Algorithms for Dynamic Speed Scaling *

Susanne Albers[1]

1 Humboldt-Universität zu Berlin
Department of Computer Science
Unter den Linden 6, 10099 Berlin, Germany
albers@informatik.hu-berlin.de
http://www2.informatik.hu-berlin.de/~albers/

Abstract

Many modern microprocessors allow the speed/frequency to be set dynamically. The general goal is to execute a sequence of jobs on a variable-speed processor so as to minimize energy consumption. This paper surveys algorithmic results on dynamic speed scaling. We address settings where (1) jobs have strict deadlines and (2) job flow times are to be minimized.

2010 Mathematics Subject Classification 68Q25, 68W27, 68W40, 90B35

Keywords and phrases Competitive analysis, energy-efficiency, flow time, job deadline, offline algorithm, online algorithm, response time, scheduling, variable-speed processor.

Digital Object Identifier 10.4230/LIPIcs.STACS.2011.1

1 Introduction

Energy has become a scarce and expensive resource. There is a growing awareness in society that energy conservation and an efficient energy use are important issues. Power dissipation is also critical in computer systems. Electricity costs impose a substantial strain on the budget of data and computing centers. Google representatives report that if power consumption continues to grow, power costs can easily overtake hardware costs by a large margin [11]. In this context engineers are interested in low power rather than speed [30]. Moreover, energy-efficiency is a concern in portable and battery-operated devices that have proliferated in recent years. An effective energy use can considerably prolong the lifetime of a battery and hence the availability of a system.

A relatively new and very promising technique to save energy in computer systems is dynamic speed scaling. Chip manufacturers such as Intel, AMD and IBM produce microprocessors that can run at variable speed. Examples are the Intel SpeedStep and the AMD PowerNow. High speeds result in high performance but also high energy consumption. Lower speeds save energy but performance degrades. In dynamic speed scaling the processor speed is adjusted based on demand and performance constraints. The goal is to minimize energy consumption, while still providing a desired quality of service. The past years have witnessed considerable research interest in dynamic speed scaling. In this paper we survey results that have been developed in the algorithms community.

The well-known cube-root rule for CMOS devices states that the speed s of a device is proportional to the cube-root of the power or, equivalently, that power is proportional to s^3. The algorithms literature considers a generalization of this rule. If a processor runs at speed s, then the required power is $P(s) = s^\alpha$, where $\alpha > 1$ is a constant. Most algorithms papers

* Work supported by a Gottfried Wilhelm Leibniz Award of the German Research Foundation.

consider this power function $P(s)$; some even work with more generalized convex functions. Obviously, energy consumption is power integrated over time.

Dynamic speed scaling leads to many challenging scheduling problems. The general goal is to execute a sequence of jobs on a variable-speed processor so as to optimize energy consumption and, possibly, a second objective. However, problems in speed scaling are more complex than those in standard scheduling: At any time a scheduler has to decide not only which job to execute but also which speed to use.

There has recently been considerable research interest in the design and analysis of dynamic speed scaling algorithms. The algorithms literature so far focuses mostly on two scenarios. In a first scenario jobs have strict deadlines and a scheduler has to construct feasible schedules minimizing energy consumption. We review important results for this setting in Section 2. In a second scenario jobs have no deadlines but their response times or flow times are to be minimized, measuring the responsiveness of a system. Here one has to combine energy minimization and flow time minimization. We present results for this scenario in Section 3.

For the various scenarios, two problem settings are of interest. In the offline setting all jobs to be processed are known in advance. Here one is interested in complexity results and fast polynomial time algorithms for computing optimal or nearly optimal schedules. In the online setting jobs arrive over time and an algorithm, at any time, has to make scheduling decisions without knowledge of any future jobs. Online strategies are evaluated using competitive analysis [33]. An online algorithm ALG is called c-competitive if for every input, i.e. for any job sequence, the objective function value (typically the energy consumption) of ALG is within c times the value of an optimal solution for that input.

2 Scheduling with deadlines

In a seminal paper, initiating the algorithmic study of speed scaling, Yao, Demers and Shenker [34] investigated a scheduling problem with strict job deadlines. It is by far the most extensively studied speed scaling problem.

Consider n jobs $J_1, \ldots J_n$ that have to be processed on a variable-speed processor. Each job J_i is specified by a release time r_i, a deadline d_i and a processing volume w_i. The release time and the deadline specify the time interval $[r_i, d_i]$ during which the job must be executed. The job may not be started before r_i and must be finished until d_i. The processing volume w_i is the amount of work that must be completed to finish the job. Intuitively w_i can be viewed as the total number of CPU cycles required by the job. The processing time of the job depends on the processor speed. If J_i is executed at speed s, then it takes w_i/s time units to finish the task. Preemption of jobs is allowed, i.e. the processing of a job may be stopped and resumed later. The goal is to construct a feasible schedule minimizing the total energy consumption

Yao, Demers and Shenker [34] make two simplifying assumptions. (1) There is no upper bound on the allowed processor speed. Hence a feasible schedule always exists. (2) The processor has a continuous spectrum of speeds. In the following we will present algorithms for this enhanced model. Then we will discuss how to relax the assumptions.

2.1 Basic algorithms

Yao et al. [34] developed elegant online and offline algorithms. We first present the offline strategy, which knows all the jobs along with their characteristics in advance. The algorithm is known as *YDS*, referring to the initials of the authors. Algorithm *YDS* computes a minimum energy schedule for a given job set in a series of rounds. In each round the algorithm identifies an interval of maximum density and computes a corresponding partial schedule for that interval. The *density* Δ_I of a time interval $I = [t, t']$ is the total processing volume to be completed in I divided by the length of I. More formally, let S_I be the set of jobs J_i that must be processed in I, i.e. that satisfy $[r_i, d_i] \subseteq I$. Then

$$\Delta_I = \frac{1}{|I|} \sum_{J_i \in S_I} w_i.$$

Intuitively, Δ_I is the minimum average speed necessary to complete all jobs that must be scheduled in I.

In each round, *YDS* determines the interval I of maximum density. In I the algorithm schedules the jobs of S_I at speed Δ_I according to *Earliest Deadline First (EDF)*. The *EDF* policy always processes the job having the earliest deadline among the available unfinished jobs. Then *YDS* removes the set S_I as well as the time interval I from the problem instance. More specifically, for any unscheduled job J_i with $d_i \in I$, the new deadline is set to $d_i := t$. For any unscheduled J_i with $r_i \in I$, the new release time is $r_i := t'$. Time interval I is discarded. A summary of *YDS* in pseudo-code is given below.

Algorithm YDS: Initially $\mathcal{J} := \{J_1, \ldots, J_n\}$. While $\mathcal{J} \neq \emptyset$, execute the following two steps. (1) Determine the interval I of maximum density. In I process the jobs of S_I at speed Δ_I according to *EDF*. (2) Set $\mathcal{J} := \mathcal{J} \setminus S_I$. Remove I from the time horizon and update the release times and deadlines of unscheduled jobs accordingly.

The algorithm computes optimal schedules.

▶ **Theorem 2.1.** [34] *For any job instance, YDS computes an optimal schedule minimizing the total energy consumption.*

Obviously, the running time of *YDS* is polynomial. When identifying intervals of maximum density, the algorithm only has to consider intervals whose boundaries are equal to the release times and deadlines of the jobs. Hence, a straightforward implementation of the algorithm has a running time of $O(n^3)$. Li et al. [29] showed that the time can be reduced to $O(n^2 \log n)$. Further improvements are possible if the job execution intervals form a tree structure [27].

In the online version of the problem, the jobs $J_1, \ldots, J_n$ arrive over time. A job J_i becomes known only at its arrival time r_i. At that time the deadline d_i and the processing volume w_i are also revealed. Recall that an online algorithm *ALG* is c-competitive if, for any job sequence, the total energy consumption of *ALG* is at most c times that of an optimal offline algorithm *OPT*.

Yao et al. [34] devised two online algorithms, called *Average Rate* and *Optimal Available*. For any incoming job J_i, *Average Rate* considers the *density* $\delta_i = w_i/(d_i - r_i)$, which is the minimum average speed necessary to complete the job in time if no other jobs were present. At any time t the speed $s(t)$ is set to the accumulated density of jobs active at time t. A job J_i is *active at time* t if $t \in [r_i, d_i]$. Available jobs are scheduled according to the *EDF* policy.

Algorithm Average Rate: At any time t the processor uses a speed of $s(t) = \sum_{J_i : t \in [r_i, d_i]} \delta_i$. Available unfinished jobs are scheduled using *EDF*.

Yao et al. [34] proved an upper bound on the competitiveness.

▶ **Theorem 2.2.** [34] *The competitive ratio of Average Rate is at most* $2^{\alpha-1}\alpha^\alpha$, *for any* $\alpha \geq 2$.

Bansal et al. [3] demonstrated that the analysis is essentially tight by giving a nearly matching lower bound.

▶ **Theorem 2.3.** [3] *The competitive ratio of Average Rate is at least* $((2-\delta)\alpha)^\alpha/2$*, where* δ *is a function of* α *that approaches zero as* α *tends to infinity.*

The second strategy *Optimal Available* is computationally more expensive than *Average Rate*. It always computes an optimal schedule for the currently available work load. This can be done using *YDS*.

Algorithm Optimal Available: Whenever a new job arrives, compute an optimal schedule for the currently available unfinished jobs.

Bansal, Kimbrel and Pruhs [7] analyzed the above algorithm and proved the following result.

▶ **Theorem 2.4.** [7] *The competitive ratio of Optimal Available is exactly* α^α.

The above theorem implies that in terms of competitiveness, *Optimal Available* is better than *Average Rate*. Bansal et al. [7] also developed a new online algorithm, called *BKP* according to the initials of the authors, that approximates the optimal speeds of *YDS* by considering interval densities. For times t, t_1 and t_2 with $t_1 < t \le t_2$, let $w(t, t_1, t_2)$ be the total processing volume of jobs that are active at time t, have a release time of at least t_1 and a deadline of at most t_2.

Algorithm BKP: At any time t use a speed of

$$s(t) = \max_{t'>t} \frac{w(t, et-(e-1)t', t')}{t'-t}.$$

Available unfinished jobs are processed using *EDF*.

▶ **Theorem 2.5.** [7] *Algorithm BKP achieves a competitive ratio of* $2(\frac{\alpha}{\alpha-1})^\alpha e^\alpha$.

For large values of α, the competitiveness of *BKP* is better than that of *Optimal Available*. Bansal et al. [6] gave an algorithm that achieves further improved bounds for small values of α, i.e. $\alpha = 2$ and $\alpha = 3$.

All the above online algorithms attain constant competitive ratios that depend on α but no other problem parameter. The dependence on α is exponential. For small values of α, which occur in practice, the competitive ratios are reasonably small. Moreover, results by Bansal et al. [6, 7] imply that the exponential dependence on α is inherent to the problem.

▶ **Theorem 2.6.** [6] *Any deterministic online algorithm has a competitiveness of at least* $e^{\alpha-1}/\alpha$.

Even randomized online algorithms have a competitive ratio of $\Omega((4/3)^\alpha)$, see [7]. An interesting open problem is to determine the best competitiveness that can be achieved by online algorithms.

2.2 Speed-bounded processors

The algorithms presented in the last section are designed for processors having available a continuous, unbounded spectrum of speeds. However, in practice a processor is equipped with only a finite set of discrete speed levels $s_1 < s_2 < \ldots < s_d$. The offline algorithm *YDS* can be modified easily to handle feasible job instances, i.e. inputs for which feasible schedules exist using the restricted set of speeds. Feasibility can be checked easily by always using the maximum speed s_d and scheduling available jobs according to the *EDF* policy. Given a feasible job instance the modification of *YDS* is as follows. We first construct the schedule according to *YDS*. For each identified interval I of maximum density we approximate the desired speed Δ_I by the two adjacent speed levels s_k and s_{k+1}, such that $s_k < \Delta_I < s_{k+1}$. Speed s_{k+1} is used first for some δ time units and s_k is used for the last $|I| - \delta$ time units in I, where δ is chosen such that the total work completed in I is equal to the original amount of $|I|\Delta_I$. An algorithm with an improved running time of $O(dn \log n)$ was presented by Li and Yao [28].

If the given job instance is not feasible, it is impossible to complete all the jobs. Here the goal is to design algorithms that achieve good *throughput*, which is the total processing volume of jobs finished by their deadline, and at the same time optimize energy consumption. Papers [4, 15] present algorithms that even work online. At any time the strategies maintain a pool of jobs they intend to complete. Newly arriving jobs may be admitted to this pool. If the pool contains too large a processing volume, jobs are expelled such that the throughput is not diminished significantly. The algorithm with the best competitiveness currently known is due to Bansal et al. [4]. The algorithm, called *Slow-D*, is 4-competitive in terms of throughput and constant competitive with respect to energy consumption. We describe the strategy.

Slow-D assumes that the processor has a continuous speed spectrum that is upper bounded by a maximum speed $s_{\max}$. The algorithm always keeps track of the speeds that *Optimal Available* would use for the workload currently available. At any time t *Slow-D* uses the speed that *Optimal Available* would set at time t provided that this speed does not exceed $s_{\max}$; otherwise *Slow-D* uses $s_{\max}$. The algorithm also considers scheduling times that are critical in terms of speed. For any t, down-time(t) is the latest time $t' \geq t$ in the future schedule such that the speed of *Optimal Available* is at least $s_{\max}$. If no such time exists, down-time(t) is set to the most recent time when $s_{\max}$ was used or to 0 if this has never been the case. Using this definition, jobs are labeled as *urgent* or *slack*. These labels may change over time. A job J_i is called t-urgent if $d_i \leq$ down-time(t); otherwise it is called t-slack. Additionally, *Slow-D* maintains two queues Q_{work} and Q_{wait} of jobs it intends to process. The status of Q_{work} defines *urgent periods*. An urgent period starts at the release time r_i of a job J_i if Q_{work} contained no urgent job right before r_i and J_i is an urgent job admitted to Q_{work} at time r_i. An urgent period ends at time t if Q_{work} contains no more t-urgent jobs. *Slow-D* works as follows.

Algorithm Slow-D: JOB ARRIVAL: A job J_i arriving at time r_i is admitted to Q_{work} if it is r_i-slack or if J_i and all the remaining work of r_i-urgent jobs in Q_{work} can be completed using $s_{\max}$. Otherwise J_i is appended to Q_{wait}.
JOB INTERRUPT: Whenever a job J_i in Q_{wait} reaches its last starting time $t = d_i - w_i/s_{\max}$, it raises an interrupt. At this time the algorithm is in an urgent period. Let J_k be the last job transfered from Q_{wait} to Q_{work} in the current period. If no such job exists, let J_k be a dummy job of processing volume zero transfered just before the current period started. Let W be the total original work of jobs ever admitted to Q_{work} that have become urgent after J_k was transfered to Q_{work}. If $w_i > 2(w_k + W)$, then remove all t-urgent jobs from Q_{work} and admit J_i; otherwise discard J_i.
JOB COMPLETION: Whenever a job is completed, it is removed from Q_{work}.

Bansal et al. [4] analyzed the above algorithm and proved the following result.

▶ **Theorem 2.7.** [4] *Slow-D is 4-competitive with respect to throughput and* $(\alpha^\alpha + \alpha^2 4^\alpha)$*-competitive with respect to energy.*

Interestingly, the competitiveness of 4 is best possible, even if energy is ignored, see [12].

2.3 Problem extensions

We consider further extensions of the classical deadline-based scheduling setting.

Sleep states: Irani et al. [22] investigate an extended scenario where a variable-speed processor may be transitioned into a sleep state. In the sleep state, the energy consumption is 0 while in the active state even at speed 0 some non-negative amount of energy is consumed. Hence [22] combines speed scaling with power-down mechanisms. In the standard setting without sleep state, algorithms tend to use low speed levels subject to release time and deadline constraints. In contrast, in the setting with sleep state it can be beneficial to speed up a job so as to generate idle times in which the processor can be transitioned to the sleep mode. Irani et al. [22] develop online and offline algorithms for this extended setting. For the online setting an algorithm with an improved competitiveness was presented by Han et al. [21]; their strategy achieves a competitiveness of $\alpha^\alpha + 2$. Baptiste [9], Baptiste et al. [10] and Demaine et al. [18] also study scheduling problems where a processor may be set asleep, albeit in a setting without speed scaling.

Parallel processors: The results presented so far address single-processor architectures. However, energy consumption is also a major concern in multi-processor environments. Consider a setting with m identical parallel processors. As usual the processing of a jobs may be preempted at any time. We distinguish two problem variants depending on whether or not *job migration* is allowed. If job migration is feasible, then whenever a job is preempted it may be moved to another processor. In some applications job migration can be an expensive or undesirable operation, and thus might be infeasible. In any case the goal is to minimize the total energy consumption on all the processors. Bingham and Greenstreet [13] showed that if job migration is allowed, the offline problem is polynomially solvable. However the corresponding algorithm relies on linear programming and, as the authors mention, the complexity of the algorithm might be too high for most practical applications.

Albers et al. [2] assume that job migration is not allowed. They show that the offline problem is NP-hard, even for unit-size jobs. Albers et al. [2] then develop polynomial time offline algorithms that achieve constant factor approximations, i.e. for any input the consumed energy is within a constant factor of the true optimum. They also devise online algorithms attaining constant competitive ratios. Greiner et al. [19] gave a strategy that converts a c-approximation algorithm for a single processor into a randomized cB_α-approximation algorithm for multiple processors. Here B_α is the α-th Bell number. A corresponding statement holds for online algorithms.

Lam et al. [24] study deadline-based scheduling on two speed-bounded processors. They present a strategy that is constant competitive in terms of throughput maximization and energy minimization.

3 Minimizing flow times

A classical objective in scheduling is the minimization of response times. A user releasing a task to a system would like to receive feedback, say the result of a computation, as quickly as possible. User satisfaction often depends on how fast a device reacts. Unfortunately, response time minimization and energy minimization are contradicting objectives. To achieve fast response times a system must usually use high processor speeds, which lead to high energy consumption. On the other hand, to save energy low speeds should be used, which result in high response times. Hence one has to find ways to integrate both objectives.

Consider n jobs $J_1, \ldots, J_n$ that have to be scheduled on a variable-speed processor. Each job J_i is specified by a release time r_i and a processing volume w_i. When a job arrives, its processing volume is known. Preemption of jobs is allowed. In the scheduling literature, response time is referred to as *flow time*. The flow time f_i of a job J_i is the length of the time interval between release time and completion time of the job. We seek schedules minimizing the total flow time $\sum_{i=1}^{n} f_i$.

3.1 Energy plus flow

Albers and Fujiwara [1] proposed the following approach to integrate energy and flow time minimization. They consider a combined objective function that simply adds the two costs. Let E denote the energy consumption of a schedule. We wish to minimize $g = E + \sum_{i=1}^{n} f_i$. By multiplying either the energy or the flow time by a scalar, we can also consider a weighted combination of the two costs, expressing the relative value of the two terms in the total cost. Albers and Fujiwara [1] concentrate on the setting where all jobs have the same processing volume. By scaling, one can assume that all jobs have unit-size. They show that optimal offline schedules can be constructed in polynomial time using a dynamic programming approach.

Most of [1] is concerned with the online setting where jobs arrive over time. Albers and Fujiwara present a simple online strategy that processes jobs in batches and achieves a constant competitive ratio. Batched processing allows one to make scheduling decisions, which are computationally expensive, only every once in a while. This is certainly an advantage in low-power computing environments. Nonetheless, Albers and Fujiwara conjectured that the following algorithm achieves a better performance with respect to the minimization of g: At any time, if there are ℓ active jobs, use speed $\sqrt[\alpha]{\ell}$. A job is active if it has been released but is still unfinished. Intuitively, this is a reasonable strategy because, in each time unit, the incurred energy of $(\sqrt[\alpha]{\ell})^\alpha = \ell$ is equal to the additional flow time accumulated by the ℓ jobs during that time unit. Hence, both energy and flow time contribute the same value to the objective function. The algorithm and variants thereof have been the subject of extensive analyses [4, 5, 8, 26], not only for unit-size jobs but also for arbitrary size jobs. Moreover, unweighted and weighted flow times have been considered.

The currently best result is due to Bansal et al. [5]. They modify the above algorithm slightly by using a speed of $\sqrt[\alpha]{\ell+1}$ whenever ℓ jobs are active. Inspired by a paper of Lam et al. [26] they apply the *Shortest Remaining Processing Time (SRPT)* policy to the available jobs. More precisely, at any time among the active jobs, the one with the least remaining work is scheduled.

Algorithm Job Count: At any time if there are $\ell \geq 1$ active jobs, use speed $\sqrt[\alpha]{\ell+1}$. If no job is available, use speed 0. Always schedule the job with the least remaining unfinished work.

▶ **Theorem 3.1.** [5] *Job Count is 3-competitive for arbitrary size jobs.*

The above result even holds for a very general class of convex power functions. Bansal et al. [5, 8] study a generalized setting where each job J_i has a weight β_i associated with it and in objective function g the total flow time is replaced by the weighted flow time $\sum_{i=1}^{n} \beta_i f_i$. The proposed algorithms rely on the *Highest Density First (HDF)* policy, i.e. at any time among the available unfinished jobs the one with the highest *density* is processed. The density of a job J_i is the ratio β_i/w_i of its weight to its work. Bansal et al. [8] introduced a relaxed objective function consisting of energy plus the fractional weighted flow time of the jobs. In the fractional weighted flow time measure, at any time a job contributes its weight times the percentage of unfinished work to the objective. In their first paper Bansal et al. [8] gave a constant competitive online algorithm for minimizing energy plus fractional weighted flow. An algorithm achieving a small constant competitive ratio of 2 was shown in the second paper [5]. This algorithm always applies *HDF* for job selection and sets the processor power equal to the total fractional weight of the unfinished jobs. A constant competitive algorithm for the original objective function of energy plus (integral) weighted flow was shown in [8].

Bansal et al. [4] and Lam et al. [26] propose algorithms for the setting that there is an upper bound on the maximum processor speed. All the results mentioned so far assume that when a job arrives, its processing volume is known. Articles [16, 26] investigate the harder case that this information is not available.

3.2 Problem extensions and modifications

Sleep states: Lam et al. [23] study an extended setting where a variable-speed processor is equipped with one or several sleep states. The processing time of incoming jobs may or may not be known. The authors devise online algorithms achieving constant competitive ratios for minimizing energy plus flow.

Parallel processors: Lam et al. [25] and Gupta et al. [20] investigate scenarios with m parallel processors. Both articles assume that job migration is not allowed. For identical processors Lam et al. [25] present a constant competitive online algorithm for minimizing energy plus flow. The performance ratio even holds against migratory offline schedules. The corresponding algorithm classifies jobs according to their processing volumes and was originally proposed by Albers et al. [2]. Gupta et al. [20] consider heterogeneous processors and study the effect of resource augmentation: If an offline algorithm can run a processor at speed s and power $P(s)$, then an online algorithm is able to run the processor at speed $(1+\epsilon)s$ and power $P(s)$, for any given $\epsilon > 0$. Gupta et al. present an online algorithm that is *scalable* for minimizing energy plus weighted flow. Here scalable means that the online cost is upper bounded by $O(f(\epsilon))$ time the optimum cost, where f is a polynomial function of small degree. Again the result holds for a very general class of power functions. If the power functions of all the processors are of the form $P_i(s) = s^{\alpha_i}$, $1 \leq i \leq m$, Gupta et al. show a $O(\alpha^2)$-competitive algorithm, where $\alpha = \max_i \alpha_i$. Hence resource augmentation is not needed. Chan et al. [17] investigate parallel processor scheduling assuming that jobs have varying degrees of parallelizability and their processing times are initially unknown.

Limited energy: Pruhs et al. [31] consider another approach to integrate energy and flow time minimization. More specifically they study a problem where a fixed energy volume E is given and the goal is to minimize the total flow time of the jobs. Pruhs et al. [31] assume that all jobs have unit-size. They consider the offline scenario and show that optimal schedules can be computed in polynomial time. Bunde [14] extends the result to parallel processor environments and gives an arbitrarily-good approximation for scheduling unit-size jobs. He also shows that the optimal flow time value cannot be exactly computed on a machine supporting exact real arithmetic, including the extraction of roots. We remark that in the framework with a limited energy volume it is hard to construct good online algorithms. If future jobs are unknown, it is unclear how much energy to invest for the currently available tasks.

4 Conclusions

In this paper we have surveyed algorithmic results on dynamic speed scaling, focusing on settings with strict job deadlines and on the minimization of job flow times. Various papers have also addressed other scenarios. A basic objective function in scheduling is makespan minimization, i.e. the minimization of the point in time when the entire schedule ends. Bunde [9] develops algorithms for single and multi-processor environments. Pruhs et al. [32] consider tasks having precedence constraints defined between them. They devise algorithms for parallel processors given a fixed energy volume. In summary, practical applications motivate the investigation of many further settings and we expect that dynamic speed scaling continues to be an active area of research.

References

1 S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3, 2007.

2 S. Albers, F. Müller and S. Schmelzer. Speed scaling on parallel processors. *Proc. 19th ACM Symposium on Parallelism in Algorithms and Architectures*, 289–298, 2007.

3 N. Bansal, D.P. Bunde, H.-L. Chan and K. Pruhs. Average rate speed scaling. *Proc. 8th Latin American Symposium on Theoretical Informatics,* Springer LNCS 4957, 240–251, 2008.

4 N. Bansal, H.-L. Chan, T.-W. Lam and K.-L. Lee. Scheduling for speed bounded processors. *Proc. 35th International Colloquium on Automata, Languages and Programming*, Springer LNCS 5125, 409–420, 2008.

5 N. Bansal, H.-L. Chan and K. Pruhs. Speed scaling with an arbitrary power function. *Proc. 20th ACM-SIAM Symposium on Discrete Algorithm*, 693–701, 2009.

6 N. Bansal, H.-L. Chan, K. Pruhs and D. Katz. Improved bounds for speed scaling in devices obeying the cube-root rule. *Proc. 36th International Colloqium on Automata, Languages and Programming*, Springer LNCS 5555, 144–155, 2009.

7 N. Bansal, T. Kimbrel and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM,* 54, 2007.

8 N. Bansal, K. Pruhs and C. Stein. Speed scaling for weighted flow time. *SIAM Journal on Computing*, 39:1294–1308, 2009.

9 P. Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, 364–367, 2006.

10 P. Baptiste, M. Chrobak and C. Dürr. Polynomial time algorithms for minimum energy scheduling. *Proc. 15th Annual European Symposium on Algorithms*, Springer LNCS 4698, 136–150, 2007.

11 L.A. Barroso. The price of performance. *ACM Queue*, 3, 2005.

12 S.K. Baruah, G. Koren, B. Mishra, A. Raghunathan, L.E. Rosier and D. Shasha. On-line scheduling in the presence of overload. *Proc. 32nd Annual Symposium on Foundations of Computer Science*, 100–110, 1991.

13 B.D. Bingham and M.R. Greenstreet. Energy optimal scheduling on multiprocessors with migration. *Proc. IEEE International Symposium on Parallel and Distributed Processing with Applications*, 143–152, 2008.

14 D.P. Bunde. Power-aware scheduling for makespan and flow. *Journal of Scheduling*, 12:489–500, 2009.

15 H.-L. Chan, W.-T. Chan, T.-W. Lam, K.-L. Lee, K.-S. Mak and P.W.H. Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Transactions on Algorithms*, 6, 2009.

16 H.-L. Chan, J. Edmonds, T.-W. Lam, L.-K. Lee, A. Marchetti-Spaccamela and K. Pruhs. Nonclairvoyant speed scaling for flow and energy. *Proc. 26th International Symposium on Theoretical Aspects of Computer Science*, 255–264, 2009.

17 H.-L. Chan, J. Edmonds and K. Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Proc. 21st Annual ACM Symposium on Parallel Algorithms and Architectures*, 1–10, 2009.

18 E.D. Demaine, M. Ghodsi, M.T. Hajiaghayi, A.S. Sayedi-Roshkhar and M. Zadimoghaddam. Scheduling to minimize gaps and power consumption. *Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, 46–54, 2007.

19 G. Greiner, T. Nonner and A. Souza. The bell is ringing in speed-scaled multiprocessor scheduling. *Proc. 21st Annual ACM Symposium on Parallel Algorithms and Architectures*, 11–18, 2009.

20 A. Gupta, R. Krishnaswamy and K. Pruhs. Scalably scheduling power-heterogeneous processors. *Proc. 37th International Colloqium on Automata, Languages and Programming*, Springer LNCS 6198, 312–323, 2010.

21 X. Han, T.W, Lam, L.-K. Lee, I.K.-K. To and P.W.H. Wong. Deadline scheduling and power management for speed bounded processors. *Theoretical Computer Science*, 411:3587-3600, 2010.

22 S. Irani, S.K. Shukla and R. Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3, 2007.

23 T.W. Lam, L.-K. Lee, H.-F. Ting, I.K.-K. To and P.W.H. Wong. Sleep with guilt and work faster to minimize flow plus energy. *Proc. 36th International Colloqium on Automata, Languages and Programming*, Springer LNCS 5555, 665–676, 2009.

24 T.-W. Lam, L.-K. Lee, I.K.-K. To and P.W.H. Wong. Energy efficient deadline scheduling in two processor systems. *Proc. 18th International Symposium on Algorithms and Computation*, Springer LNCS 4835, 476–487, 2007.

25 T.-W. Lam, L.-K. Lee, I.K.-K. To and P.W.H. Wong. Competitive non-migratory scheduling for flow time and energy. *Proc. 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, 256–264, 2008.

26 T.-W. Lam, L.-K. Lee, I.K.-K. To and P.W.H. Wong. Speed scaling functions for flow time scheduling based on active job count. *Proc. 16th Annual European Symposium on Algorithms*, Springer LNCS 5193, 647–659, 2008.

27 M. Li, B.J. Liu and F.F. Yao. Min-energy voltage allocation for tree-structured tasks. *Journal on Combintorial Optimization*, 11:305–319, 2006.

28 M. Li and F.F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. *SIAM Journal on Computing*, 35:658–671, 2005.

29 M. Li, A.C. Yao and F.F. Yao. Discrete and continuous min-energy schedules for variable voltage processors. *Proc. National Academy of Sciences USA*, 103, 3983–3987, 2006.

30 J. Markoff and S. Lohr. Intel's huge bet turns iffy. *The New York Times*, Septemer 29, 2002.

31 K. Pruhs, P. Uthaisombut and G.J. Woeginger. Getting the best response for your erg. *ACM Transactions on Algorithms*, 4, 2008.

32 K. Pruhs, R. van Stee and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43:67–80, 2008.

33 D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

34 F.F. Yao, A.J. Demers and S. Shenker. A scheduling model for reduced CPU energy. *Proc. 36th IEEE Symposium on Foundations of Computer Science*, 374–382, 1995.