x}(t) = 2m_o(t) \leq 2 \cdot \frac{\mathtt{d}G_o(t)}{\mathtt{dt}}$. Combining each of the conditions shows that $Greedy$ is $(1+\epsilon)$-speed $(2 + 2/\epsilon)$-competitive. □

As another warm-up example of an amortized local competitiveness argument, we prove that HDF is scalable for fractional weighted flow. Note that this analysis is weak as HDF is in fact optimal.

Theorem 5. *The algorithm HDF is scalable for the objective of fractional weighted flow.*

Proof. For simplicity, assume that all jobs have distinct densities. Recall that the density d_i of job J_i is $\frac{w_i}{p_i}$. Let $z_i(t) = p_i^a(t) - p_i^o(t)$ be the lag for the online algorithm on job J_i. The potential function is defined as follows:

$$\Phi(t) = \frac{1}{\epsilon} \sum_{J_i \in A(t) \cup O(t)} d_i z_i(t) \sum_{\substack{J_j \in A(t) \cup O(t) \\ d_j \geq d_i}} z_j(t)$$

Notice that if no jobs were to arrive in the future, $\sum_{J_i \in A(t)} d_i p_i^a(t) \sum_{J_j \in A(t), d_j \geq d_i} p_j^a(t)$ is the future cost of HDF at time t if HDF was given 1-speed. To see this, note that $d_i p_i^a(t)$ is the fractional weight for job J_i, and $\sum_{J_j \in A(t), d_j \geq d_i} p_j^a(t)$ is the work of the higher density jobs that it will have to wait for. Also, note that it is possible for the potential function to be negative. The boundary condition is easy to check. When a job J_i arrives, the value of $\Phi(t)$ does not change, since $z_i(t) = 0$. When a job J_i is finished by both A and OPT, the terms for job J_i disappear, but they have value 0. Hence job completion does not change the potential function value. We now turn to the running condition. We will give a sketch only for the most interesting case: $d_k > d_{k'}$ where J_k and $J_{k'}$ are the jobs that A and OPT are working on at the current time t respectively; the other cases are left as exercises. Note that $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} = \sum_{J_i \in A(t)} d_i p_i^a(t)$ and

$\frac{\mathsf{d}G_o(t)}{\mathsf{dt}} = \sum_{J_i \in O(t)} d_i p_i^o(t)$. The rate of change of $\Phi(t)$ due to A's processing J_k is

$$\begin{aligned} & -\frac{1+\epsilon}{\epsilon}\Big[d_k \sum_{\substack{J_j \in A(t)\cup O(t) \\ d_j \geq d_k}} (p_j^a(t) - p_j^o(t)) + \sum_{\substack{J_i \in A(t)\cup O(t) \\ d_i \leq d_k}} d_i(p_i^a(t) - p_i^o(t))\Big] \\ \leq & -\frac{1+\epsilon}{\epsilon}\Big[\sum_{J_i \in A(t)} d_i p^a(t) - \sum_{\substack{J_j \in O(t) \\ d_j \geq d_k}} d_k p_j^o(t) - \sum_{\substack{J_i \in O(t) \\ d_i \leq d_k}} d_i p_i^o(t)\Big] \\ \leq & -\frac{1+\epsilon}{\epsilon}\Big[\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} - \frac{\mathsf{d}G_o(t)}{\mathsf{dt}}\Big] \qquad [\text{ Since } d_j \geq d_k \text{ in the second summation}] \end{aligned} \tag{5}$$

On the other hand, the rate of change of $\Phi(t)$ due to OPT's processing $J_{k'}$ is

$$\begin{aligned} & \frac{1}{\epsilon}\Big[d_{k'} \sum_{\substack{J_j \in A(t)\cup O(t) \\ d_j \geq d_{k'}}} (p_j^a(t) - p_j^o(t)) + \sum_{\substack{J_i \in A(t)\cup O(t) \\ d_i \leq d_{k'}}} d_i(p_i^a(t) - p_i^o(t))\Big] \\ \leq & \frac{1}{\epsilon}\Big[\sum_{\substack{J_j \in A(t) \\ d_j \geq d_{k'}}} d_{k'} p_j^a(t) + \sum_{\substack{J_i \in A(t) \\ d_i \leq d_{k'}}} d_i p_i^a(t)\Big] \leq \frac{1}{\epsilon}\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} \end{aligned} \tag{6}$$

Hence we have that $\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} + \frac{\mathsf{d}\Phi(t)}{\mathsf{dt}} \leq (1+\frac{1}{\epsilon})\frac{\mathsf{d}G_o(t)}{\mathsf{dt}}$. The other cases left as exercises also give similar running conditions. Combining all conditions, we conclude that HDF is scalable.

Finally we note that in Equation (5), to simplify our analysis, we assumed that no job in $O(t)$ has density d_k. Likewise, in Equation (6) we assumed that no job in $A(t)$ has density $d_{k'}$. These simplifying assumptions can be easily removed by a careful calculation. □

We now turn to the speed scaling setting.

Theorem 6. *Assume that the processor uses power $P(s) = s^\alpha$, for some constant $\alpha > 1$, when running at speed s. Assume that all jobs are unit size. Then any non-idling algorithm that uses power equal to the current total fractional weight $m_a(t)$ is $O(1)$-competitive for the objective of minimizing fractional flow time plus energy.*

Proof. Note that the algorithm runs at the speed $(m_a(t))^{1/\alpha}$ by using power $m_a(t)$, and hence $\frac{\mathsf{d}G_a(t)}{\mathsf{dt}} = 2m_a(t)$. Knowing the the fractional weight of the jobs in the algorithm's queue is $m_a(t)$, and that the ratio $m_a(t)/(m_a(t))^{1/\alpha}$ is a rough estimate of time that the algorithm will have to spend to finish all jobs in $A(t)$, we can approximate the future cost of A as $m_a(t) \cdot m_a(t)/(m_a(t))^{1/\alpha} = (m_a(t))^{2-1/\alpha}$. Let $m_o(t)$ be the fractional remaining work of the jobs in the optimal solution's queue at time t. By replacing $m_a(t)$ with the lag $z(t) = \max(m_a(t) - m_o(t), 0)$ (and multiplying by a constant factor), we obtain the potential function:

$$\Phi(t) := 8(z(t))^{2-1/\alpha}$$

Notice that the lag in this case was chosen to be based on the entire queue and not on individual jobs.

It is easy to check the boundary, arrival and completion conditions. As for the running condition, we consider two cases: (a) $m_a(t) > 2m_o(t)$ and (b) $m_a(t) \leq 2m_o(t)$. We prove only the more interesting case (a) leaving the other case (b) as an exercise. Note that $\Phi(t) > 0$, and A's processing decreases $\Phi(t)$ in this

case because $m_a(t) > 2m_o(t)$. Let $P_o(t)$ denote the power used by OPT at time t. The change rate of $\Phi(t)$ due to OPT's processing is $\frac{\mathsf{d}\Phi(t)}{\mathsf{d}m_o(t)}\frac{\mathsf{d}m_o(t)}{\mathsf{d}t} \leq 16(m_a(t))^{1-1/\alpha}(P_o(t))^{1/\alpha}$. We further consider two cases: $P_o(t) < \frac{m_a(t)}{2^\alpha}$ or not. For the second case $P_o(t) \geq \frac{m_a(t)}{2^\alpha}$, by ignoring the decrease of $\Phi(t)$ due to A's processing, we have $\frac{\mathsf{d}G_a(t)}{\mathsf{d}t} + \frac{\mathsf{d}\Phi(t)}{\mathsf{d}t} \leq 2^{\alpha+1}P_o(t) + 2^{\alpha+3}P_o(t) \leq 2^{\alpha+4}P_o(t) \leq 2^{\alpha+4}\frac{\mathsf{d}G_o(t)}{\mathsf{d}t}$. We now consider the first case $P_o(t) < \frac{m_a(t)}{2^\alpha}$. Noticing that OPT runs at a rate of at most $(m_a(t))^{1/\alpha}/2$, we have $\frac{\mathsf{d}\Phi(t)}{\mathsf{d}t} = \frac{\mathsf{d}\Phi(t)}{\mathsf{d}z(t)}\frac{\mathsf{d}z(t)}{\mathsf{d}t} \leq 8(2-1/\alpha)(z(t))^{1-1/\alpha}(-(m_a(t))^{1/\alpha}+(m_a(t))^{1/\alpha}/2) \leq -8(m_a(t)/2)^{1-1/\alpha}(m_a(t))^{1/\alpha}/2 \leq -2m_a(t)$. Hence it follows that $\frac{\mathsf{d}G_a(t)}{\mathsf{d}t} + \frac{\mathsf{d}\Phi(t)}{\mathsf{d}t} \leq 2m_a(t) - 2m_a(t) \leq 0$. □

We now give an analysis of Round Robin that is a variation of the analysis in [21].

Theorem 7. *RR is $(2+\epsilon)$-speed $O(1)$-competitive for the objective of total integral flow time.*

Proof. Let $z_i(t) = \max\{p_i^a(t) - p_i^o(t), 0\}$ be the lag of the online algorithm on job J_i. Then consider the potential function

$$\Phi(t) := \frac{4}{\epsilon}\sum_{J_i,J_j \in A(t)} \min\{z_i(t), z_j(t)\}$$

Notice that this potential function is always non-negative. It is worth noting that $\sum_{J_j \in A(t)} \min\{p_i^a(t), p_j^a(t)\}$ is exactly the amount of time that job J_i has to wait to be completed by RR without resource augmentation if no more jobs arrive in the future. Thus, summing this over all jobs gives the future cost. As usual, in obtaining $\Phi(t)$, the remaining work $p_i^a(t)$ is replaced with the lag $z_i(t)$.

The boundary condition is trivially satisfied. Job arrivals do not change the potential function value. Job completion does not increase $\Phi(t)$. Recall that for integral flow time, $\frac{\mathsf{d}G_a(t)}{\mathsf{d}t} = |A(t)|$ and $\frac{\mathsf{d}G_o(t)}{\mathsf{d}t} = |O(t)|$. For the running condition, we consider two cases: (a) $|O(t)| \leq \frac{\epsilon}{8}|A(t)|$ and (b) $|O(t)| \geq \frac{\epsilon}{8}|A(t)|$. We first consider case (a). Note that the term $\min\{z_i(t), z_j(t)\}$ in $\Phi(t)$ is non-zero if OPT completed both jobs J_i and J_j, i.e. $J_i, J_j \notin O(t)$. Hence there are at least $(1-\epsilon/8)^2(|A(t)|)^2$ positive terms $\min\{z_i(t), z_j(t)\}$, and RR decreases each positive term at a rate of $(2+\epsilon)/|A(t)|$. The change rate of $\Phi(t)$ due to RR's processing is at most $-\frac{4}{\epsilon}(1-\epsilon/8)^2(|A(t)|)^2(2+\epsilon)/|A(t)| \leq -(1+\frac{8}{\epsilon})|A(t)|$ when $0 < \epsilon \leq 1$. For simpler argument, we can assume that the optimal schedule works on only one job at time t. Since OPT can increase at most $2|A(t)|$ terms $\min\{z_i(t), z_j(t)\}$ in $\Phi(t)$ at a rate of 1, the maximum change rate of $\Phi(t)$ due to OPT's processing is $\frac{8}{\epsilon}|A(t)|$. Hence we obtain $\frac{\mathsf{d}G_a(t)}{\mathsf{d}t} + \frac{\mathsf{d}\Phi(t)}{\mathsf{d}t} \leq 0$. For case (b), we ignore decrease of $\Phi(t)$ due to RR's processing. Knowing that $\frac{\mathsf{d}\Phi(t)}{\mathsf{d}t} \leq \frac{8}{\epsilon}|A(t)|$ due to OPT's processing, we have $\frac{\mathsf{d}G_a(t)}{\mathsf{d}t} + \frac{\mathsf{d}\Phi(t)}{\mathsf{d}t} \leq \frac{8}{\epsilon}|O(t)| + \frac{64}{\epsilon^2}|O(t)| = \frac{72}{\epsilon^2}\frac{\mathsf{d}G_o(t)}{\mathsf{d}t}$. Combining all conditions together, we have shown that RR is $(2+\epsilon)$-speed $\frac{72}{\epsilon^2}$-competitive. □

The standard potential function is generally only useful when the online algorithm has some resource augmentation. To see why this is the case, consider the situation where optimal has a much lower current cost than the online algorithm, but the adversary is processing the same job as the online algorithm. Then because the online algorithm and optimal have the same speed processor, there is no change in the lags of any of the jobs. And thus there is no decrease in the potential to pay for the online algorithm's cost. In this situation, the online algorithm needs resource augmentation to decrease the lag on the job that it is running. The one example that we know where a potential function is used without resource augmentation, and one of the few examples of a potential function in the literature that is not of the standard form, occurs in [3] in the speed scaling setting for the objective of flow plus energy. [3] makes use of the fact that, given fixed processing resources, SRPT does not fall further behind optimal, even if SRPT is in a different state/configuration than optimal.

Notice that in the preceding examples there are some subtle differences in the potential functions, although all potential functions are designed around estimating the algorithm's future cost as if the lags of the jobs are their sizes. Two issues usually arise when designing the potential function. First is determining whether to allow the lags to go negative. For instance, in Theorem 5 the lags are allowed to go negative, while in Theorem 7 the lags are always kept non-negative. Roughly speaking, the issue here is that if the lags go negative then the potential can increase on the arrival and completion of jobs. However, if the potential does not go negative, then the running condition for some algorithms is difficult to show. The other subtle difference is determining which jobs are summed over in the potential function. In Theorem 5 the summation is over all jobs in the algorithm's queue and the optimal solution's queue, while in Theorem 7 the summation is only over jobs in the algorithm's queue. The question of which jobs to take the summation over is intimately tied to whether or not the potential can go negative, how the potential changes on job arrival and completion, and the algorithm considered. In general these two conditions need to be tailored to the specific problem and algorithm. It would be useful to have a better understanding of these two issues.

As an illustration of these issues, consider changing the potential function from the proof of Theorem 5:

$$\Phi(t) = \frac{1}{\epsilon} \sum_{J_i \in A(t)} d_i z_i(t) \sum_{J_j \in A(t), d_j \geq d_i} z_j(t)$$

Recall that $z_j(t) = p_j^a(t) - p_j^o(t)$. Note that this potential function can be negative, and only sums over jobs in the algorithm's queue. Now when a job J_i is completed by HDF, the change in the potential function is roughly $-\frac{1}{\epsilon} z_i(t) \sum_{J_j \in A(t)} d_j z_j(t)$ since HDF always completes the highest density job in its queue. However, the z variables can be negative or positive and therefore this could be a positive increase in the potential that is not straightforward to bound without knowing the structure of the optimal solution. One way around this would be to not allow the z variables to go negative, that is $z_j(t) = \max\{p_j^a(t) - p_j^o(t), 0\}$. Now there is no increase on the arrival and completion of jobs. However, the running condition causes issues. Consider the case where the optimal solution has only one job J_j at time t and the algorithm has a lot of jobs. In this case, the algorithm cannot charge to the optimal solution's local cost and must use the potential function. Further, assume that J_j is the job with the highest density in HDF's schedule and J_j has been processed a lot by the algorithm and none by the optimal solution. In this case $z_j(t) = 0$. Therefore, at time t HDF will process J_j and it will not change the variable $z_j(t)$. Thus there is no negative change in the potential function at time t and HDF cannot charge to the optimal solution's local cost. The running condition cannot be shown in this setting.

5 More Examples and Exercises

Here we give several exercises of algorithm analysis that can be performed by a relatively straightforward amortized local competitiveness argument using the standard potential function.

Weighted Round Robin: At each time, the algorithm Weighted Round Robin (WRR) distributes its processing to jobs in proportion to their weight. Formally, WRR processes job J_i at a rate of $w_i / \sum_{i \in A(t)} w_i$. Show that WRR is $(2+\epsilon)$-speed $O(1/\epsilon)$-competitive for minimizing the total weighted flow time.

Another Variation of Round Robin: If a job j has been released but not completed at time t, then define $\texttt{rank}(j)$ to be one more than the number jobs released strictly before r_j that are still unfinished at time t. For simplicity assume that no two jobs are released at the same time. Consider the scheduling algorithm

that shares the processing proportionally to the rank of the released but unfinished jobs. So if there are 3 released but unfinished jobs, the earliest arriving of these jobs would get 1/(1+2+3) fraction of the processor, the second arriving job would get 2/(1+2+3) fraction of the processor, and the latest arriving job would get 3/(1+2+3) fraction of the processor. Show that the average flow time for this scheduling algorithm is $O(1)$-competitive with the optimal scheduler with as little speed augmentation as possible. (Hint: Use the potential function $\Phi(t) := \sum_{i \in A(t)} z_i(t)\mathsf{rank}(j)$ where $z_i(t) = \max\{p_i^A(t) - p_i^O(t), 0)\}$.)

Greedy on Related Machines: Consider the problem of scheduling unit sized jobs on related machine to minimize the average fractional flow time. In the related machine setting, each machine x runs at some speed s_x and any job can be scheduled on any machine. The total processing a job requires on machine x is $\frac{1}{s_x}$. For this problem, consider the $Greedy$ algorithm that always assigns a job J_i to the machine x such that $\frac{m_{a,x}(t)}{s_x}$ is minimized when J_i arrives at time t. For the jobs assigned to a machine, $Greedy$ schedules the jobs in a FIFO order. Fix the optimal nonmigratory schedule, and assume the optimal solution processes jobs in FIFO order on each machine. Generalize the potential function given in the proof of Theorem 4 to this setting and show that the $Greedy$ algorithm is scalable.

SJF on Uniform Machines: Consider the problem of scheduling varying sized jobs on M identical machines to minimize the total fraction flow time. For this problem consider the algorithm SJF that at anytime schedules the M jobs with shortest original work on the M machines. Here a job cannot be processed simultaneously on two machines, but migration between processors is allowed. The goal is to show that SJF is $O(1)$-competitive with as little resource augmentation as possible. Let $V_{a,j}(t) = \sum_{J_i \in A(t), p_i \leq p_j} p_i^a(t)$ be the total remaining work of jobs with original work less than p_j at time t in the algorithm's schedule. Let $V_{o,j}(t)$ be defined similarly for OPT. Consider the potential function

$$c \sum_{j \in A(t)} \frac{p_j^a(t)}{p_j} \frac{1}{M}(V_{a,j}(t) - V_{o,j}(t)) + p_j^a(t)$$

where $c \geq 1$ is some constant. Notice that $\frac{1}{M}V_{a,j}(t)$ is the amount of time that job J_j will have to wait to be satisfied if the algorithm was running SJF on a single machine of speed M. For the arrival condition, accept without proof the fact that at any time it is the case that $V_{a,j}(t) - V_{o,j}(t) \leq Mp_j$ for any job J_j and that J_j has fractional flow time at least $\frac{p_j}{2}$ in the optimal schedule.

Acknowledgments: We thank our many collaborators for discussions that have been indispensable to development of our understanding in this area. We also thank Chandra Chekuri for his comments on this paper.

References

[1] Lachlan L. H. Andrew, Minghong Lin, and Adam Wierman. Optimality, fairness, and robustness in speed scaling designs. In *SIGMETRICS*, pages 37–48, 2010.

[2] Lachlan L. H. Andrew, Adam Wierman, and Ao Tang. Optimal speed scaling under arbitrary power functions. *SIGMETRICS Performance Evaluation Review*, 37(2):39–41, 2009.

[3] Nikhil Bansal, David P. Bunde, Ho-Leung Chan, and Kirk Pruhs. Average rate speed scaling. In *LATIN*, pages 240–251, 2008.

[4] Nikhil Bansal, Ho-Leung Chan, Tak Wah Lam, and Lap-Kei Lee. Scheduling for speed bounded processors. In *ICALP (1)*, pages 409–420, 2008.

[5] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *SODA*, pages 693–701, 2009.

[6] Nikhil Bansal, Ho-Leung Chan, Kirk Pruhs, and Dmitriy Katz. Improved bounds for speed scaling in devices obeying the cube-root rule. In *ICALP (1)*, pages 144–155, 2009.

[7] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1), 2007. Preliminary version in *FOCS* 2004.

[8] Nikhil Bansal, Ravishankar Krishnaswamy, and Viswanath Nagarajan. Better scalable algorithms for broadcast scheduling. In *ICALP*, 2010.

[9] Nikhil Bansal, Kirk Pruhs, and Clifford Stein. Speed scaling for weighted flow time. *SIAM J. Comput.*, 39(4):1294–1308, 2009. Preliminary version in *SODA* 2007.

[10] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. *J. Discrete Algorithms*, 4(3):339–352, 2006. Preliminary version in *APPROX* 2001.

[11] Jivitej S. Chadha, Naveen Garg, Amit Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation. In *STOC*, pages 679–684, 2009.

[12] Ho-Leung Chan, Joseph Wun-Tat Chan, Tak Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence W. H. Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Transactions on Algorithms*, 6(1), 2009. Preliminary version in *SODA* 2007.

[13] Ho-Leung Chan, Wun-Tat Chan, Tak Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence W. H. Wong. Energy efficient online deadline scheduling. In *SODA*, pages 795–804, 2007.

[14] Ho-Leung Chan, Jeff Edmonds, Tak Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. In *STACS*, pages 255–264, 2009.

[15] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *SPAA*, pages 1–10, 2009.

[16] Ho-Leung Chan, Tak Wah Lam, and Rongbin Li. Tradeoff between energy and throughput for online deadline scheduling. In *WAOA*, pages 59–70, 2010.

[17] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Non-clairvoyant speed scaling for weighted flow time. In *ESA (1)*, pages 23–35, 2010.

[18] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Scheduling for weighted flow time and energy with rejection penalty. In *STACS*, 2011.

[19] Jeff Edmonds. Scheduling in the dark. *Theor. Comput. Sci.*, 235(1):109–141, 2000. Preliminary version in *STOC* 1999.

[20] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the ℓ_k-norms of flow time without conservation of work. In *SODA*, 2011.

[21] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *SODA*, pages 685–692, 2009.

[22] Kyle Fox and Benjamin Moseley. Online scheduling on identical machines using srpt. In *SODA*, 2011.

[23] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *SPAA '10: 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2010.

[24] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Scalably scheduling power-heterogeneous processors. In *ICALP (1)*, pages 312–323, 2010.

[25] Xin Han, Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Deadline scheduling and power management for speed bounded processors. *Theor. Comput. Sci.*, 411(40-42):3587–3600, 2010. Preliminary version in *MAPSP* 2009.

[26] Sungjin Im and Benjamin Moseley. An online scalable algorithm for minimizing ℓ_k-norms of weighted flow time on unrelated machines. In *SODA*, 2011.

[27] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000. Preliminary version in *FOCS* 1995.

[28] Tak Wah Lam, Lap-Kei Lee, Hing-Fung Ting, Isaac Kar-Keung To, and Prudence W. H. Wong. Sleep with guilt and work faster to minimize flow plus energy. In *ICALP (1)*, pages 665–676, 2009.

[29] Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Energy efficient deadline scheduling in two processor systems. In *ISAAC*, pages 476–487, 2007.

[30] Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Competitive non-migratory scheduling for flow time and energy. In *SPAA*, pages 256–264, 2008.

[31] Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Speed scaling functions for flow time scheduling based on active job count. In *ESA*, pages 647–659, 2008.

[32] John D. C. Little. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[33] Benjamin Moseley. Scheduling to minimize energy and flow time in broadcast scheduling. *CoRR*, abs/1007.3747, 2010.

[34] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theor. Comput. Sci.*, 130(1):17–47, 1994. Preliminary version in *SODA* 1993.

[35] S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes Gehrke. Online scheduling to minimize average stretch. *SIAM J. Comput.*, 34(2):433–452, 2004. Preliminary version in *FOCS* 1999.

[36] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002. Preliminary version in *STOC* 1997.

[37] Kirk Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Perform. Eval. Rev.*, 34(4):52–58, 2007.

[38] Kirk Pruhs, Jiri Sgall, and Eric Torng. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter Online Scheduling. 2004.

[39] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[40] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.