Average Rate Speed Scaling

Nikhil Bansal* David P. Bunde† Ho-Leung Chan‡ Kirk Pruhs§

May 12, 2008

Abstract

Speed scaling is a power management technique that involves dynamically changing the speed of a processor. This gives rise to dual-objective scheduling problems, where the operating system both wants to conserve energy and optimize some Quality of Service (QoS) measure of the resulting schedule. Yao, Demers, and Shenker [4] considered the problem where the QoS constraint is deadline feasibility and the objective is to minimize the energy used. They proposed an online speed scaling algorithm Average Rate (AVR) that runs each job at a constant speed between its release and its deadline. They showed that the competitive ratio of AVR is at most $(2\alpha)^\alpha/2$ if a processor running at speed s uses power s^α. We show the competitive ratio of AVR is at least $((2-\delta)\alpha)^\alpha/2$, where δ is a function of α that approaches zero as α approaches infinity. This shows that the competitive analysis of AVR by Yao, Demers, and Shenker is essentially tight, at least for large α. We also give an alternative proof that the competitive ratio of AVR is at most $(2\alpha)^\alpha/2$ using a potential function argument. We believe that this analysis is significantly simpler and more elementary than the original analysis of AVR in [4].

1 Introduction

Current processors produced by Intel and AMD allow the speed of the processor to be changed dynamically. Intel's SpeedStep and AMD's PowerNOW technologies allow the Windows XP operating system to dynamically change the speed of such a processor to conserve energy. 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 consumption 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.

The first theoretical worst-case study of speed scaling algorithms was in the seminal paper [4] by Yao, Demers, and Shenker. Their QoS objective was deadline feasibility and the objective was to minimize the energy used. More precisely, each job i has a release time r_i when it arrives in the system, a work requirement w_i, and a deadline d_i by which the job must be finished. If job i runs at constant speed s, then

*IBM T. J. Watson Research Center, nikhil@us.ibm.com

†Computer Science Department, Knox College, dbunde@knox.edu. Supported in part by Howard Hughes Medical Institute grant 52005130.

‡Computer Science Department, University of Pittsburgh, hlchan@cs.pitt.edu.

§Computer Science Department, University of Pittsburgh, kirk@cs.pitt.edu. Supported in part by NSF grants CNS-0325353, CCF-0514058 and IIS-0534531.

it completes in w_i/s units of time. In this setting, an optimal job selection policy is Earliest Deadline First (EDF). They assumed a speed to power function $P(s) = s^\alpha$, where $\alpha > 1$ is some constant. If the cube-root rule holds, then $\alpha = 3$. Yao, Demers, and Shenker [4] showed that the optimal energy feasible schedule is found by a simple greedy algorithm that we call YDS.

Yao, Demers, and Shenker [4] also proposed an online speed scaling algorithm, Average Rate (AVR). Conceptually, AVR runs each job i at speed $w_i/(d_i - r_i)$ throughout interval $[r_i, d_i]$, independent of other jobs. This spreads the work of each job as evenly over time as possible. By the convexity of the speed to power function, this even spreading is energy optimal if the instance consists of only one job. The speed of the processor at any time t is then just the sum of the speeds of the jobs active at that time, that is $\sum_{i:t\in[r_i,d_i]} \frac{w_i}{d_i - r_i}$. AVR is an appealing speed scaling algorithm because in some sense it is perfectly fair to all jobs, and each job runs as if it were the only job in the instance.

Yao, Demers, and Shenker [4] showed that the competitive ratio, with respect to energy, of AVR is at least α^α. They also showed that the competitive ratio of AVR, with respect to energy, is at most $(2\alpha)^\alpha/2$. We now outline this upper bound competitive analysis of AVR. A job is defined to be of *type A* if the optimal schedule is always ahead of AVR on this job. A job is defined to be of *type B* if AVR is always ahead of the optimal schedule on this job. A schedule is *bitonic* if every job is of type A or type B. [4] observes that there is a worst-case instance that is bitonic, and that the competitive ratio of AVR is at most $2^{\alpha-1}$ times the competitive ratio of AVR on instances of jobs of just one type (A or B). [4] then considers instances consisting only of type-A jobs. [4] then introduces an auxiliary objective function that is related to, but is not exactly, the energy used. In a somewhat involved reduction, [4] shows that with respect to this auxiliary objective, there is a worst-case instance where the optimal schedule is non-preemptive, each job starts when it is released, and the spans of the jobs are nested (where the *span* of job i is the interval $[r_i, d_i]$). When $\alpha = 2$, [4] then shows that for such instances, optimizing the auxiliary objective function can be represented in terms of the eigenvalues of a particular tree-induced matrix, and shows how to bound the largest eigenvalue for such tree-induced matrices. [4] states that this argument can be readily generalized to an arbitrary α, and using Hölder's inequality, give a bound on the ℓ_p norm of a certain tree-induced matrix that would replace the eigenvalue argument used in the $\alpha = 2$ case.

So the natural question left open is, "What is the exact competitive ratio of AVR?" Based on simulation results, [4] conjectured that the competitive ratio of AVR is exactly α^α. That is, that the lower bound in [4] is correct, and intuitively, that AVR can not simultaneously be losing badly on both type-A and type-B jobs. In the case that the cube-root rule holds, $\alpha^\alpha = 3^3 = 27$ is the best known competitive ratio for any online algorithm. If the conjecture from [4] was true, this would be evidence in favor of adopting the AVR speed scaling policy. Not only would AVR have the best known competitive ratio in the case that the cube-root rule holds, but AVR is appealingly fair to all jobs.

Unfortunately, in section 4, we show that the upper bound on the competitive ratio from [4] is essentially tight, at least for larger α. More precisely, we show that AVR has competitive ratio at least $((2-\delta)\alpha)^\alpha/2$, where δ is a function of α that approaches zero as α approaches infinity. In the case obeying the cube-root rule, we get a lower bound of approximately 48 on the competitive ratio of AVR.

In section 5, we give an alternative proof that the competitive ratio of AVR is at most $(2\alpha)^\alpha/2$. Our analysis uses a potential function argument. We believe that this analysis is significantly simpler and more elementary than the original analysis of AVR in [4]. Our competitive analysis of AVR branches off from the analysis in [4] outlined above after the observation that the competitive ratio of AVR is at most $2^{\alpha-1}$ times the competitive ratio of AVR on jobs of just one type. We give a potential function argument that AVR is α^α-competitive on type-A jobs. We include a complete analysis of AVR in this paper, including the elements of the analysis from [4] that we use. In principle, verifying this analysis requires only basic

algebra, except that some basic calculus is used to verify the positivity/negativity of certain polynomials over particular ranges.

2 Other Related Results

There are now enough speed scaling papers in the literature that it is not practical to survey all such papers here. We limit ourselves to those papers most related to the results presented here.

Yao, Demers, and Shenker [4] also proposed another online speed scaling algorithm, Optimal Available (OA). The algorithm OA runs at the optimal speed (which can be computed using the YDS algorithm) assuming the current state and that no more jobs will be released in the future. [4] showed that the competitive ratio of OA is at least α^α. Using a potential function analysis, Bansal, Kimbrel, and Pruhs [2] showed that OA is actually α^α-competitive.

Bansal, Kimbrel, and Pruhs [2] also introduced an online speed scaling algorithm that we call BKP. Intuitively, BKP tries to mimic the offline YDS schedule in some way. Formally, at time t BKP runs at speed $e\ v(t)$ where $v(t) = \max_{t'>t} \frac{w(t,et-(e-1)t',t')}{e(t'-t)}$ and $w(t, t_1, t_2)$ is the amount of work that has release time at least t_1, deadline at most t_2, and that has already arrived by time t. [2] showed that BKP is simultaneously $O(1)$-competitive for total energy, maximum temperature (assuming cooling obeys Newton's law), maximum power, and maximum speed. Specifically, [2] showed that the competitive ratio of BKP with respect to energy is at most $2(\alpha/(\alpha-1))^\alpha e^\alpha$.

Albers, Müller, and Schmelzer [1] consider the problem of finding energy-efficient deadline-feasible schedules on multiprocessors. [1] showed that the offline problem is NP-hard, and gave $O(1)$-approximation algorithms. [1] also gave online algorithms that are $O(1)$-competitive when job deadlines occur in the same order as their release times.

3 Formal Problem Statement

A problem instance consists of n jobs. Job i has a release time r_i, a deadline $d_i > r_i$, and work $w_i > 0$. In the online version of the problem, the scheduler learns about a job only at its release time; at this time, the scheduler also learns the exact work requirement and the deadline of the job. We assume that time is continuous. A schedule specifies for each time a job to be run and a speed at which to run the job. The speed is the amount of work performed on the job per unit time. A job with work w run at a constant speed s thus takes $\frac{w}{s}$ time to complete. More generally, the work done on a job during a time period is the integral over that time period of the speed at which the job is run. A schedule is *feasible* if for each job i, work at least w_i is done on job i during $[r_i, d_i]$. Note that the times at which work is performed on job i do not have to be contiguous. If a job is run at speed s, then the power is $P(s) = s^\alpha$ for some constant $\alpha > 1$.

The energy used during a time period is the integral of the power over that time period. Our objective is to minimize the total energy used by the schedule.

If A is a scheduling algorithm, then $A(I)$ denotes the schedule output by A on input I. A schedule is R-competitive for a particular objective function if the value of that objective function on the schedule is at most R times the value of the objective function on an optimal schedule. An online scheduling algorithm A is R-competitive, or has competitive ratio R, if $A(I)$ is R-competitive for all instances I.

For a schedule T, let $s_{T,j}(t)$ denote the speed job j runs at time t in the schedule T, and let $s_T(t) = \sum_j s_{T,j}(t)$ denote the speed of the processor at time t in schedule T. If U is a subcollection of jobs, let $s_{T,U}(t)$ denote the sum of the speeds of the jobs in U at time t in the schedule T. We will also substitute