Algorithms for Power Savings

Sandy Irani* Sandeep Shukla† Rajesh Gupta‡

Abstract

This paper examines two different mechanisms for saving power in battery-operated embedded systems. The first is that the system can be placed in a sleep state if it is idle. However, a fixed amount of energy is required to bring the system back into an active state in which it can resume work. The second way in which power savings can be achieved is by varying the speed at which jobs are run. We utilize a power consumption curve $P(s)$ which indicates the power consumption level given a particular speed. We assume that $P(s)$ is convex, non-decreasing and non-negative for $s \geq 0$. The problem is to schedule arriving jobs in a way that minimizes total energy use and so that each job is completed after its release time and before its deadline. We assume that all jobs can be preempted and resumed at no cost. Although each problem has been considered separately, this is the first theoretical analysis of systems that can use both mechanisms. We give an offline algorithm that is within a factor of two of the optimal algorithm. We also give an online algorithm with a constant competitive ratio.

1 Introduction

As battery-operated embedded systems proliferate, energy efficiency is becoming an increasingly critical consideration in system design. This paper examines strategies that seek to minimize power usage in such systems via two different mechanisms:

1. **Sleep State:** if a system or device is idle it can be put into a low-power *sleep* state. While the device consumes less power in this state, a fixed amount of energy is required to transition the system back to an *on* state in which tasks can be performed. An offline algorithm which knows ahead of time the length of the idle period can determine whether the idle period is long enough so that the savings in energy from being in the *sleep* state outweighs the cost to transition back to the *on* state. An online algorithm does not know the length of the idle period in advance and must determine a threshold T such that if the idle period lasts for at least time T, it will transition to the *sleep* state after that time. The embedded systems literature refers to the problem of deciding when to transition to a low-power *sleep* state as *Dynamic Power Management*.

2. **Dynamic Speed Scaling:** some systems can perform tasks at different speeds. The power usage of such a system is typically described by a convex function $P(s)$, where $P(s)$ is the power-usage level of the system when it is running at speed s. In many settings, the amount of work required by

*Information and Computer Science, UC Irvine, 92697, irani@ics.uci.edu. Supported in part by NSF grant CCR-0105498 and by ONR Award N00014-00-1-0617.

†Electrical and Computer Engineering Virginia Tech, Blacksburg, VA 24061, shukla@vt.edu. Supported in part by NSF grant CCR-0098335, SRC, and DARPA/ITO supported PADS project under the PAC/C program.

‡Department of Computer Science and Engineering, AP&M 3110, UC San Diego, La Jolla, CA 92093, gupta@cs.ucsd.edu. Supported in part by NSF grant CCR-0098335, SRC, and DARPA/ITO supported PADS project under the PAC/C program.

jobs can be estimated when they arrive into the system. The goal is to complete all jobs between their release time and their deadline in a way that minimizes the total energy consumption. Since the power function is convex, it is more energy efficient to slow down the execution of jobs as much as possible while still respecting their timing constraints. An offline algorithm knows about all jobs in advance while an online algorithm only learns about a job upon its release. This problem often goes under the name *Dynamic Voltage Scaling* or *Dynamic Frequency Scaling* in the embedded systems literature.

We design algorithms for the Dynamic Speed Scaling problem in which the system has the additional feature of a *sleep* state. We call this problem Dynamic Speed Scaling with Sleep State (DSS-S). DSS-NS (no sleep) will denote the Dynamic Speed Scaling without a *sleep* state. Combining these two problems (Dynamic Speed Scaling and Dynamic Power Management) introduces challenges which do not appear in either of the original problems. In the first problem, the lengths of the idle intervals are given as part of the input whereas in our problem they are created by the scheduler which decides when and how fast to perform the tasks. In the DSS-NS problem, it is always in the best interest of the scheduler to run jobs as slowly as possible within the constraints of the release times and deadlines due to the convexity of the power function. By contrast in DSS-S, it may be beneficial to speed up a task in order to create an idle period in which the system can sleep.

There are numerous examples of systems that can be run at multiple speeds, have a *sleep* state and receive jobs with deadlines. For example, the Rockwell WINS node is a mobile sensing/computing node that has onboard environmental sensors. It gathers the data and then sends it over ad hoc wireless links through an onboard radio to other nodes. The onboard computation has two parts (a) a full fledged processor that does application as well as many of the networking protocols; (b) a microcontroller that enables the sensors. Data can be transmitted at different speeds and each speed has a different power usage rate. The system also has a *sleep* state in which the power usage level is greatly reduced. [16, 14]

2 Previous Work

The problem of when to transition a device to a *sleep* state when it is idle is a continuous version of the *Ski Rental* problem [5]. It is well known that the optimal competitive ratio that can be achieved by any deterministic online algorithm for this problem is 2. Karlin *et al.* examine the problem when the length of the idle period is generated by a known probability distribution [8]. Irani *et al.* examine the generalization in which there are multiple *sleep* states, each with a different power usage rate and start-up cost [6]. There has also been experimental work that investigates how to use trace data to estimate a probability distribution that can be used to guide probabilistic algorithms [9, 6]. The embedded systems literature refers to the problem of deciding when to transition to a low-power *sleep* state as *Dynamic Power Management*. Benini, Bogliolo and De Micheli give an excellent review of this work [2].

The Dynamic Speed Scaling problem without the *sleep* state has been examined by Yao, Demers and Shenker (although not under that name) [17]. They give an optimal offline algorithm for the problem. Their algorithm plays an important role in our algorithms for DSS-S, so we will discuss it in depth in a subsequent section. Yao *et al.* also define a simple online algorithm called Average Rate (AVR)

and prove that the competitive ratio for AVR is between d^d and $2^d d^d$, when the power usage as a function of speed is a degree-d polynomial. Recently, Bansal, Kimbrel and Pruhs have shown that another natural online algorithm called Opitmal Available obtains a competitive ratio of d^d which is tight for that algorithm [1]. They also introduce a new algorithm which obtains a competitive ratio of $2(d/(d-1))^d e^d$.

Dynamic Speed Scaling is also a well studied problem in the embedded systems literature. (See [11] and references therein). The problem often goes by the name *Dynamic Voltage Scaling* or *Dynamic Frequency Scaling*. We adopt the more generic term *Dynamic Speed Scaling* to emphasize the fact that the algorithm selects the speed of the system to minimize power usage. Simunic examines the problem of combining Dynamic Speed Scaling and Dynamic Power Management for an embedded system called SmartBadge [15]. Another related paper examines task scheduling (although not with multiple speeds) so as to create idle periods for putting a device into a sleep state [10]. This problem captures some of the features of the problem we address here.

There are a number of issues in the real-world problem of power management that are not incorporated into the model we use in this paper. The first of these has to do with the latency incurred in transitioning from one state to another. Some previous work on Dynamic Power Management does incorporate the latency involved in transitioning from the *on* to the *sleep* state and vice versa [2]. Ramanathan *et al.* perform an experimental study of the latency/power tradeoff in Dynamic Power Management [13]. In [6], algorithms which are designed using a model which does not incorporate this latency perform very well empirically even when this additional latency is taken into account.

The model we use here also omits the transition time from one speed to another as well as the time to preempt and resume jobs. In addition, we assume here that the power function is continuous and that there is no upper bound on the speed of the system. In reality, there are a finite number of speeds at which the system can run and the algorithm must select one of these values. Some work in the systems literature address models in which the system can not change instantaneously or continuously between speeds [3, 4]. Naturally, this makes the problem much harder to solve. As a result, much of the work on Dynamic Speed Scaling makes all of the assumptions we make here. It remains to determine experimentally whether these assumptions are in fact reasonable.

3 Our Results

We prove two results in this paper. These results hold for convex power functions $P(s)$. The convexity of $P(s)$ is a standard assumption in this area and corresponds well to analytical models for $P(s)$ [12]. We give an offline algorithm for the DSS-S problem that produces a schedule whose total energy consumption for any set of jobs is within a factor of two of optimal. We still do not know whether the offline problem is NP-hard.

We also present an online algorithm for DSS-S that makes use of an online algorithm for DSS-NS. We define the following properties of an algorithm for DSS-NS: let $\mathcal{J}$ be a set of input tasks to an algorithm for DSS-NS. Let $\mathcal{J}'$ be a subset of $\mathcal{J}$. An online algorithm for DSS-NS is said to be *additive* if for every

t:

$$s_{A,\mathcal{J}'}(t) \leq s_{A,\mathcal{J}}.$$

This will be important because our algorithm schedules some jobs according to an online algorithm A for DSS-NS and some jobs according to a difference scheme. The set of jobs given to A is a subset of all of the jobs in the instance. We will need to argue that at every point in time the speed incurred by the jobs given to A is at most the speed that A would have incurred on the total set of jobs.

An online algorithm is said to be *monotonic* if the only points in time at which it increases its speed are when a new job is released. In other words, an online algorithm always has a plan as to how fast it will schedule the set of jobs that have been released but have not yet been completed. This is exactly the speed function it will employ if no additional jobs arrive. If the online algorithm is monotonic, this plan will not involve increasing the speed. The monotonic property will be important later in the proof because we need to lower bound the speed of the device while it is on. This is to ensure that as long as the algorithm is paying the price to keep the device on, a certain amount of work is being accomplished. We only invoke the online algorithm A for DSS-NS when the current set of uncompleted jobs can not be finished by their deadlines at a certain speed s. In this case, we want to be sure that A is running at a speed of s or greater. Running more slowly would require an increase in speed at some later time. Both AVR and Optimal Available are additive and monotonic.

Now suppose there is an online algorithm for DSS-NS that is additive, monotonic and c_1-competitive. Let $f(s) = P(s) - P(0)$. Let c_2 be such that for all $x, y > 0$, $f(x + y) \leq c_2(f(x) + f(y))$. The competitive ratio of our online algorithm is at most $\max\{c_2c_1 + c_2 + 2, 4\}$. Using the upper bound for Optimal Available given by Bansal *et al.* , this yields an upper bound of 8 for quadratic power functions and 114 for cubic power functions. It should be noted that the biggest bottleneck for improvement for DSS-S is to devise more competitive algorithms for the version of the problem with no sleep state (DSS-NS). The best known upper bound for DSS-NS with a cubic power function is 27. This is obtained by plugging $d = 3$ into the upper bound of d^d proven by Bansal *et al.* for the competitive ratio of Optimal Available [1].

4 Problem Definition

First we define the Dynamic Speed Scaling problem with no *sleep* state (DSS-NS) and then augment the model with a *sleep* state. A system can execute jobs at different speeds. The power consumption rate of the system is a function $P(s)$ of the speed s at which it runs. The input consists of a set $\mathcal{J}$ of jobs. Each job j has a release time r_j and a deadline d_j. We will sometimes refer to the interval $[r_j, d_j]$ as j's *execution interval*. R_j is the number of units of work required to complete the job. A *schedule* is a pair $\mathcal{S} = (s, job)$ of functions defined over $[t_0, t_1]$. (t_0 is the first release time and t_1 is the last deadline). $s(t)$ indicates the speed of the system as a function of time and $job(t)$ indicates which job is being run at time t. $job(t)$ can be *null* if there is no job running at time t. A schedule is feasible if all

jobs are completed between the time of their release and deadline. That is, for all jobs j:

$$\int_{r_j}^{d_j} s(t)\delta(job(t), j)dt = R_j,$$

where $\delta(x, y)$ is 1 if $x = y$ and is 0 otherwise. The total energy consumed by a schedule $\mathcal{S}$ is

$$\texttt{cost}(\mathcal{S}) = \int_{t_0}^{t_1} P(s(t))dt.$$

The goal is to find for any problem instance a feasible schedule $\mathcal{S}$ which minimizes $\texttt{cost}(\mathcal{S})$.

In the Dynamic Speed Scaling Problem with sleep state (DSS-S), the system can be in one of two states: *on* or *sleep* . A schedule $\mathcal{S}$ now consists of a triplet $\mathcal{S} = (s, \phi, job)$ where $\phi(t)$ is defined over $[t_0, t_1]$ and indicates which state the system is in (*sleep* or *on*) as a function of t. The criteria for a feasible schedule is the same as in DSS-NS except that we place the additional constraint that if $\phi(t) = sleep$, then $s(t) = 0$. Power consumption is now defined by a function $P(s, \phi)$, where s is a non-negative real number representing the speed of the system and ϕ is the state. The power function is defined as follows:

$$P(s, \phi) = \begin{cases} P(s) & \text{if } \phi = on \\ 0 & \text{if } \phi = sleep \end{cases}$$

where $P(s)$ is a convex function. All values are normalized so that it costs the system 1 unit of energy to transition from the *sleep* to the *on* state. The value $P(0)$ will play an important role in the development of our algorithms. This is the power rate when the system is idle (i.e. speed is 0) and on. Throughout this paper, we make the assumption that $P(0) > 0$. Without this assumption, there is no benefit for the system to transition to a sleep state and the problem reduced to dynamic speed scaling with no sleep state. Note that the power consumption goes from $P(0)$ down to 0 when the system transitions to the sleep state.

Let k be the number of times that a schedule $\mathcal{S}$ transitions from the *sleep* state to the *on* state. The total energy consumed by $\mathcal{S}$ is

$$\texttt{cost}(\mathcal{S}) = k + \int_{t_0}^{t_1} P(s(t), \phi(t))dt.$$

We call the system *active* when it is running a job. The system is *idle* if it is not running a job. Note that when the system is idle, it can be in either the *on* or *sleep* state. However, if it is active, it must be in the *on* state.

For any power function $P(s)$ and instance $\mathcal{J}$, both DSS-NS and DSS-S are well defined and we can talk about the optimal schedule for both of these problems. In the case of DSS-NS, the optimal schedule does not change if the power function $P(s) - c$ is used instead for any constant c. Thus, we can assume for DSS-NS that $0 = P(0)$ and that no power is expended when the speed of the system is 0.

We assume throughout this paper that jobs are preemptive. Note that the difficult part of the problem is to determine $s(t)$, the speed at which the system will run. If there is a feasible schedule which uses speed $s(t)$, then the schedule which runs the system at speed $s(t)$ and uses the Earliest-Deadline-First

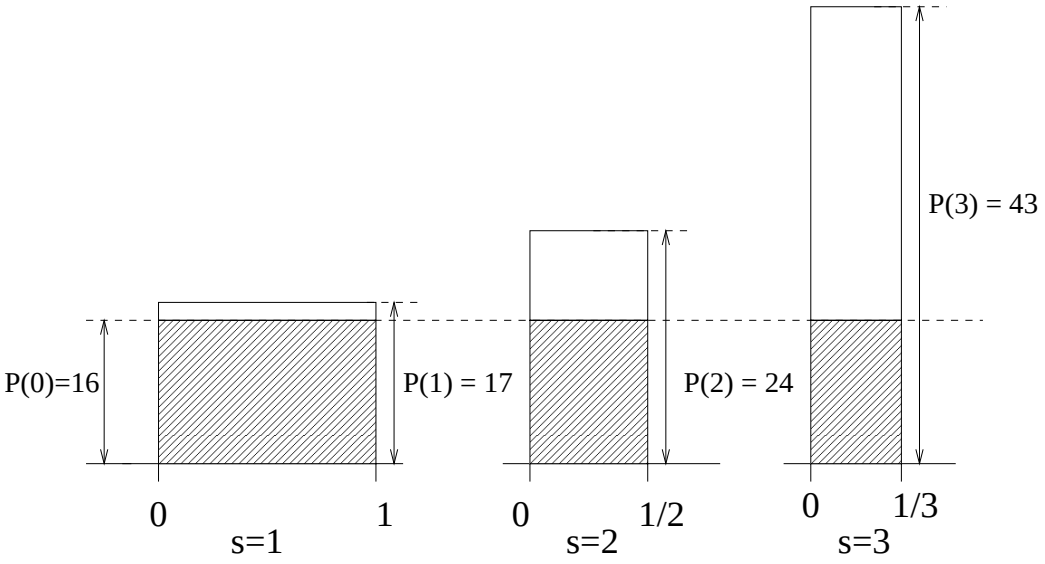

Figure 1: Example of energy consumed by a single job of size 1 run at three different speeds.

strategy to decide which job to run will result in a feasible schedule.

5 An Offline Algorithm

Consider the problem of running a single job in isolation. The system will be turned on to start the job and will go to the *sleep* state after the job is done regardless of when or how fast it is run. As a result, for this simple example we can disregard the transition cost in searching for an optimal strategy. Let R be the amount of work required to complete the job. If the job is executed at a constant speed of s, it will require power expenditure of $P(s)$ for a period of time of R/s. Thus, the optimal speed for the job would be the s that minimizes

$$P(s)\frac{R}{s}.$$

We call this speed the *critical speed* and denote it by s_{crit}. Note that the critical speed is independent of R, the size of the job.

Let us first examine a simple example. Consider $P(s) = s^3 + 16$ and a single job of size 1. Figure 5 shows three possible speeds at which the job could be run: 1, 2, and 3. The total area of the rectangle is the total energy expended. We have divided the energy in each case into two components:

- **Idle energy:** the energy spent keeping the system on. This is $P(0)$ times the length of execution of the job. (Shown shaded in Figure 5.)
- **Running energy:** the energy spent in running at a particular speed. This is just $P(s) - P(0)$ times the length of the execution of the job, when the job is run at a speed of s. (Shown not shaded in Figure 5.)

Because of the convexity of P, the running energy will not decrease as the speed increases. Since the

execution time decreases as speed increases, the idle energy decreases as the speed increases. By setting the derivative of $P(s)/s$ to zero and solving for s, we can see that when the speed is 2, the total energy is minimized.

In the example, the decrease in idle energy in going from a speed of 2 to a speed of 3 is $16(1/2-1/3) = 16/6$. Note however that this is actually an upper bound for the savings in idle energy this individual job would have in the context of a larger schedule since we have not taken into account the transition costs in going back to the on state in order to begin running a another job. However, since even this savings does not compensate for the additional running energy required to run the job faster, we can be sure that we would prefer a speed of 2 to a speed of 3 if the release times and deadlines allow for it. We can not reach the same conclusion however for the slower speeds. It may be more beneficial to run a job more slowly than the critical speed in the context of an entire schedule. This will depend on the length of the idle period created by running it faster and whether the savings in idle energy offsets the additional running costs and the transition costs.

Let us now examine the critical speed more carefully. The function that concerns us is $P(s)/s$. The derivative of this function is

$$\frac{sP'(s) - P(s)}{s^2}.$$

Since we are trying to determine where this derivative is 0, so we will only be concerned with where this function is negative, zero or positive. The denominator is positive for any $s > 0$, so we will focus on the numerator $sP'(s) - P(s)$ which we will denote by $F(s)$. The derivative of $F(s)$ is $sP''(s)$. Since P is assumed to be convex, this means that $F(s)$ is a non-decreasing function. At $s = 0$, $F(s) = -P(0)$. Recall our assumption that $P(0) > 0$ which implies that $F(0) < 0$.

There are two possibilities. The first is that the function $F(s)$ never reaches 0 for any $s \geq 0$. This is not a particularly interesting case as it means that $P(s)/s$ always decreases as s grows and we should execute all jobs as fast as possible. It also does not correspond to realistic models of power as a function of speed ([12]). As we have discussed above, real devices have an upper bound on the speed at which they can run. In this case, we could take the critical speed to be the maximum speed of the device or we could stick with our theoretical model and assume that the critical speed is infinite and each job that runs at the critical speed has an execution time of 0. In either case, the rest of the discussion in the paper would still hold.

The more interesting case is where $F(s)$ does in fact reach 0. We define the critical speed to be the smallest $s > 0$ for which $F(s) = 0$. (This is the smallest s for which $d(P(s)/s)/ds = 0$.) Moreover since $F(s)$ is non-decreasing, it is non-negative for any $s \geq s_{crit}$. Thus, the derivative of $P(s)/s$ is non-negative for any $s \geq s_{crit}$ and we have the following fact:

Fact 1 *If the critical speed s_{crit} is well defined for $P(s)$, then for any $s_{scrit} \leq s_1 \leq s_2$,*

$$\frac{P(s_{crit})}{s_{crit}} \leq \frac{P(s_1)}{s_1} \leq \frac{P(s_2)}{s_2}. \tag{1}$$

5.1 Scheduling Fast Jobs

It will be useful at this point to prove a few properties of an optimal schedule for DSS-S. The following lemma is noted in [17] without proof. For completeness, we include the proof.

Lemma 2 *Consider an instance $\mathcal{J}$ of DSS-S with power function P. There is an optimal schedule in which for every job j in $\mathcal{J}$, there is a constant speed s_j such that whenever $job(j,t) = 1$, $s(t) = s_j$.*

Proof. The lemma states that there is an optimal schedule in which each job is run at a constant speed, although not necessarily in one contiguous interval. The lemma follows directly from the convexity of P. Consider a schedule $\mathcal{S}$. If there is a job that is not run at uniform speed, we can transform it into a schedule in which the job function remains unchanged (i.e. all jobs are worked on during the same intervals as before), and jobs j is run at a uniform speed. Furthermore, the total energy consumed by the new schedule is no greater than the original schedule. This process can then be iterated until each job is run at its own uniform speed.

Fix the intervals in which the system works on job j. That is, fix the intervals in which $\delta(job(t), j) = 1$. The total amount of work done during these intervals

$$\int_{r_j}^{d_j} s(t)\delta(job(t), j)dt,$$

must be a constant R_j. Given this fact, since the power function P is convex, the energy consumed in completing job j

$$\int_{r_j}^{d_j} P(s(t))\delta(job(t), j)dt,$$

is minimized when the speed is uniform during these intervals. This is a direct result of the continuous version of Jensen's Inequality. Since $\int \delta(job(t), j)dt$ is the total length of the intervals during which the system works on job j, this uniform speed is $s_j = R_j / \int \delta(job(t), j)dt$. Change the speed function s so that $s(t) = s_j$ whenever $job(t) = j$. □

Now we need a few definitions:

- A *partial schedule* for both DSS-S and DSS-NS is a specification for the functions (s, job) in the case of DSS-NS and (s, ϕ, job) for DSS-S defined over a finite set of intervals. We will only consider partial schedules that have the property that for each job, either all of the work or none of the work for that job is completed during the portions of s and job that have been defined.

- A partial schedule $\mathcal{P}'$ is an *extension* of a partial schedule $\mathcal{P}$ if the intervals over which $\mathcal{P}'$ is specified include all the intervals over which $\mathcal{P}$ is specified.

- A *complete* schedule (or just a *schedule*) is a partial schedule in which the functions are defined over the entire interval of interest (i.e. from the first release time to the the last deadline).

- An extension of a partial schedule $\mathcal{P}$ is said to be *optimal* if it is complete and has the minimum cost among all extensions of $\mathcal{P}$.

Lemma 3 *Let $\mathcal{P}$ be a partial schedule for an instance $\mathcal{J}$ of DSS-S with power function P. There is an optimal extension of $\mathcal{P}$ in which every job j that remains undefined in $\mathcal{P}$ is run at a uniform speed s_j. Furthermore, the system never runs more slowly than s_j in the extension during those portions of interval $[r_j, d_j]$ that are not already scheduled under $\mathcal{P}$.*

Proof. We introduce a dummy cost function. The cost of a schedule according to the dummy cost function is

$$\int (s(t))^2 dt.$$

We use the dummy cost function to break ties among schedules that have the same cost according to the real cost function. Let $\mathcal{S}$ be an optimal extension of $\mathcal{P}$ such that it has the smallest cost according to the dummy cost function among all optimal extensions of $\mathcal{P}$. We will prove that $\mathcal{S}$ has the property that we desire. It will be convenient to assume for the remainer of this proof that when we refer to some interval I, we are referring to those portions of I that have not already been scheduled under $\mathcal{P}$.

From Lemma 2, we can assume that each remaining job is run at its own uniform speed s_j. Suppose that jobs j and k are not defined in $\mathcal{P}$ and that in $\mathcal{S}$, there is an interval $I_k \subseteq [r_j, d_j]$ such that the system works on job k at speed $s_k < s_j$ during I_k. Let I_j be an interval during which the system works on job j. The total amount of energy consumed during these two intervals is $|I_k|P(s_k) + |I_j|P(s_j)$. The total amount of work completed is $|I_k|s_k + |I_j|s_j$. Now we will change the schedule so that the system runs at a constant speed of $\lambda s_k + (1-\lambda)s_j$, where $\lambda = |I_k|/(|I_k| + |I_j|)$. Note that this speed is strictly greater than s_k and strictly less than s_j. Also note that the same amount of work is completed as before. In the new schedule, the work that was previously done in I_k will take a little less time and the work that now spills over from I_j can be completed during the extra time created in I_k. This is possible because only job j is worked on in I_j and $I_k \subseteq [r_j, d_j]$. The energy expended under the new schedule in $I_k \cup I_j$ is $(|I_j| + |I_k|)P(\lambda s_k + (1-\lambda)s_j)$. The energy expended under the old schedule in $I_k \cup I_j$ is $(|I_j| + |I_k|)(\lambda P(s_k) + (1-\lambda)P(s_j))$. Because P is convex, the energy expended under the new schedule is no greater than the energy expended under the old schedule. If the power consumed in the new schedule is strictly less than that used in the old schedule, this contradicts the optimality of $\mathcal{S}$. If they are the same, then we will show that the new schedule has a strictly lower value according to the dummy cost function, again contradicting our assumption about $\mathcal{S}$. Since $s_k < s_j$,

$$(|I_k| + |I_j|)[(\lambda s_k)^2 + ((1-\lambda)s_j)^2] > (|I_k| + |I_j|)(\lambda s_k + (1-\lambda)s_j)^2.$$

The expression on the left is the cost under the dummy cost function of the original schedule and the expression on the right is the cost under the dummy cost function of the new schedule. □

It will be useful now to describe the optimal offline algorithm for DSS-NS given by Yao *et al.* in [17]. The algorithm of Yao *et al.* always maintains some partial schedule in which subset $\mathcal{J}'$ of the jobs are scheduled for a set of time intervals $\mathcal{I}$. The intervals $\mathcal{I}$ are said to be *blacked out* and the system is reserved only for jobs in $\mathcal{J}'$ during these times. In a given iteration, a subset of the remaining jobs are selected. The current schedule is extended so that this subset of jobs are all completed at the same constant speed (although not necessarily during a single contiguous interval). The intervals of time that

```
OPTIMAL-DSS-NOSLEEP(J)
    While 𝒥 is not empty:
        Let [z, z'] be the interval of maximum intensity.
        Let 𝒥' be the set of jobs in 𝒥 that are contained in [z, z']
        Define s(t) and job(t) for all t ∈ [z, z'] that are not already
            contained in a blacked-out interval by scheduling all jobs in 𝒥'
            using Earliest Deadline First at a speed of g(z, z')
        Black out the interval [z, z'].
        Remove all jobs in 𝒥' from 𝒥.
        For any j ∈ 𝒥,
            if r_j ∈ [z, z'] then r_j ← z'.
            if d_j ∈ [z, z'] then d_j ← z.
```

are used to execute these jobs are then *blacked out*, the newly scheduled jobs are added to $\mathcal{J}'$ and all the remaining jobs must be executed during the remaining time that is not blacked out.

We will prove several facts about this algorithm. First of all, in each successive iteration, the speed at which each set of jobs is run does not increase. Furthermore, for all the jobs that are scheduled in this manner at a speed of s_{crit} or greater, there is an optimal schedule for DSS-S that schedules these jobs in exactly the same way. This means that we can follow the optimal algorithm DSS-NS problem until a job is scheduled at a speed less than the critical speed. At this point, we are left with a partial schedule in which all the remaining jobs can be completed at a speed which is no greater than s_{crit}. The difficulty then is to determine how to schedule these remaining jobs.

We now describe the details of the optimal algorithm for DSS-NS. A job j is said to be *contained* in interval $[z, z']$ if $[r_j, d_j] \subseteq [z, z']$. For any interval $[z, z']$, define $l(z, z')$ to be the length of the interval excluding those intervals in which the machine has already been blacked out. Define the intensity of interval $[z, z']$ to be

$$g(z, z') = \frac{\sum_{j \text{ such that } [r_j, d_j] \subseteq [z, z']} R_j}{l(z, z')}. \tag{2}$$

An iteration of the optimal algorithm for DSS-NS proceeds by selecting an interval $[z, z']$ of maximum intensity. It schedules all jobs contained in $[z, z']$ at a speed of $g(z, z')$ using Earliest-Deadline-First and then blacks out the interval $[z, z']$. The pseudocode for the algorithm is given in Figure 5.1. The scheduled jobs are then removed from the set of remaining jobs. The algorithm iterates in this manner until all jobs are scheduled. At any point in time, given the schedule that has been determined so far, it is clear that $g(z, z')$ is a lower bound for the average speed during the interval $[z, z']$. The proof of optimality entails proving that for the interval $[z, z']$ of maximum intensity, the optimal schedule runs the jobs contained in $[z, z']$ at a speed of exactly $g(z, z')$.

Suppose that the algorithm iterates r times. Let $I_1, I_2, \ldots, I_r$ be the sequence of intervals that are blacked out in the course of the algorithm. Let $g(I_k)$ be the critical speed of I_k at the time that it is blacked out and let S_k be the set of jobs that are contained in I_k at the time that I_k is blacked out (i.e. the jobs that are scheduled in I_k). We need the following lemma:

Lemma 4 *For any $k = 2, \ldots r$, $g(I_{k-1}) \geq g(I_k)$.*

Proof. Suppose the lemma is not true. Consider the first k such that $g(I_{k-1}) < g(I_k)$. If I_k and I_{k-1} are not contiguous, then exactly the same set of jobs are contained in I_k before and after I_{k-1} is blacked out. This means that I_k must have higher intensity than I_{k-1} before I_{k-1} was blacked out which contradicts the fact that we always black out the interval with the highest intensity. If I_k and I_{k-1} are contiguous, then before I_{k-1} is blacked out, the intensity of $I_k \cup I_{k-1}$ is

$$\frac{\sum_{j \in S_{k-1}} R_j + \sum_{j \in S_k} R_j}{|I_{k-1}| + |I_k|}.$$

This value is strictly greater than $g(I_{k-1})$:

$$g(I_{k-1}) = \frac{\sum_{j \in S_{k-1}} R_j}{|I_{k-1}|} < \frac{\sum_{j \in S_{k-1}} R_j + \sum_{j \in S_k} R_j}{|I_{k-1}| + |I_k|} < \frac{\sum_{j \in S_k} R_j}{|I_k|} = g(I_k).$$

This inequality holds because the expression in the middle is a weighted average of the expressions on either side. This contradicts the fact that the interval of maximum intensity was chosen when I_{k-1} was blacked out. □

This lemma tells us that if you look at the order in which jobs are scheduled, they are run at slower and slower speeds. The next lemma tells us that an optimal algorithm for DSS-S can use the optimal algorithm for DSS-NS, scheduling jobs in the same manner until the speed at which a job will be scheduled drops below s_{crit}.

Lemma 5 *Suppose that we have a partial schedule $\mathcal{P}$ for the DSS-S problem. Let I be the interval of maximum intensity. If the intensity of interval I is at least s_{crit}, then there is an optimal extension of $\mathcal{P}$ in which only jobs contained in I are scheduled during interval I. Furthermore the optimal schedule schedules all jobs contained in I using Earliest-Deadline-First with no* sleep *periods.*

Proof. Consider an optimal extension $\mathcal{S}$ of $\mathcal{P}$. By Lemmas 2 and 3, we can assume that each job j not scheduled in $\mathcal{P}$ are run at its own uniform speed s_j. Furthermore, the system never runs slower than s_j during those intervals in $[r_j, d_j]$ that are not already scheduled under $\mathcal{P}$.

Let I be an interval of maximum intensity and suppose that the system works on some job j during I that is not contained in I. It must be the case then that at some point in $\mathcal{S}$, the speed of the system is greater than $g(I)$ during interval I, since the average speed during I must be at least $g(I)$, even when job j is excluded. Let I' be a maximal interval in which the system runs faster than $g(I)$ during all the time in I' that it is not already scheduled under $\mathcal{P}$. It must be the case that there is some job k that the system works on during I' and whose interval of execution is not contained in I' (otherwise, we have found an interval of greater intensity than I). Since I' was chosen to be maximal, the system must have speed at most $g(I)$ during some portion of $[r_k, d_k]$ and yet the system works on k at a speed strictly greater than $g(I)$. This contradicts our assumptions about $\mathcal{S}$.

Now we will establish that the jobs contained in I can be scheduled optimally at a speed of $g(I)$ with no transition to the *sleep* state. Let R be the total amount of work required for the jobs contained in I. Define $s(t)$ to be the average speed of the system in interval I if a total of time t is spent in the

sleep state. $s(t) = R/(|I| - t)$. Because P is convex, the expression:

$$P\left(\frac{R}{|I|-t}\right)(|I|-t) = \left(\frac{P(s(t))}{s(t)}\right)R$$

is a lower bound for the energy expended during the interval I. Note that $s(t) > s(0) = g(I) \geq s_{crit}$. Fact 1 indicates that running at speed $s(0)$ with no sleep time will minimize the total energy consumed in the interval. If all jobs in I are run at a speed of $s(0) = g(I)$, Earliest-Deadline-First will produce a feasible schedule or else there is an interval of greater intensity. □

5.2 Scheduling Slow Jobs

Let $\mathcal{J}_{fast}$ denote the set of jobs that are scheduled according to the optimal algorithm for DSS-NS at a speed of s_{crit} or higher. This subsection will focus only on the remaining jobs and will only account for the energy expended when the system is not running a job from $\mathcal{J}_{fast}$. We can readjust the release times and deadlines for these remaining jobs so that they do not occur during a blacked-out interval. Release times will be moved to the end of the blacked-out interval and deadlines will be adjusted to the beginning of the blacked-out interval.

Now we must decide at what speed and at what time to run the jobs which would run slower than s_{crit} in the no-sleep version of the problem. We are guaranteed that there is a feasible solution in which these remaining jobs run no faster than s_{crit}. Our algorithm decides to run all jobs at a speed of s_{crit}. Any algorithm that makes this choice will be active and idle for the same amount of time. The algorithm must decide during what intervals of time the system will be idle given the release times and deadlines of the jobs. When all these idle periods have been determined, it is decided whether the system will transition into the *sleep* state or not during each such interval (depending on whether the interval has length at least $1/P(0)$). Naturally, then it would be better to have fewer and longer idle periods (as opposed to many fragmented idle periods) since that gives the algorithm the opportunity to transition to the *sleep* state and save energy with fewer start-up costs.

Note that one could further improve the performance of the algorithm by using our method only to determine when the job is in the *sleep* state and then re-running the optimal algorithm for DSS-NS with all the sleep intervals blacked out. This would have the effect of allowing the algorithm to use idle intervals that are too short to transition to the *sleep* state. Some jobs would then run more slowly and save energy. However, we will bound the algorithm without this final energy-reducing step.

A job is said to be *pending* at time t if it's release time has been reached but it has not yet been completed. All jobs are run at speed s_{crit}. We will assume that the system is in the *on* state when the first job arrives. Thus if t_0 is the first release time, the system starts running the task with the earliest deadline among all those which arrive at time t_0. The subsequent events are handled according to the algorithm given in the figures below. The basic idea is that while the algorithm is active it stays active, running jobs until there are no more jobs to run. When it becomes idle, it stays idle as long as it possibly can until it has to wake-up in order to complete all jobs by their deadline at a speed of s_{crit}. The algorithm is called LeftToRight because it sweeps through time from left to right. Throughout this paper, we think of time as the x-axis. An event is said to be to the 'left' of another event if it

occurs earlier. Similarly, an event is said to be to the 'right' of another event if it occurs later.

```
LeftToRight:FindIdleIntervals(J)
Simulate the execution of jobs through time
as they have been schedules so far.  Stop and
update the schedule when the current time t
is one of the following events:
(1) if a new job j arrives at time t
(2)     if the system is currently
        running a job
(3)         Run the job with the
            earliest deadline
(4)     if the system is not currently
        running a job
(5)         SetWakeUpTime()
(6) if the system completes a job at time t
(7)     if there are pending jobs,
(8)         work on the pending job
            with the earliest deadline
(9)     if there are no pending jobs,
(10)        SetWakeUpTime()
(11)if the wake-up time is reached at time t
(12)    Start working on pending job
        with earliest deadline.
(13)if the beginning of a scheduled
    interval is reached at time t
(14)    Process the jobs from I_fast which
        were scheduled for this interval
(15)if the end of a scheduled interval
    is reached at time t
(16)    Complete lines (7)-(10)
```

Figure 2: The algorithm LeftToRight.

Theorem 6 *If the power function $P(s)$ is convex, then the algorithm Left-To-Right achieves an approximation ratio of* 2.

Before proving Theorem 6, we prove the following useful lemmas. The first established that the algorithm given in Figure 3, calculates the latest time the system can wake up and still complete all jobs by their deadline at the critical speed.

Lemma 7 *If the system is idle at time t, then t_w computed in Figure 3 is the largest value such that the system can remain idle for the interval $[t, t_w]$ and complete all jobs in $\mathcal{J} - \mathcal{J}_{fast}$ by their deadlines at a speed of s_{crit}.*

Proof. Recall the definition of the intensity of an interval given in Equation 2. For the purposes of this proof, we will consider only jobs in $\mathcal{J} - \mathcal{J}_{fast}$ in determining the intensity of an interval.

```
SetWakeUpTime()
    Order all the jobs in J − J_fast according to their
    deadlines.  Thus we have t ≤ d_1 ≤ ··· ≤ d_k.
    For each j ∈ {1,...,k}, let
        t_j = d_j − (Σ_{j=1}^{k} R_j/s_crit)
    Let t_w = min_j t_j
    Set the wake-up time to be t_w.
```

Figure 3: The procedure SetWakeUpTime() called from LeftToRight.

If the system wakes up at time t_w, then for any job in $\mathcal{J} - \mathcal{J}_{fast}$ whose execution interval contains t_w, its arrival time is effectively t_w. What we want to know, is the largest value for t_w such that the intensity of all intervals is no more than s_{crit}. Then according to the Yao *et al.* algorithm, all jobs can be scheduled after t_w at a speed no greater than s_{crit}.

We need only consider intervals that begin at time t_w. This is because, by definition, all jobs in $\mathcal{J} - \mathcal{J}_{fast}$ can be completed by their deadlines at a speed of s_{crit} or less. Thus, for any arrival time a and deadline d such that $t_w < a < d$, the intensity of the interval $[a, d]$ is no more than s_{crit}.

Order the jobs in $\mathcal{J} - \mathcal{J}_{fast}$ that have not been completed at time t by their deadline. Thus, we have $t \leq d_1 \leq \cdots \leq d_k$. Each deadline d_j is a candidate right endpoint of the interval of maximum intensity. Jobs 1 through j must be completed by d_j. If the system is running at speed s_{crit}, it will require a time interval of $\sum_{j=1}^{k} R_j/s_{crit}$ to finish these jobs at a speed of s_{crit} which means the system must begin by time $d_j - \sum_{j=1}^{k} R_j/s_{crit}$. Thus, if we wait until $\min_j t_j$, the jobs contained in the maximum intensity interval will require a speed of exactly s_{crit} to complete. Waiting any longer will force the algorithm to run faster than s_{crit}. □

In the proof of the next lemma as well as the proof of the theorem, we let $\mathcal{S}_{OPT}$ be the optimal schedule for a particular input. Let $\mathcal{S}_{LTR}$ be the schedule produced by the Left-To-Right algorithm on the same input. Let $\mathcal{P}_{OPT}$ (resp. $\mathcal{P}_{LTR}$) denote the set of maximal intervals during which the system is in the sleep state for $\mathcal{S}_{OPT}$ (resp. $\mathcal{S}_{LTR}$). Let $\mathcal{D}_{OPT}$ (resp. $\mathcal{D}_{LTR}$) denote the set of maximal intervals during which the system is idle in $\mathcal{S}_{OPT}$ (resp. $\mathcal{S}_{LTR}$). Recall that when the system is asleep, it is in the sleep state. When the system is idle, it is running at speed zero. Note that the system can either be in the sleep or the active state when it is idle.

Lemma 8 *At most two intervals from $\mathcal{D}_{LTR}$ can intersect a single interval from $\mathcal{P}_{OPT}$.*

Proof. Suppose to the contrary that there is an interval $I \in \mathcal{P}_{OPT}$ which is intersected by three intervals $A, B, C \in \mathcal{D}_{LTR}$. Suppose without loss of generality that A is to the left of B and B is to the left of C. Refer to Figure 4. Consider the first job j that LTR runs when it wakes up after B. It must be the case that j has a release time after the beginning of B or else the system would not have gone idle at the beginning of B. Recall that LTR does not go idle if there are any pending jobs in the system. Job j can not have a deadline after the beginning of C or else the system would have delayed starting work at the end of B since it delays waking up until it is necessary in order to complete all jobs

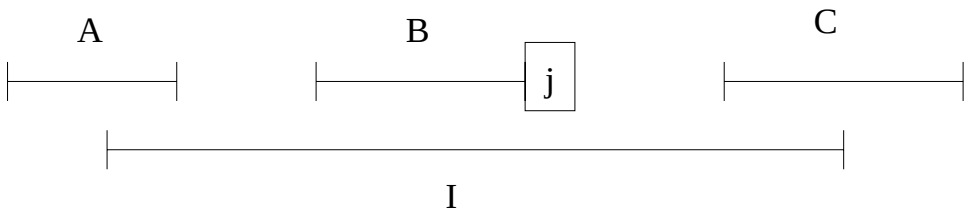

Figure 4: Figure for proof of Lemma 8

by their deadlines at a speed of s_{crit}.

This means that the execution interval for j must be contained in the interval from the beginning of B to the beginning of C. However, this interval of time is completely contained in I during which the optimal schedule is idle. This implies that the optimal schedule can not complete job j. □

It will be useful to isolate certain portions of the energy expenditure for a schedule $\mathcal{S} = (s, \phi, job)$ as follows:

1. The energy expended while the system is active. Let $\delta_s(t) = 1$ if $s(t) > 0$ and 0 otherwise.

$$\texttt{active}(\mathcal{S}) = \int_{t_0}^{t_1} P(s(t))\delta_s(t)dt.$$

2. The cost to keep the system active or shut-down and wake-up the system during the idle periods (depending on which action is the most energy efficient). Let $\mathcal{D}$ be the set of idle periods for the schedule $\mathcal{S}$.

$$\texttt{idle}(\mathcal{S}) = \sum_{I \in \mathcal{D}} \min(P(0)|I|, 1).$$

3. The cost to keep the system in the *on* state while the system is on

$$\texttt{on}(\mathcal{S}) = \int_{t_0}^{t_1} P(0)\delta(\phi(t), on\ )dt.$$

4. The cost to wake-up the system at the end of each sleep interval. If $\mathcal{I}$ is the set of maximal intervals in which the algorithm is in the *sleep* state, this is just the number of intervals in $\mathcal{I}$. We denote this by $\texttt{sleep}(\mathcal{S})$.

Fix a problem instance. We will prove the following two lemmas from which Theorem 6 follows easily.

Lemma 9 $\texttt{active}(\mathcal{S}_{LTR}) \leq \texttt{active}(\mathcal{S}_{OPT})$.

Lemma 10 $\texttt{idle}(\mathcal{S}_{LTR}) \leq \texttt{on}(\mathcal{S}_{OPT}) + 2\texttt{sleep}(\mathcal{S}_{OPT})$. [1]

Proof of Theorem 6

$$\texttt{cost}(\mathcal{S}_{LTR}) \quad = \quad \texttt{active}(\mathcal{S}_{LTR}) + \texttt{idle}(\mathcal{S}_{LTR})$$

[1] The original version of this lemma only proved that $\texttt{idle}(\mathcal{S}_{LTR}) \leq \texttt{on}(\mathcal{S}_{OPT}) + 3\texttt{sleep}(\mathcal{S}_{OPT})$. The factor of 3 was improved to 2 due to an observation of Kimbrel, Schieber, and Sviridenko that strengthened Lemma 8.

$$
\begin{aligned}
&\leq \texttt{active}(\mathcal{S}_{OPT}) + \texttt{on}(\mathcal{S}_{OPT}) + 2\texttt{sleep}(\mathcal{S}_{OPT}) \\
&\leq 2\texttt{active}(\mathcal{S}_{OPT}) + 2\texttt{idle}(\mathcal{S}_{OPT}) \\
&= 2\texttt{cost}(\mathcal{S}_{OPT})
\end{aligned}
$$

The first inequality uses the fact that for any schedule $\mathcal{S}$, $\texttt{cost}(\mathcal{S}) = \texttt{active}(\mathcal{S}) + \texttt{idle}(\mathcal{S})$. The next inequality comes from applying Lemmas 9 and 10. Now we will divide the energy in $\texttt{on}(\mathcal{S}_{OPT})$ in two parts. The first part is the energy expended in keeping the system on while it is active and the second is the energy in keeping the system on while it is idle (but not sleeping). This first part can be bounded by $\texttt{active}(\mathcal{S}_{OPT})$. To bound the second part, we observe that the energy spent when the system is idle idle consists of the cost to keep the system *on* when idle (the second part of `on`) and the cost of waking the system up after a sleep period (`sleep`). Thus, the second part of `on` plus `sleep` is bounded by `idle`. □

We now give the proofs for the two lemmas stated above.

Proof of Lemma 9. We will prove the lemma for each job j. That is, the total energy expended by LTR in running job j is bounded by the total energy expended by the optimal in running jobs j. Since the system must be running some job while it is active, the lemma follows.

Lemma 2 tells us that we can assume that the optimal schedule runs job j at a uniform speed s_j. The optimal algorithm spends a total time of R_j/s_j on job j for a total energy expenditure of $P(s_j)R_j/s_j$. LTR runs the job at speed s_{crit} for a total energy expenditure of $P(s_{crit})R_j/s_{crit}$. Since s_{crit} is the value for s that minimizes $P(s)/s$, the lemma follows. □

Proof of Lemma 10. Recall that $\mathcal{D}_{LTR}$ is the set of maximal intervals in which the system is idle under Left-To-Right's schedule and $\mathcal{P}_{OPT}$ is the set of maximal intervals during which the system is in the *sleep* state in the optimal schedule. First consider the intervals in $\mathcal{D}_{LTR}$ which have no intersection with any interval in $\mathcal{P}_{OPT}$. The sum of the lengths of these intervals is at most the total length of time that the optimal algorithm is in the *on* state. Since the cost of any interval is bounded by $P(0)$ times its length, the cost of all these intervals is at most $\texttt{on}(\mathcal{S}_{OPT})$.

Next consider the intervals in $\mathcal{D}_{LTR}$ that have a non-zero intersection with some interval in $\mathcal{P}_{OPT}$. By Lemma 8, each interval in $\mathcal{P}_{OPT}$ intersects at most two intervals from $\mathcal{D}_{LTR}$. Thus, the number of intervals in $\mathcal{D}_{LTR}$ which have a non-zero intersection with an interval in $\mathcal{P}_{OPT}$ is at most two times the number of intervals in $\mathcal{P}_{OPT}$ which is exactly $2\texttt{sleep}(\mathcal{S}_{OPT})$. Since the cost of any interval in $\mathcal{D}_{LTR}$ is bounded by 1, the cost of all the intervals in $\mathcal{D}_{LTR}$ that intersect an interval in $\mathcal{P}_{OPT}$ is at most $2\texttt{sleep}(\mathcal{S}_{OPT})$. □

6 An Online Algorithm

The online algorithm for DSS-S which we present here makes use of an online algorithm A for DSS-NS that is additive and monotonic. At this point in time, the only known competitive algorithm for DSS-NS is the Average Rate algorithm given by Yao *et al.* which does have both properties. We will use $s_A(t, \mathcal{J})$

to denote the speed of the system as a function of time chosen by A when the input conists of the jobs in $\mathcal{J}$.

Our algorithm runs in two different modes: *fast* mode and *slow* mode. The algorithm is in slow mode if and only if it is feasible to complete all pending jobs by their deadline at a speed of s_{crit}. We maintain a set of jobs $\mathcal{J}_{fast}$. When a job arrives and the algorithm is in fast mode or if the release of a job causes the system to transition to fast mode, it is placed in $\mathcal{J}_{fast}$. When the system transitions back to slow mode, $\mathcal{J}_{fast} \leftarrow \emptyset$. The algorithm maintains two speed functions $s_{slow}(t)$ and $s_{fast}(t)$. The speed of the system is always $s_{slow}(t) + s_{fast}(t)$ evaluated at the current time. $s_{fast}(t)$ is always $s_A(t, \mathcal{J}_{fast})$. Since jobs are only added to $\mathcal{J}_{fast}$ when the algorithm is in fast mode and $\mathcal{J}_{fast}$ is set to be the empty set when the algorithm transitions back to slow mode, this means that $s_{fast}(t) = 0$ when the algorithm is in slow mode.

s_{slow} is always s_{crit} or 0. To specify $s(t)$, it remains to determine when $s_{slow}(t)$ is s_{crit} and when it is 0. The algorithm maintains a current plan for $s_{slow}(t)$ and only alters this plan at three types of events:

1. A new job arrives and the algorithm remains in slow mode.

2. The algorithm transitions from fast mode to slow mode.

3. A new job arrives when the system is idle, causing the system to transition to fast mode.

In each case $s_{slow}(t)$ is set as follows. $t_{current}$ will denote the current time. If the system is currently active or just becoming active, then let t_{start} be the current time. If the system is idle, then let t_{start} be the latest time t such that if all pending jobs are run at a speed of s_{crit} starting at time t, they will finish by their deadlines. Let R denote the remaining work of all pending jobs in the system that are not in $\mathcal{J}_{fast}$.

$$s_{slow}(t) = \begin{cases} s_{crit} & \text{for } t_{start} \leq t \leq R/s_{crit} + t_{start} \\ 0 & \text{for } t > R/s_{crit} + t_{start} \text{ or } t_{current} \leq t < t_{start} \end{cases}$$

We define the notion of the *excess at time t* to help in determining when the algorithm needs to switch from the fast mode to the slow mode. This value is simply the total amount of work that would not get completed by its deadline if the algorithm were to use speed s_{crit}. If the algorithm is in slow mode, it just needs to check whenever a new job arrives that the excess is 0 to see whether it needs to transition to fast mode. When the algorithm transitions to fast mode (or whenever a new job arrives when it is in fast mode), it computes a *slow-down* time which is the next time that the system can transition to slow mode unless new jobs arrive. This is the smallest value t_s such that

$$\int_{t_{current}}^{t_s} (s_{fast}(t) + s_{slow}(t) - s_{crit})dt \geq \quad \text{excess at the current time.}$$

If the system becomes idle, it maintains a wake-up time t_w which is the latest time such that all pending jobs can be completed at a speed of s_{crit} if it wakes up at time t_w. If a new job arrives, it may have to update t_w to some earlier point in time. If the new job is large enough it may have to wake up immediately and transition to fast mode.

When the system becomes idle, it will transition to the *sleep* state if the idle period lasts at least time $1/P(0)$. Since the algorithm postpones processing any jobs until it is absolutely necessary in

order to complete all pending at speed s_{crit}, we call the algorithm PROCRASTINATOR. The algorithm is defined in the figures below. The figures show how PROCRASTINATOR determines the functions $s_{slow}(t)$ and $s_{fast}(t)$. The algorithm maintains a value for these functions for all t after the current time and then periodically updates these values. The speed of the system at the current time is always $s_{slow}(t_{current}) + s_{fast}(t_{current})$. All jobs are scheduled by the EDF policy.

```
PROCRASTINATOR:DETERMINESPEED(J)
(1)     if a new job j arrives
(2)         if the system is in fast mode
(3)             J_fast ← J_fast ∪ {j}.
(4)             s_fast(t) = s_A(t, J_fast) for t > t_current
(5)             SETSLOWDOWNTIME()
(6)         if the system is in slow mode
(7)             if pending jobs can be completed at rate s_crit,
(8)                 if system is idle SETWAKEUPTIME()
(9)                 RESET-S-SLOW()
(10)            if pending jobs can not be completed at rate s_crit,
(11)                Transition to fast mode.
(12)                J_fast ← {j}.
(13)                s_fast(t) = s_A(t, J_fast) for t > t_current
(14)                if system is idle, set wake up time t_w to current time.
(15)                RESET-S-SLOW()
(16)                SETSLOWDOWNTIME()
(17)    if the system completes a job
(18)         if there are no pending jobs,
(19)            Set timer to 1/P(0).
(20)    if wake-up time is reached
(21)        if system is in sleep state
(22)            Transition to on state.
(23)        Start working on pending job with earliest deadline.
(24)        Clear timer.
(25)    if timer expires,
(26)        Transition to sleep state.
(27)    if the slowdown time is reached,
(28)        Transition to slow mode.
(29)        J_fast ← ∅
(30)        Set s_fast(t) = 0 for all t > t_current.
(31)        RESET-S-SLOW()
```

For the lemmas that follow, $\mathcal{S}_P$ will denote the schedule for PROCRASTINATOR. Let $\mathcal{P}_P$ denote the set of maximal intervals during which the system is in the sleep state for $\mathcal{S}_P$. Let $\mathcal{D}_P$ denote the set of maximal intervals during which the system idle in $\mathcal{S}_P$ and let $s_P(t)$ denote the speed function in $\mathcal{S}_P$.

Lemma 11 *No single interval in $\mathcal{P}_{OPT}$ can intersect more than two intervals in $\mathcal{D}_P$.*

SetWakeUpTime()

Order all the pending jobs according to their deadlines. Thus we have $t \le d_1 \le \cdots \le d_k$.

For each $j \in \{1, \ldots, k\}$, let

$t_j = d_j - \left(\sum_{j=1}^{k} R_j / s_{crit}\right)$

Let $t_w = \min_j t_j$

Set the *wake-up* time to be t_w.

SetSlowDownTime()

Compute E, the excess at the current time.

Set the slowdown time to be the minimum value for t_s which satisfies

$\int_{t_{current}}^{t_s} (s_{fast}(t) + s_{slow}(t) - s_{crit}) \ge E$.

Reset-S-slow()

Let R be the total amount of work left on pending jobs that are not in $\mathcal{J}_{fast}$.

If the system is idle, let t_{start} be the wake-up time.

If the system is active, let t_{start} be the current time.

Set $s_{slow}(t) = s_{crit}$

for $t_{start} \le t \le t_{start} + R/s_{crit}$

Set $s_{slow}(t) = 0$

for $t > t_{start} + R/s_{crit}$ and $t_{current} \le t < t_{start}$

Proof. Similar to the proof of Lemma 8 except for one case. This is the case where job j's deadline is after the beginning of C. If the algorithm is in slow mode when it wakes up and starts work on j, the argument is the same as in Lemma 8. The only case that needs to be addressed is if the release of job j causes the algorithm to wake-up in fast mode. We will argue that in this case, the algorithm must stay busy until j's deadline.

At any point the algorithm is in fast mode define the excess at time t to be the amount of work that would not get completed if the algorithm performed the EDF algorithm at speed s_{crit}. The algorithm is in fast mode if and only if the excess is greater than 0. As long as the excess is greater than 0, there are jobs in the system and the system stays active. Suppose that the excess reaches 0 at some time $\hat{t} \in [r_j, d_j]$. If there is an idle period anywhere in $[\hat{t}, d_j]$ then that time could have been used to work on j at speed s_{crit} which means that the excess would have reached 0 before time $\hat{t}$. □

Lemma 12 *Fix an input $\mathcal{J}$ and let $s_{fast}(t)$ denote the function s_{fast} produced by* PROCRASTINATOR *on input $\mathcal{J}$. For all t, $s_{fast}(t) \leq s_{A,\mathcal{J}}(t)$.*

Proof. Suppose that Procrastinator transitions to fast mode k times. Let $\mathcal{J}_i$ denote the set of jobs that arrive while the system is in fast mode for the i^{th} time. Let I_i denote the interval of time in which the system is in fast mode for the i^{th} time. The I_i's are disjoint as are the $\mathcal{J}_i$'s. Furthermore, $s_{fast}(t) = 0$ for any t that is not contained in the union of the I_i's. Since all jobs that arrive when the system is in fast mode are scheduled according to algorithm A, we know that for $t \in I_i$, $s_{fast}(t) = s_{A,\mathcal{J}_i}(t)$. This means that for all t,

$$s_{fast}(t) \leq \max_{1 \leq i \leq k} s_{A,\mathcal{J}_i}(t).$$

The fact that A is an additive algorithm means that for all t,

$$\max_{1 \leq i \leq k} s_{A,\mathcal{J}_i}(t) \leq s_{A,\mathcal{J}}(t).$$

□

Lemma 13 *Whenever the system is active under the algorithm* PROCRASTINATOR*, its speed is at least s_{crit}.*

Proof. Suppose that the system is active at the current time t_c. We will first show that the two projected speed functions s_{slow} and s_{fast} are non-decreasing from time t_c on. This means that both functions will in fact be non-increasing if no additional jobs arrive. We start with s_{fast}. For $t > t_c$, s_{fast} is $s_{A,\mathcal{J}_{fast}}(t)$. $s_{A,\mathcal{J}_{fast}}(t)$ is the speed that algorithm A would run if no further jobs besides those in $\mathcal{J}_{fast}$ arrive. Since A is a monotonic algorithm, it will not increase its speed unless a new job arrives. Therefore, $s_{fast}(t)$ is non-increasing. Now for s_{slow}. RESET-S-SLOW is the procedure in which s_{slow} is determined. The only time it is set to be increasing is when the system is idle and there is a wake-up time for some point in the future. In this case, s_{slow} will increase from 0 to s_{crit} at some time t_w in the future and will be non-increasing for any t after t_w. If the system is in slow mode the next time it wakes up, this means that it is waking up at the current wake-up time t_w and s_{slow} will be non-increasing

for any $t > t_w$. If it wakes up in fast mode, then RESET-S-SLOW is called and s_{slow} is reset to be non-increasing.

Now to prove the lemma, suppose for a contradiction that at some time t, the system goes to some speed s which is less than s_{crit} and more than 0. It must be the case that all pending jobs can be completed at a speed of s or less at time t since the algorithm has a current plan for completing them without getting faster than s. This follows from the fact that we have just proved that the projected speed of the system is non-increasing. However, this also implies that the system would have transitioned to slow mode at time t and the speed would be reset to at least s_{crit}. □

Let δ_P be an indicator function for when the schedule is active under Procrastinator: $\delta_P(t) = 1$ if $s_P(t) > 0$ and 0 otherwise.

Lemma 14 $\int_{t_0}^{t_1} P(s_{crit})\delta_P(t)dt \leq \texttt{active}(\mathcal{S}_{OPT})$.

Proof. We will prove the lemma for each job j. Lemma 2 tells us that we can assume that the optimal schedule runs job j at a uniform speed s_j. The optimal algorithm spends a total time of R_j/s_j on job j for a total energy expenditure of $P(s_j)R_j/s_j$. Now consider the intervals of time in which Procrastinator works on job j. Since Procrastinator runs at a speed that is at least s_{crit}, the total lengths of these intervals is no greater than R_j/s_{crit}. Since s_{crit} is the value for s that minimizes $P(s)/s$, we know that $P(s_{crit})R_j/s_{crit} \leq P(s_j)R_j/s_j$. Thus, we have that

$$\int_{t_0}^{t_1} P(s_{crit})\delta_P(t)dt \leq \sum_{j\in\mathcal{J}} \frac{P(s_{crit})R_j}{s_{crit}} \leq \sum_{j\in\mathcal{J}} \frac{P(s_j)R_j}{s_j} = \texttt{active}(\mathcal{S}_{OPT}).$$

□

Lemma 15 *Assume that Procrastinator uses an algorithm for DSS-NS that is additive, monotonic and c_1-competitive. Then,* $\int_{t_0}^{t_1}[P(s_P(t) - s_{crit}) - P(0)]\delta_P(t)dt \leq c_1\texttt{active}(\mathcal{S}_{OPT})$.

Proof. Fix an input $\mathcal{J}$. Consider the problem for DSS-S with the power function $P(s) - P(0)$. Since this function uses no energy while the system is idle, the optimal schedule for this power function will be the same as the optimal schedule for DSS-NS. Let $\mathcal{S}_{OPT-NS}$ be the optimal schedule for DSS-NS and let $\texttt{cost}(\mathcal{S}_{OPT-NS})$ denote the cost of this schedule for power function $P(s) - P(0)$ on input $\mathcal{J}$. We know that $\texttt{cost}(\mathcal{S}_{OPT-NS}) \leq \texttt{active}(\mathcal{S}_{OPT})$ since $cals_{OPT-NS}$ can always schedule its jobs exactly like OPT. Furthermore, OPT has the disadvantage that it has to pay an additional $P(0)$ just to keep the system on.

The cost for algorithm A using power function $P(s) - P(0)$ is $\int_{t_0}^{t_1}[P(s_{A,\mathcal{J}}(t)) - P(0)]\delta_P(t)dt$. From the competitiveness of A, we know that

$$\int_{t_0}^{t_1}[P(s_{A,\mathcal{J}}(t)) - P(0)]\delta_P(t)dt \leq c_1\texttt{cost}(\mathcal{S}_{OPT-NS}) \leq c_1\texttt{active}(\mathcal{S}_{OPT}).$$

Lemma 12 says that

$$\int_{t_0}^{t_1} P(s_{fast})\delta_P(t)dt \leq \int_{t_0}^{t_1} P(s_{A,\mathcal{J}}(t))\delta_P(t)dt.$$

Thus, we have that

$$\int_{t_0}^{t_1} [P(s_{fast}(t)) - P(0)]\delta_P(t)dt \leq c_1 \texttt{cost}(\mathcal{S}_{OPT\_NS}) \leq c_1 \texttt{active}(\mathcal{S}_{OPT}).$$

We know that $s_P(t) = s_{fast}(t) + s_{slow}(t)$ for all t and $s_{slow}(t) \leq s_{crit}$ for any t. This means that for any t,

$$s_P(t) - s_{crit} \leq s_P(t) - s_{slow}(t) = s_{fast}(t).$$

Putting this all together, we get that

$$\int_{t_0}^{t_1} [P(s_P(t) - s_{crit}) - P(0)]dt \leq c_1 \texttt{active}(\mathcal{S}_{OPT}).$$

□

Lemma 16 $\texttt{idle}(\mathcal{S}_P) \leq 2\texttt{on}(\mathcal{S}_{OPT}) + 4\texttt{sleep}(\mathcal{S}_{OPT})$.

Proof. Consider the algorithm which we will call P-OPT (for Procrastinator-Optimal) which has the same set of active and idle periods as Procrastinator but is told in advance the length of each idle period. Such an algorithm can make the optimal decision as to whether or not to transition to the *sleep* state at the beginning of an idle period. Using Lemma 11 instead of Lemma 8, and an identical argument to that used in Lemma 10, we get that $\texttt{idle}(\mathcal{S}_{P\_OPT}) \leq \texttt{on}(\mathcal{S}_{OPT}) + 2\texttt{sleep}(\mathcal{S}_{OPT})$.

Since Procrastinator uses the algorithm which shuts down as soon as the cost of staying active equals the cost of powering up, we know that for any idle period, the cost of that period for Procrastinator is at most twice the cost for that period to P-OPT. Thus, we have that $2\texttt{idle}(\mathcal{S}_{P\_OPT}) \geq \texttt{idle}(\mathcal{S}_P)$. □

Theorem 17 *Assume that $P(s)$ is a convex function. Let c_1 be the competitive ratio for A, an additive algorithm for the DSS-NS problem. Let $f(x) = P(x) - P(0)$. Let c_2 be such that for all $x, y > 0$, $f(x + y) \leq c_2(f(x) + f(y))$. The competitive ratio of Procrastinator is at most* $\max\{c_2c_1 + c_2 + 2, 4\}$.

Proof. Fix an input sequence $\mathcal{J}$. We will refer to the schedule produced by Procrastinator (resp. Optimal, Left-To-Right, A) by $\mathcal{S}_P$ (resp. $\mathcal{S}_{OPT}$, $\mathcal{S}_{LTR}$, $\mathcal{S}_A$). Let $s_P(t)$ denote the speed of the system as a function of time under Procrastinator's schedule. Let $s_{fast}(t)$ and $s_{slow}(t)$ be as defined in the algorithm description for PROCRASTINATOR.

We first address the energy spent by Procrastinator while it is active:

$$\begin{aligned}
\texttt{active}(\mathcal{S}_P) &= \int_{t_0}^{t_1} P(s_P(t))\delta_P(t)dt \\
&= \int_{t_0}^{t_1} [P(s_P(t)) - P(0)]\delta_P(t)dt + \int_{t_0}^{t_1} P(0)\delta_P(t)dt \\
&= \int_{t_0}^{t_1} f(s_P(t))\delta_P(t)dt + \int_{t_0}^{t_1} P(0)\delta_P(t)dt \\
&\leq \int_{t_0}^{t_1} [c_2 f(s_P(t) - s_{crit}) + c_2 f(s_{crit})]\delta_P(t)dt + \int_{t_0}^{t_1} P(0)\delta_P(t)dt
\end{aligned}$$

$$
\begin{aligned}
&= \int_{t_0}^{t_1} c_2[P(s_P(t) - s_{crit}) - P(0)]\delta_P(t)dt + \int_{t_0}^{t_1} c_2[f(s_{crit}) + P(0)]\delta_P(t)dt \\
&\leq \int_{t_0}^{t_1} c_2[P(s_P(t) - s_{crit}) - P(0)]\delta_P(t)dt + \int_{t_0}^{t_1} c_2 P(s_{crit})\delta_P(t)dt \\
&\leq c_2(c_1+1)\texttt{active}(\mathcal{S}_{OPT})
\end{aligned}
$$

The last inequality uses Lemmas 14 and 15. From Lemma 16, we know that

$$\texttt{idle}(\mathcal{S}_P) \leq 2\texttt{on}(\mathcal{S}_{OPT}) + 4 \cdot \texttt{sleep}(\mathcal{S}_{OPT}).$$

$\texttt{on}(\mathcal{S}_P)$ can be divided into two parts. The first is the cost of keeping the system on while it is active. This part clearly overlaps with $\texttt{active}(\mathcal{S}_{OPT})$. The second part is the cost of keeping the system on while it is idle. This part is included in $\texttt{idle}(\mathcal{S}_{OPT})$ but does not overlap with $\texttt{sleep}(\mathcal{S}_{OPT})$. Combining with the above bound we get that

$$
\begin{aligned}
\texttt{cost}(\mathcal{S}_P) &= \texttt{active}(\mathcal{S}_P) + \texttt{idle}(\mathcal{S}_P) \\
&\leq (c_1c_2 + c_2 + 2)\texttt{active}(\mathcal{S}_{OPT}) + 4 \cdot \texttt{idle}(\mathcal{S}_{OPT}) \\
&\leq \max\{c_1c_2 + c_1 + 2, 4\}\texttt{cost}(\mathcal{S}_{OPT})
\end{aligned}
$$

□

7 Conclusion

This paper has examined the problem of Dynamic Speed Scaling for systems that have capacity to transition to a sleep state when idle. We have developed an offline algorithm whose total cost comes within a factor of two of optimal. We have also given an online algorithm that makes use of an online algorithm for Dynamic Speed Scaling without a sleep state. One of the most important questions that remains open in this model is whether the offline problem is NP-hard. Also the competitive ratio for the online problem is a large constant. The major bottleneck for improving this constant is to devise more competitive algorithms for the Dynamic Speed Scaling without a sleep state.

References

[1] N. Bansal, T. Kimbrel and K. Pruhs "Dynamic speed scaling to manage energy and temperature," in *Symposium on the Foundations of Computing*, 2004

[2] L. Benini, A. Bogliolo and G. De Micheli "A Survey of Design Techniques for System-Level Dynamic Power Management," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 8, No. 3, June 2000.

[3] I. Hong, G. Qu, M. Potkonjak and M.B. Srivastava, "Synthesis techniques for low-power hard real-time systems on variable voltage processors," In the *Proceedings of Real-Time Systems Symposium*, pages 178-187, 1998.

[4] T. Ishihara and H. Yasuura, "Voltage scheduling problems for dynamically variable voltage processors," In the *International Symposium on Low Power Electronics and Design*, pages 179-202, August 1998.

[5] S. Irani and A. Karlin, "Online Computation," from *Approximations for NP-Hard Problems*, ed. Dorit Hochbaum, PWS Publishing Co, 1995.

[6] S. Irani and S. Shukla and R. Gupta, "Competitive analysis of dynamic power management strategies for systems with multiple power saving states," *the ACM Transactions on Embedded Computing Systems*, Vol. 2, Num. 3, pp 325–346, 2003, *http://doi.acm.org/10.1145/860176.860180.*

[7] S. Irani and S. Shukla and R. Gupta, "Online Strategies for Dynamic Power Management in Systems with Multiple Power Saving States," In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, 2002, pp.117, IEEE Computer Society.

[8] A Karlin, M. Manasse, L. McGeoch, and S. Owicki, "Randomized competitive algorithms for non-uniform problems," in *First Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, pp. 301–309.

[9] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran, "An empirical evaluation of virtual circuit holding time policies in ip-over-atm networks," *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 1371–1382, 1995.

[10] Y.-H. Lu, L. Benini and G. De Micheli, "Low-Power Task Scheduling for Multiple Devices," in the *Proceedings of the International Workshop on Hardware/Software Codesign*, 2000, p39-43.

[11] G. Quan and X. Hu, "Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors," in the *Proceedings of the Design Automation Conference*, 2001.

[12] V. Raghunathan, P. Spanos and M. Srivastava. "Adaptive power-fidelity in energy aware wireless embedded systems," In *IEEE Real-Time Systems Symposium*, 2001.

[13] D. Ramanathan, S. Irani, , and R. K. Gupta, "Latency Effects of System Level Power Management Algorithms," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, 2000.

[14] http://www.rockwellscientific.com/hidra/

[15] T. Simunic, "Energy Efficient System Design and Utilization", PhD Thesis, Stanford University, 2001.

[16] http://www2.parc.com/spl/projects/cosense/csp/ slides/Srivastava.pdf

[17] Yao F, Demers A, Shenker S. A scheduling model for reduced CPU energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 374-82.