Competitive Online Scheduling for Server Systems

Kirk Pruhs*
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
kirk@cs.pitt.edu

ABSTRACT

Our goal here is to illustrate the competitive online scheduling research community's approach to online server scheduling problems by enumerating some of the results obtained for problems related to response and slowdown, and by explaining some of the standard analysis techniques.

1. INTRODUCTION

Our goal here is to illustrate the competitive online scheduling research community's approach to online server scheduling problems by enumerating some of the results obtained for problems related to response and slowdown, and by explaining some of the standard analysis techniques.

We consider the setting of a collection of n jobs arriving at a server, or multiservers, over time. Examples of possible servers include databases, name servers, web servers and operating systems. Let r_i be the time that a job J_i is released to the server, and p_i be the work of J_i. On an s speed server, a job with work p_i is completed at the time C_i when it has been be processed for time p_i/s. Online scheduling means that the server's scheduling decisions can not be based on any information about the jobs that will arrive in the future. If jobs can have widely varying work, the scheduler must be able to preempt jobs (and later resume execution from the point of preemption) in order to be able to guarantee any reasonable performance. There are two standard Quality of Service (QoS) measures for a job. The *response* of a job is $F_i = C_i - r_i$, and the *slowdown* of a job is $S_i = F_i/p_i$. For example, a job with slowdown 2 behaves as though it was served by a dedicated speed $\frac{1}{2}$ server. One can then obtain a QoS measure for a schedule by taking the ℓ_p norm, $1 \leq p \leq \infty$, of the QoS measures of the jobs. Mostly commonly the ℓ_1 norm, the average, and the ℓ_∞ norm, the maximum, are considered.

We will explain the standard analysis techniques in the context of the basic scheduling problems of minimizing the average response and the average slowdown on one server. The most obvious worst-case measure of the goodness of an online scheduling algorithm is the competitive ratio. An online scheduling algorithm A is *c-competitive* if:

$$\max_I \frac{A(I)}{Opt(I)} \leq c$$

where $A(I)$ is the QoS measure of the schedule produced by algorithm A on input I, and $Opt(I)$ is the optimal QoS value. So for example, if A is 2-competitive, it means that on all instances $A(I)$ is at most twice optimal. Sometimes one wishes to measure the competitive ratio as a function of some parameter, say the number of jobs. In this case, the maximum is over all instances I with that parameter.

Since the competitive ratio is a worst-case concept, one generally thinks of the competitive ratio as the pay-off of a game played between the online scheduling algorithm A and an adversary. Algorithm A's move at time t is to specify the job that it will run, and the adversary's move at time t is to specify the jobs that arrive at time t. The payoff of the game is then essentially the relative difference between the QoS measure of A's schedule and the QoS measure of the optimal schedule.

The algorithm Shortest Remaining Processing Time (SRPT) always runs the job with the least amount of unfinished work. It is well known that SRPT is 1-competitive (optimal) for average response. Further, SRPT is 2-competitive for average slowdown [21] In each case, the standard proof uses a local competitiveness argument. Local competitiveness is both the most commonly used, and generally the most straight-forward to apply, analysis technique. To understand the local competitiveness analysis technique, let $A(t)$ be the increase of the QoS measure for algorithm A at time t for some understood input I. If the QoS measure was average response, then $A(t)$ is the number of jobs released but unfinished by A at time t. If the QoS measure was average slowdown, then $A(t)$ is the number of jobs released but unfinished by A at time t divided by the aggregate work of these jobs. An algorithm A is *locally c-competitive* if for all inputs I, and all times t, it is the case that $A(t) \leq c \cdot Opt(t)$. An algorithm that is locally c-competitive is then c-competitive since $A(I) = \int_t A(t)dt \leq \int_t c \cdot Opt(t)dt = c \cdot Opt(I)$. To apply local competitiveness to show that SRPT is optimal for average response, ones needs to show that at all times, SRPT always has the least possible number of unfinished jobs. To apply local competitiveness to show that SRPT is 2-competitive for average slowdown, one needs to show that at all times t, SRPT is at most twice optimal with respect to the objective of minimizing the number of unfinished jobs at time t divided by the work of these jobs. Note that when applying local competitiveness, one may potentially have to compare the online algorithm A to a different schedule for each time t.

SRPT is a *clairvoyant* online algorithm in that it requires knowledge of the work the jobs. A clairvoyant scheduling algorithm may be implementable in a web server serving static content, but it would not be implementable in an operating system where the work of the jobs is unknown. The nonclairvoyant algorithm Shortest Elapsed Time First (SETF) runs the job that has been run the least so far. SETF is also called Least Attained Service and Foreground-Background. SETF can be seen as a highly idealized form of the process scheduling algorithm used by Unix. The competitive ratio of SETF can be linear in the number of jobs. To see this, con-

*Supported in part by NSF grants CNS-0325353, CCF-0448196, CCF-0514058 and IIS-0534531.

sider the case of jobs with work λ that arrive at the integer times $0, 1, \ldots, n-1$. For now let $\lambda = 1 + \frac{1}{n}$. From time $n/2$ to time n, SETF has $\Theta(n)$ unfinished jobs with $1/n$ amount of unfinished work, resulting in average response of $\Theta(n)$. But SRPT completes earlier arriving jobs before starting later arriving jobs, and has an average response of $\Theta(1)$.

More generally, the competitive ratio of every deterministic algorithm is $\Omega(n^{1/3})$ [20]. We give the adversarial strategy that establishes this lower bound for an arbitrary deterministic online nonclairvoyant algorithm A. The adversary releases k jobs at time zero, each with work $1/k$ more than the amount that A processes these jobs by time $k-1$ (the fact that this is possible follows from the fact that A is deterministic). It is easy to see that the adversary can have one unfinished job at time $k-1$. Then the adversary brings in a stream of jobs of work $1/k$ for k^2 time units. This results in a competitive ratio for A of $\Omega(n^{1/3})$ since $n = k^3$.

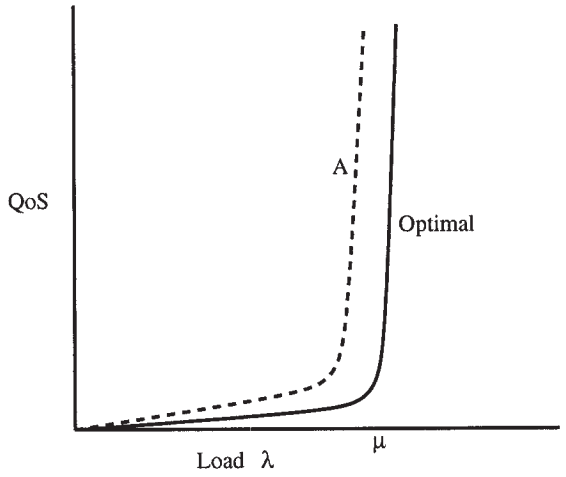

Figure 1: QoS curves of an almost fully scalable online algorithm A and the optimal algorithm for a system with the threshold property.

In spite of this negative worst-case result for SETF, the Unix process scheduling algorithm seems to perform reasonably well over a wide range of inputs. To give one possible explanation for this phenomenon, Kalyanasundaram and Pruhs in [15] introduced what has come to be called *Resource Augmentation Analysis*. The term *resource augmentation analysis* and the notation that we use here were introduced in [22]. To understand the motivation for resource augmentation analysis, note that it is common for systems to posses the following informally defined *threshold property*:

1. The input or input distributions are parameterized by a load λ, and the server is parameterized by a capacity μ. The QoS provided by the server is reasonable if the load λ is at most 90% of the server capacity μ, and the QoS is horrible if λ is more that 110% of μ.

For example, an $M \mid M \mid 1$ queue with the SRPT scheduling discipline has the threshold property [3]. The instances that showed that SETF is $\Omega(n)$-competitive with respect to response have the threshold property. Figure 1 gives an example of the QoS curve for a system that has the threshold property if the optimal scheduling algorithm is used. From Figure 1 it seems that the online scheduling algorithm A performs reasonably well in comparison to the optimal scheduling algorithm. But one can see that the competitive ratio of A is huge by looking at the vertical gap between the curves when the load is near capacity μ. To explain why the curves for A and optimal in Figure 1 are close, we need to also measure the horizontal gap between curves. We would like to say something like A performs at most c times worse than optimal on inputs with s times higher load. Notice that multiplying the load by a factor of s is like slowing the server down by a factor of s. So this is roughly equivalent to saying that A with an s times faster server is at most c times as bad as optimal. Formally, an online algorithm A is an *s-speed c-approximation* algorithm if

$$\max_I \frac{A_s(I)}{Opt_1(I)} \leq c$$

where $A_s(I)$ is the QoS measure of the schedule produced by algorithm A with a speed s server on input I, and $Opt_1(I)$ is the optimal QoS value achievable on a unit speed server. Essentially the best possible resource augmentation result that one can obtain is what is called an *almost fully scalable* algorithm, which is one that is $(1+\epsilon)$-speed $O(1)$-competitive algorithm. The constant in the $O(1)$ term will generally depend on ϵ. If you have a system, that with the optimal offline scheduling algorithm, has the threshold property with threshold μ, and an online algorithm A that is s-speed c-competitive, where c is modest, then the system with A as the scheduling algorithm has the threshold property with threshold μ/s. So in particular, an online scheduling algorithm that is almost fully scalable has essentially the same threshold as the optimal offline algorithm.

In [15] it was shown, using local competitiveness, that SETF was almost fully scalable for average response. More precisely, it was shown that SETF is $(1+\epsilon)$-speed $(1+\frac{1}{\epsilon})$-competitive. This illustrates a phenomenon that is common in many scheduling problems: There is an almost fully scalable algorithm even though there are no $O(1)$-competitive algorithms. The intuition behind this is that if a system's load is near its capacity, then the online scheduler has no time to recover from even small mistakes. Many of the strong worst-case lower bounds for online scheduling problems utilize input instances where the load is essentially the capacity of the system. One example is the instance showing that the competitive ratio of SETF is $\Omega(n)$.

2. RANDOMIZED ALGORITHMS

Another approach to obtaining positive results is to consider randomized scheduling algorithms. Generally randomized online algorithms are compared against an *oblivious adversary* that must specify the input before the online algorithm begins. If the adversary is allowed to change the future input in response to the random events internal to the scheduling algorithm, then generally randomization is not so helpful to the online algorithm.

As an example, consider the problem of minimizing response on one server. One nonclairvoyant algorithm that is discussed in many introductory texts on operating systems is the Multi-Level Feedback algorithm, which can be viewed as mimicking SETF, while keeping the number of preemptions per job to be logarithmic. In MLF, there are a collection $Q_0, Q_1, \ldots$ of queues. There is a target processing time T_i associated with each queue. Typically, $T_i = 2^{i+1}$, but some results require more slowly growing targets, e.g. $T_i = (1+\epsilon)^{i+1}$. Each job J_j gets processed for $T_i - T_{i-1}$ units of time while in queue Q_i before being promoted to the next queue, Q_{i+1}. MLF maintains the invariant that it is always running the job in the front of the lowest nonempty queue.

[16] propose a randomized variation, call RMLF, of MLF that is is identical to MLF except that the target of each job in queue Q_i is 2^{i+1} minus an exponentially distributed independent random variable. [16] shows that RMLF is $\Theta(\log n \log\log n)$ against an adversary that at all times knows the outcome of all of the random events internal to RMLF up until that time. This accounts for the possibility of inputs where future jobs may depend on the past schedule. [8] show that RMLF is $\Theta(\log n)$-competitive against an oblivious adversary. Both of these analyses used local competitiveness.

Every randomized algorithm is $\Omega(\log n)$-competitive for average response [20] against an oblivious adversary. Since this is a cost-minimization problem, we can apply Yao's technique to lower bound the competitive ratio. That is, we need only give an input distribution on which the ratio of the expected response for any deterministic algorithm A divided by the expected optimal response is $\Omega(\log n)$. The input distribution consists of k jobs released at time zero. The work of these jobs is exponentially distributed with mean 1. By the memoryless property of the exponential distribution, expected remaining processing times are independent of A. At time $k - k^{3/4}$, A has $k^{3/4}$ unfinished jobs, but SRPT would only have $k^{3/4}/\log k$ unfinished jobs. The competitive ratio of A can then be forced to $\log n$ by bringing in a stream of short jobs.

The fact that SETF/MLF is almost fully scalable, and RMLF is optimally competitive amongst randomized algorithms, provide support for the adoption of MLF for process scheduling within an operating system.

3. AVERAGE FLOW AND STRETCH ON PARALLEL SERVERS

In the standard model for parallel scheduling, there are m identical servers. No job can be run simultaneously on more than one server. On parallel servers, SRPT is $\Theta(\log n)$-competitive for minimizing total response, and this is known to be optimal within constant factors [19]. This analysis of SRPT used a type of local competitiveness that was widely applied in subsequent papers. The main idea was to bound the additional unfinished work, on jobs with work at most w, that SRPT has in comparison to the adversary. A simpler analysis of SRPT's performance for minimizing response is available in [18]. In [22] it is shown that that SRPT is a $(2 - 1/m)$-speed 1-competitive algorithm for minimizing total response time on parallel servers. SRPT is 14-competitive for minimizing total slowdown on parallel servers [21].

Resource augmentation is also possible on the number of servers. So an s-server c-competitive algorithm A is c-competitive with the optimal schedule with s times fewer servers [22]. However it seems that there many fewer interesting results in the literature that use server augmentation as compared to speed augmentation.

4. IMMEDIATE DISPATCH

An online scheduling algorithm has the *immediate dispatch* property if it assigns a job J_i to a server at time r_i, and all jobs are processed exclusively on the server that they are assigned. Immediate dispatch might be desirable for example if you had a load balancer sitting in front of a server farm, and migrations of jobs between servers was undesirable. In [2] an immediate dispatch algorithm is given that has the same competitive ratio for average response as SRPT, namely $\Theta(\log n)$. In this algorithm, when a job J_i is released, it is assigned to the server that has been assigned the minimum aggregate work of jobs with work about p_i. Note that this assignment rule ignores information such as what is the current load on each server or which jobs have actually been processed or completed at the current time. Each server runs the jobs it is assigned using SRPT. The key observations in the analysis of this algorithm are that: for all w, the aggregate work of jobs with work about w is evenly spread over the various servers, and hence the difference in the total work processed by any time t of jobs with work at most w on any two servers is $O(w)$. It is then established that $A(t) = O(Opt(t) + m \log n)$. The bound then follows by local competitiveness.

5. ℓ_P NORMS OF FLOW AND STRETCH

Often server systems do not implement the best known algorithms for optimizing average Quality of Service (QoS) out of concern of that these algorithms may be insufficiently fair to individual jobs. One standard way to compromise between optimizing for the average and optimizing for the worst case is to optimize the ℓ_p norm, generally for something like $p = 2$ or $p = 3$. For example, the standard way to fit a line to collection of points is to pick the line with minimum least squares, equivalently ℓ_2, distance to the points, and Knuth's TEXtypesetting system uses the ℓ_3 metric to determine line breaks. The ℓ_p, $1 < p < \infty$, metric still considers the average in the sense that it takes into account all values, but because x^p is strictly a convex function of x, the ℓ_p norm more severely penalizes outliers than the standard ℓ_1 norm.

In [5] it is shown that are no $n^{o(1)}$-competitive online clairvoyant scheduling algorithms for any l_p norm, $1 < p < \infty$ of either response or slowdown on one server. This is a bit surprising, at least for response, as there are optimal online algorithms, SRPT and FIFO, for the l_1 and l_∞ norms of response. However, in [5] it is shown that the standard clairvoyant algorithms SJF and SRPT are almost fully scalable for l_p norms of response and slowdown on one server. They showed that the nonclairvoyant algorithms SETF and MLF are almost fully scalable for response objective functions, but not for slowdown objective functions. In contrast, Round Robin (RR), which at all times shares the servers equally amongst all unfinished jobs, is not almost fully scalable even for response objective functions. This is a bit surprising as starvation avoidance is an often cited reason for adopting RR.

The analysis of these algorithms in [5] used local competitiveness. For concreteness, consider the ℓ_2 norm of response. The increase $A_{1+\epsilon}(t)$ of the ℓ_2 norm of response for an algorithm $A_{1+\epsilon}$ at time t is then twice the aggregate ages of the unfinished jobs at time t. That is, if you integrate over time t of the aggregate ages of the unfinished jobs at time t, you give half of the ℓ_2 norm of response. Thus to establish local c-competitiveness for the ℓ_2 norm of response for an algorithm $A_{1+\epsilon}$, it is sufficient to show that at all times the aggregate ages of the unfinished jobs for $A_{1+\epsilon}$ is at most c times the least possible aggregate ages for unfinished jobs at time t using a unit speed processor.

[10] show how to combine immediate dispatching algorithm of [2] with a scheduling policy such as SJF to obtain an almost fully scalable algorithm for l_p norms of response and slowdown on multiservers. The analysis is essentially a local competitive argument similar to the analysis of SJF and SRPT in [5].

6. WEIGHTED FLOW TIME

In the online weighted response problem, each job J_i has an associated positive weight w_i that is revealed to the clairvoyant scheduler at the release timer_i. The objective function is $\sum w_i F_i$. If all $w_i = 1$ then the objective function is total response, and if all $w_i = 1/p_i$ then the objective function is total slowdown. Some systems, such as the Unix operating system, allows different processes to have different priorities. In Unix, users can use the `nice`

command to set the priority of their jobs. Weights provide a way that a system might implement priorities.

For the moment let us focus on one server. For weighted response, [9] show that besides being a sufficient condition, local c-competitiveness is a necessary condition for an algorithm to be c-competitive. The idea is that if an online algorithm was ever worse than locally c-competitive at some time, then the adversary could then bring in a stream of dense short jobs that contribute little to the weighted response if they are processed as they arrive, but will increase the weighted response tremendously if they are delayed at all. The most obvious algorithm is Highest Density First (HDF) which always runs the job of highest density. The density of a job is its weight divided by its work. The competitive ratio of HDF is $\omega(1)$. To see this consider an instance consisting of many low weight and high density jobs, and one high weight and lower density job, all released at time 0. HDF will run the high density jobs first, and thus will not be locally competitive at the time right before it finishes the high weight job, since at this time it might be possible to have only one low weight job left unfinished. This instance demonstrates that the scheduler has to balance between delaying low density jobs, and delaying low weight jobs. Using this intuition, [4] give a $\Theta(\log W)$-competitive algorithm that partitions the jobs based on approximate weight, and then runs SRPT on the jobs in the partition with maximum total weight. Here W is the largest weight. The analysis is a local competitiveness argument that is a variation on the local competitiveness argument for SRPT. For a long time it was universally believed that there existed an $O(1)$-competitive algorithm for weighted response, and finding such an algorithm was viewed as the most important open problem in competitive online scheduling. Recently, Nikhil Bansal and Ho-Leung Chan have shown that that the consensus intuition was not correct by showing that there is no $O(1)$-competitive algorithm for weighted response.

Using a local competitiveness argument, [9] show that HDF is an almost fully scalable algorithm for weighted response on one server. This analysis uses a concept/technique, fractional response, that has proved useful in several other contexts. Let $p_i(t)$ be the remaining unfinished work on job i at time t. Then the increase in the fractional weighted response objective at time t is the sum over of the unfinished jobs J_i of $w_i \frac{p_i(t)}{p_i}$. So for example if a job J_i has $\frac{1}{3}$ of its original work remaining to be done, then J_i only contributes $\frac{1}{3}$ of its weight to the increase of the fractional weighted response objective. HDF is an optimal algorithm for fractional weighted response, and the optimal weighted fractional response is clearly a lower bound for the optimal weighted response. Thus it is sufficient to show that the weighted response for $HDF_{1+\epsilon}$ is competitive with the fractional response for HDF_1. This is established by showing that by the time that HDF_1 is close to finishing a job J_i, which is when J_i might not contribute much to the fractional response, then $HDF_{1+\epsilon}$ has finished J_i. As a human prover, it is often easier to deal with fractional response than integer response since fractional response has a more continuous structure. This is much the same reason it is easier to deal with linear programs than integer linear programs.

[6] give almost fully scalable algorithms for the weighted l_p norms of response. For the parallel server setting, [11] give a lower bound on the competitive ratio of any algorithm of $\Omega(\sqrt{W})$, and [9] show that HDF is $(2+\epsilon)$-speed $O(1)$-competitive.

7. AMORTIZED LOCAL COMPETITIVENESS

An interesting phenomenon arises when we try obtain a resource augmentation analysis for RR for response on one server. [15] give the following lower bound instance for RR, based on an earlier instance in [20]. Let $t_0 = 0$, and $t_1 = 1 + \epsilon$. There are two jobs of work s released at time t_0, and one job is released at each time t_i, $i \geq 1$, with work $z(i)$ that is exactly the same work that RR has left on each of the previous jobs. To guarantee that the adversary can finish the job released at time t_i by time t_{i+1}, $i \geq 1$, let $t_{i+1} = t_i + z(i)$. Then the total response for the adversary is $\Theta(\sum_{i=1}^{n} 1/i^s)$, and the total response for RR is $\Theta(\sum_{i=1}^{n} 1/i^{s-1})$. This instance shows that RR is not 2-speed $O(1)$-competitive, and RR is not locally competitive for any constant s. But, at least for this instance, RR is $(2+\epsilon)$-speed $O(1)$-competitive. In a landmark paper, Edmonds [12] proved RR is in fact $(2+\epsilon)$-speed $O(1)$-competitive. Since a local competitiveness argument can not work here, Edmonds had to develop a new technique: amortized local competitiveness. Let A be an arbitrary online scheduling algorithm. Let $A(t)$ be the rate of increase of the objective at time t. The online algorithm A is *amortized locally c-competitive with potential function* $\Phi(t)$ if the following two conditions hold:

Boundary: Φ is initially 0, and finally nonnegative.

Job Arrival: Φ does not increase when a new job arrives.

Completion: Φ does not increase when either the online algorithm or the adversary complete a job.

Running: For all times t when no job arrives or is completed,

$$A(t) - cOpt(t) + \frac{d\Phi(t)}{dt} \leq 0 \qquad (1)$$

Observe that when $\Phi(t)$ is identically zero, we have ordinary local competitiveness. To see that amortized local c-competitiveness implies global competitiveness, let $t_1, t_2, \ldots$ be the events that either a job is released, the online algorithm A completes a job, or the adversary completes a job. Let $\Delta(\Phi(t_i))$ denote the change in potential in response to event t_i. Integrating equation 1 over time, we get that

$$A(I) + \sum_{t_i} \Delta(\Phi(t_i)) \leq cOpt(I)$$

By the job arrival condition, and the completion condition, we can conclude that $A(I) + \Phi(\infty) - \Phi(0) \leq cOpt(I)$, and finally, by the boundary condition, we can conclude that $A(I) \leq cOpt(I)$.

Intuitively on can think of the potential function Φ as a bank. When $A(t) < cOpt(t)$, A is doing better than it needs to, and can save money in the bank. When $A(t) > cOpt(t)$, A is doing worse than it can afford to, and thus must withdraw money from the bank to pay for this. The conditions above just imply that A can not cheat the bank, for example, by not paying back money that it borrowed.

8. ARBITRARY SPEED-UP CURVES

An immediate question that one has to ask when formalizing a scheduling problem on parallel servers is whether a single job can simultaneously run on multiservers. In some settings this may not be possible; in other settings this may be possible but the speed-up that one obtains may vary. Thus one can get myriad different scheduling problems on parallel servers depending on what one assumes. A very general model is to assume that each job has a speed-up function that specifies how much the job is sped up when assigned to multiservers. More formally, a *speed-up function* $\Gamma(s)$ measures the rate at which work is finished on the job if s processing resources (say s servers) are given to the job.

A job is *parallelizable* if $\Gamma(s) = s$. Parallelizable work has the property that if you devote twice as many servers to the work, it completes at twice the rate. At the other extreme, a job is *constant* if $\Gamma(s) = c$ for all $s \geq 0$ and some constant $c > 0$. Devoting additional processing resources to constant jobs does not result in any faster processing of these jobs. In fact constant jobs complete at the same rate even if they are not run. The normal multiserver setting can be modeled by the speed-up function $\Gamma(s) = s$ for $s \leq 1$ and $\Gamma(s) = 1$ for $s > 1$. That is, a job is parallelizable on one server, but assigning the job to multiservers does not help.

In any real application, speed-up functions will be sublinear and non-decreasing. A speed-up function is sublinear if doubling the number of servers at most doubles the rate at which work is completed on the job. A speed-up function is non-decreasing if increasing the number of servers does not decrease the rate at which work is completed on the job. One can also generalize this so that jobs are made of phases, each with their own speed-up function. Assume that a nonclairvoyant scheduling algorithm does not know the speed-up function of any job.

In a remarkable analysis, Edmonds showed that RR is $(2+\epsilon)$-speed $O(1)$-competitive for jobs with phases that have speed-up functions that are sublinear and non-decreasing [12]. A corollary of this general result is that RR is $(2+\epsilon)$-speed $O(1)$-competitive for response on one server. This results extends, with slightly weaker bounds, to the case where RR is given extra servers instead of faster servers.

Edmonds first transforms each possible input into a canonical input that is streamlined. An input is *streamlined* if: (1) every phase is either parallelizable or constant, and (2) the adversary is able to execute each job at its maximum possible speed. This implies that at any one time, the adversary has only one parallel job phase to which it is allocating all of its resources. The idea of this transformation is that if RR is devoting more resources to some work than the adversary, it is to the adversary's advantage to make this work be constant work that completes at the rate that the adversary was originally processing that work. In contrast, if the adversary is devoting more resources to a job than is RR, and the adversary has no other unfinished jobs, then it is to the adversary's advantage to make this work to be parallelizable. As a consequence of this transformation, you get that the adversary is never behind RR on any job. The fact that the input is streamlined means that without loss of generality one can assume that RR has one server of speed $s = 2 + \epsilon$ and Opt has one server of speed 1.

We now turn to the potential function Φ used by Edmonds. The potential $\Phi(t) = F(t) + Q(t)$ where $Q(t)$ is total sequential work finished by RR by time t minus the total sequential work finished by the adversary by time t. To define $F(t)$ requires some preliminary definitions. For $u \geq t$, define $m_u(t)$ ($\ell_u(t)$) to be number of fully parallelizable (sequential) phases executing under RR at time u, for which RR at time u has still not processed as much work as the adversary processed at time t. Let $n_u(t) = m_u(t) + \ell_u(t)$. Then $F(t) = \int_t^\infty f_u(m_u(t), \ell_u(t))du$, where $f_u(m,\ell) = \frac{s}{s-2}\frac{(m-\ell)(m+\ell)}{n_u}$. As the definition of the potential function suggests, Edmond's analysis of RR is quite complicated.

9. MULTICAST PULL SCHEDULING

In a multicast/broadcast server system, when the server sends a requested page/item, all outstanding client requests to this page are satisfied by this multicast. The system may use broadcast because the underlying physical network provides broadcast as the basic form of communication (e.g. if the network is wireless or the whole system is on a LAN). We restrict our attention to the case that the objective function is total response. Multicast pull scheduling is a generalization of weighted response. If one restricts the instances in multicast pull scheduling such that for each page, all requests for that page arrive at the same time, then the multicast pull scheduling problem and the weighted response scheduling problem are identical.

Assume for now that all pages have the same work/size, say as would be the case for a name server. The most obvious algorithm is Most Requests First (MRF), which broadcasts the page with the most outstanding requests. At first, one might even be tempted to think that MRF is optimal. However, [17] show that MRF is not even $O(1)$-speed $O(1)$-competitive. To see this consider the instance where at time 0 there is single request to each of n pages, and at each time t, $0 \leq t \leq n$, there 2 requests to s other special pages. At each time MRF_s broadcasts the s special pages. Thus at time $n+s$, MRF_s will not be locally $O(1)$-competitive since it has $n - s$ pages with outstanding requests, but it is possible to have finished all the pages. By bringing in a stream of requests to new pages, one obtains an instance where the competitive ratio of MRF_s is $\Omega(n)$. This lower bound instance shows that the online scheduler has to be concerned with not only the popularity of the requests, but also with how to best aggregate jobs. Further, [13, 17] show that no $O(1)$-competitive algorithm exists for this problem.

The lower bound instance for MRF actually contains the key insight that relates multicast pull scheduling to scheduling with speed-up curves, and thus suggests a possible algorithm. After the online algorithm has finished a page that was requested by a single client, the adversary can again direct another client to request that page. The online algorithm must service this second request as well. In contrast, the optimal schedule knows not to initially give any resources to the first request because the broadcast for the second request simultaneously services the first. Thus, even though the online algorithm devotes a lot of resources to the first request and the optimal algorithm devotes no resources to the first request, it completes under both at about the same time. In this regard, the work associated with the first request can be thought of as "constant". This suggests that the real difficulty of broadcast scheduling is that the adversary can force some of the work to have a constant speed-up curve.

Formalizing this intuition, [13] give a method to convert any nonclairvoyant unicast scheduling algorithm A to a multicast scheduling algorithm B. A unicast algorithm can only answer one request at a time, as is the case on a standard web server. [13] shows that if A works well when jobs can have parallel and constant phases, then B works well if it is given twice the resources. The basic idea is that B simulates A, creating a separate job for each request, and then the amount of time that B broadcasts a page is equal to the amount of time that A runs the corresponding jobs. More formally, if A is an s-speed c-competitive unicast algorithm, then its counterpart, algorithm B, is a $2s$-speed c-competitive multicast algorithm. In the reduction, each request in the multicast pull problem is replaced by a job whose work is constant up until the time that either the adversary starts working on the job or the online algorithm finishes the job. After that time, the work of the replacement job is parallel. The amount of parallel work is such that A will complete a request exactly when B completes the corresponding job. Using the RR for algorithm A, one obtains an algorithm, called BEQUI in [13], that broadcasts each page at a rate proportional to the number of outstanding requests. Using Edmonds' analysis of RR for jobs with speed-up functions, one gets that BEQUI is $(4+\epsilon)$-speed $O(1 + 1/\epsilon)$-competitive. In fact, all the results in [13] hold if the pages have arbitrary sizes under the assumptions that the clients have to receive a page in order.

The most popular multicast pull scheduling algorithm for unit work pages in the computer systems literature is Longest Wait First (LWF). LWF always services the page for which the aggregate waiting times of the outstanding requests for that page is maximized. In the natural setting where for each page, the request arrival times have a Poisson distribution, LWF broadcasts each page with frequency roughly proportional to the square root of the page's arrival rate, which is essentially optimal. [14] show that LWF is 6-speed $O(1)$-competitive, but is not almost fully scalable. It is not too difficult to see that there is no possibility of proving such a result using local competitiveness. The authors of [13] were not able to obtain a potential function that would allow them to establish this result via amortized local competitiveness. This illustrates one potential difficulty a human prover might encounter when searching for an amortized local competitiveness argument. One has discover a potential function to establish the fundamental inequality 1 for all configurations. But there can be configurations, in which the online algorithm is not doing badly, yet the natural potential functions aren't sufficiently refined to establish inequality 1. The rather complicated analysis given in [14] sums up the total cost of LWF, and sums up the total cost to the adversary, and then compares like terms.

10. SPEED SCALING

In addition to the traditional goal of efficiently managing time and space, many computers now need to efficiently manage power usage. For example, Intel's SpeedStep and AMD's PowerNOW technologies allow the Windows XP operating system to dynamically change the speed of the processor to prolong battery life. In this setting, the operating system must not only have a *job selection policy* to determine which job to run, but also a *speed scaling* policy to determine the speed at which the job will be run. In current CMOS based processors, the speed satisfies the well known cube-root-rule, that the speed is approximately the cube root of the power. Energy is power integrated over time. The operating system is faced with a dual objective optimization problem as it both wants to conserve energy, and optimize some Quality of Service (QoS) measure of the resulting schedule.

If there is an upper bound on energy used, then there is no $O(1)$-competitive online speed scaling policy for total response. To understand intuitively why this is the case, consider the situation when the first job arrives. The scheduler has to allocate a constant fraction of the total energy to this job; otherwise, the scheduler would not be $O(1)$-competitive in the case that no more jobs arrive. However, if many more jobs arrive in the future, then the scheduler has wasted a constant fraction of its energy on only one job. By iterating this process, one obtains a bound of $\omega(1)$ on the competitive ratio with respect to total response.

Albers and Fujiwara [1] proposed combining the dual objectives of energy and response into the single of objective of energy used plus total response. Optimizing a linear combination of energy and total response has the following natural interpretation. Suppose that the user specifies how much improvement in response, call this amount ρ, is necessary to justify spending one unit of energy. For example, the user might specify to the Windows XP operating system that he is willing to spend 1 erg of energy from the battery for a decrease of 3 micro-seconds in response. Then the optimal schedule, from this user's perspective, is the schedule that optimizes $\rho = 3$ times the energy used plus the total response. By changing the units of either energy or time, one may assume without loss of generality that $\rho = 1$.

Local competitiveness is generally not achievable in speed scaling problems because the adversary may spend essentially all of its energy in some small period of time, making it impossible for any online algorithm to be locally competitive at that time. Thus amortized local analysis is the tool of choice.

Let us consider the objective of fractional weighted response plus energy. The increase of the objective for an algorithm A at time t is $w_A(t) + p_A(t) = w_A(t) + s_A(t)^\alpha$, $w_A(t)$ is the fractional weight of the unfinished jobs at time t for algorithm A, $p_A(t)$ and $s_A(t)$ are the power and speed of algorithm A at time t, and $p_A(t) = s_A(t)^\alpha$ is the speed to power function. Thus the fundamental local competitiveness equation 1 in this case is equivalent to:

$$w_A(t) + s_A(t)^\alpha - c(w_{Opt}(t) + s_{Opt}(t)^\alpha) + \frac{d\Phi(t)}{dt} \leq 0 \quad (2)$$

For reasons explained in [1], that we will not go into here, the natural algorithm to consider is the algorithm A that uses HDF for job selection, and always runs at a power $p_A(t) = w_A(t)$. Then equation 2 reduces to

$$2w_a(t) - c(w_o(t) + s_o(t)^\alpha) + \frac{d\Phi(t)}{dt} \leq 0\,, \quad (3)$$

Here one can think of $\Phi(t)$ as a measure of the energy in a bank/battery at time t. Then $\frac{d\Phi(t)}{dt}$ is then a measure of power representing the rate that energy is flowing into or out of the bank/battery.

First consider the simpler case where all jobs have unit work and unit weight, [7] shows that the algorithm A is 2-competitive for the objective function of fractional response plus energy. Let $\beta = (\alpha - 1)/\alpha$. If you have w jobs with equal release times, then Opt is proportional to $w^{\beta+1}$. Thus we know that, in order for Φ to have enough energy stored to pay for the future, assuming that no more jobs arrive, that it must be the case that $\Phi(t) \geq w_A(t)^{\beta+1} - w_{Opt}(t)^{\beta+1}$. Unfortunately setting the potential function equal to the right-hand side of this inequality does not satisfy equation 3 when $w_A(t) \gg w_{Opt}(t)$ and a new job arrives. The problem is that the incremental cost to A in this situation is much larger than the incremental cost for the adversary, and the potential function does not decrease enough to pay for this. Thus [7] use the related potential function

$$\Phi(t) = \frac{2\alpha}{(\beta+1)} \left(\max(0, w_a(t) - w_o(t))\right)^{\beta+1}$$

Note that this potential function decreases more quickly than the first candidate potential function in the situation discussed above.

[7] shows that by rounding the power used by A up to the next integer one obtains a 4-competitive algorithm for the objective function (integral) response plus energy.

Let us turn our attention back to the general case of jobs with arbitrary work and arbitrary weight, [7] show that algorithm A is $O(1)$-competitive with respect to the objective of fractional weighted response plus energy. When the cube-root rule holds, the competitive ratio is approximately 2.52. To define the potential function used in [7], let $w_A(h)$ be a function of t denoting the total fractional weight of the jobs unfinished by A that have inverse density of at least h. The inverse density of a job is the work divided by the weight. The potential function used in [7] is then

$$\Phi(t) = \frac{2}{\beta+1} \int_{h=0}^{\infty} \left(w_a(h)^\beta (w_a(h) - (\beta+1)w_o(h))\right) dh \quad (4)$$

For the objective of energy plus (integer) weighted response, [7] shows that an algorithm that tries to mimic algorithm A is a bit less than 8-competitive when the cube-root rule holds.

11. CONCLUSIONS

Recall that our goal here is to illustrate the competitive online scheduling research community's approach to online server scheduling problems by enumerating some of the results obtained for problems related to response and slowdown, and by explaining some of the standard analysis techniques. Our goal was not to present an exhaustive survey. Necessarily our choices for results to cover is idiosyncratic. We offer our apologies to the authors of the many fine papers not discussed here. The closest thing to an exhaustive server of competitive online scheduling is probably the survey article [23].

I have been asked to compare competitive analysis to stochastic analysis. Given my immature knowledge of stochastic scheduling, I am somewhat reluctant, but here is my best shot. Let us start with the disadvantages of competitive analysis. Since competitive analysis is a worst-case concept, the results are generally overly pessimistic for normal inputs. Also competitive analysis only bounds the performance relative to the optimal algorithm, it does not give any absolute measure of performance. So it might recommend a scheduling algorithm to run on your server farm, but it wouldn't tell you how many servers to buy to handle a certain number of users. Let us now turn to the advantages of competitive analysis, including resource augmentation analysis. It seems that for server systems, the key property of a good online scheduling algorithm is that it should scale as well as the optimal scheduling algorithm with the load. Algorithms that are $O(1)$-competitive, or almost fully scalable, have this scaling property. This seems to be the reason that competitive analysis generally recommends the "right" algorithms, for example, SRPT for average slowdown on a web server, SETF for average response on an operating system, and LWF for average response on a multicast name server. Further, one can reasonably obtain a competitive analysis for a wide variety of scheduling applications, without requiring probabilistic assumptions about the input distribution. It is often not clear what the "right" probabilistic assumption is, and if one assumes some general probability distribution, the resulting analysis is often intractable. Consider for example a multicast-pull web server. To apply stochastic analysis would require assuming some joint probability distribution over file size and popularity, that is, you need to know how file size correlates with popularity. It is probably not so clear a priori what the "right" assumption is for a web server, or even that there is a "right" assumption. For example, this joint distribution may depend on whether the client-side caching policy is something like LRU, which doesn't discriminate based on file size, or is something like Greedy-Dual-Size, which is more likely to evict large files. If one assumes a general joint distribution, then it is probably not so tractable to analyze algorithms using such a general assumption (although in fairness to stochastic scheduling, part of this intractability derives from the stochastic scheduling researchers' desire for exact, instead of approximate, results).

12. REFERENCES

[1] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. In *Symposium on Theoretical Aspects of Computer Science*, pages 621–633, 2006.

[2] N. Avrahami and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proc. 15th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 11–18. ACM, 2003.

[3] N. Bansal. On the average sojourn time under m—m—1—srpt. *Opererations Research Letters*, 33(2):195–200, 2005.

[4] N. Bansal and K. Dhamdhere. Minimizing weighted flow time. In *Proc. 14th Symp. on Discrete Algorithms (SODA)*, pages 508–516. ACM/SIAM, 2003.

[5] N. Bansal and K. Pruhs. Server scheduling in the L_p norm: A rising tide lifts all boats. In *Proc. 35th Symp. Theory of Computing (STOC)*, pages 242–250. ACM, 2003.

[6] N. Bansal and K. Pruhs. Server scheduling in the weighted l_p norm. Manuscript, 2003.

[7] N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. In *ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[8] L. Becchetti and S. Leonardi. Non-clairvoyant scheduling to minimize the average flow time on single and parallel machines. In *Proc. 33rd Symp. Theory of Computing (STOC)*, pages 94–103. ACM, 2001. To appear in JACM.

[9] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs. Online weighted flow time and deadline scheduling. In *RANDOM-APPROX*, volume 2129 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2001.

[10] C. Chekuri, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize l_p norms of flow and stretch. Manuscript, 2003.

[11] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for weighted flow time. In *Proc. 33rd Symp. Theory of Computing (STOC)*, pages 84–93. ACM, 2001.

[12] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235:109–141, 2000.

[13] J. Edmonds and K. Pruhs. Multicast pull scheduling: when fairness is fine. *Algorithmica*, 36:315–330, 2003.

[14] J. Edmonds and K. Pruhs. A maiden analysis of longest wait first. In *Proc. 15th Symp. on Discrete Algorithms (SODA)*. ACM/SIAM, 2004.

[15] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:214–221, 2000.

[16] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly. *Journal of the ACM*, 50:551–567, 2003.

[17] B. Kalyanasundaram, K. R. Pruhs, and M. Velauthapillai. Scheduling broadcasts in wireless networks. *Journal of Scheduling*, 4:339–354, 2001.

[18] S. Leonardi. A simpler proof of preemptive flow-time approximation. In *Approximation and On-line Algorithms*, Lecture Notes in Computer Science. Springer, 2003.

[19] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. 29th Symp. Theory of Computing (STOC)*, pages 110–119. ACM, 1997.

[20] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130:17–47, 1994.

[21] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize avarage strech. In *Proc. 40th Symp. Foundations of Computer Science (FOCS)*, pages 433–443. IEEE, 1999.

[22] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, pages 163–200, 2002.

[23] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In *Handbook on Scheduling*. CRC Press, 2004.