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 $P(s)$ and $P(s)/s$ are convex. The problem is to schedule arriving jobs in a way that minimizes total energy use and so that each job is completed after its arrival time and before its deadline. Although each problem has been considered separately, this is the first theoretical analysis of systems which can use both mechanisms. We give an offline algorithm which is within a factor of three 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 which 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.

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 jobs can be estimated when they arrive into the system. The goal is to complete all jobs between their arrival 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 arrival.

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 introduces challenges which do not appear in either of the original problems. In the first problem, the length of the idle intervals is given 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 arrival times and deadlines due to the convexity of the power function. By contrast in DSS-S, it may be beneficial to speed up the tasks 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. Below is a brief description of such a system:

- The Rockwell WINS node is mobile sensing/computing node that has onboard environmental sensors. It gathers the data and then sends it

*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.

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. [15, 13]

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 [4]. It is well known that the optimal competitive ratio that can be achieved by any 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 [7]. Irani *et al.* examine the generalization in which there are multiple *sleep* states, each with a different power usage rate and start-up cost [5]. 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 [8, 5]. 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 [1].

The Dynamic Speed Scaling problem without the *sleep* state has been examined by Yao, Demers and Shenkel (although not under that name). 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 more 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 $2^d d^d$, where power usage as a function of speed is a degree-d polynomial. For each job j, let a_j be its arrival time , b_j its deadline and R_j the total amount of work required to complete the job. AVR defines a speed function for j as follows:

$$s_j(t) = \begin{cases} \frac{R_j}{b_j - a_j} & \text{for } a_i \leq t \leq b_i \\ 0 & \text{otherwise} \end{cases}$$

The speed of the system as a function of time $s(t)$ is then $\sum_j s_j(t)$. Jobs are then scheduled according to the Earliest Deadline First (EDF) policy.

Dynamic Speed Scaling is also a well studied problem in the embedded systems literature. (See [10] 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 algoritm 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 [14]. 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 [9]. 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 [1]. Ramanathan *et al.* perform an experimental study of the latency/power tradeoff in Dynamic Power Management [12]. In [5], 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 a 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 [2, 3]. 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 power convex power functions $P(s)$ such that $P(s)/s$ is also a convex function. The convexity of $P(s)$ is a standard assumption in this work. The convexity of $P(s)/s$ is also a reasonable assumption based on analytical models for $P(s)$ [11]. 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 three 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 notion of a *monotonic* online algorithm for DSS-NS which is an algorithm that only increases its speed at the arrival time of a job. This is a reasonable restriction since an online algorithm will likely only plan its speed according to the jobs it already knows about and these are exactly the jobs whose arrival time have already passed. The only known competitive algorithm for DSS-NS (AVR) is monotonic. Now suppose there is a monotonic online algorithm which is c_1-competitive for DSS-NS. 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 $\max\{c_2(c_1 + 1), c_2 + 3, 6\}$. Using the upper bound for AVR given by Yao *et al.* , this yields an upper bound of 24 for quadratic power functions and 540 for cubic power functions. In the latter case, the analysis can be optimized to give an upper bound of 193. It should be noted that the biggest bottleneck for improvement for DSS-S is to come up with better competitive ratios 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 108.

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 an arrival time a_j and a deadline b_j. We will sometimes refer to the interval $[a_j, b_j]$ 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 arrival 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:

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

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

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

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

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 in 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, so we will denote it by α. This is the power rate when the system is idle (i.e. speed is 0) and on.

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

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

Recall that R is the total amount of work required to complete the job. 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 set of mutually disjoint intervals $\mathcal{I}$, we denote the number of intervals in $\mathcal{I}$ by $|\mathcal{I}|$ and the sum of the lengths of the intervals in $\mathcal{I}$ by $||\mathcal{I}||$.

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

5 An Offline Algorithm

If the cost to transition from the *sleep* state to the *on* state were zero, then the optimal speed for all jobs would be the s that minimizes

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

which is the s which satisfies

$$P(s) = sP'(s).$$

We call this speed the *critical speed* and denote it by s_{crit}. If there is no minimum value, s_{crit} can be defined to be 0 or ∞ depending on whether $P(s)/s$ is increasing or decreasing.

Suppose we consider compressing the execution of a task so that we spend x less time running that task. It will require more energy to complete the task since it is being run at a higher power consumption level. However, we potentially save αx since the time saved can be spent in the *sleep* state. If the speed of a job is greater than s_{crit}, then the energy saved is smaller than the additional energy spent so it is not beneficial to compress the task beyond a speed of s_{crit}. On the other hand, if the cost of transitioning back to the active state is not taken into account, it is always beneficial to compress a job which is run slower than s_{crit}.

Consider a system with power function $P(s)$. This function can be used to define a system with a sleep state or without. A set of jobs $\mathcal{J}$ can be considered as an input to both the sleep and the no-sleep version of DSS. The lemma below relates the optimal solutions to these two version of the problem:

LEMMA 5.1. *Given a set of jobs and power function $P(s)$, let $\mathcal{S}_{NS}$ be an optimal schedule for the version of the problem with no sleep state. There is an optimal solution to the version with a sleep state $\mathcal{S}_S$ such that every job in $\mathcal{S}_{NS}$ which runs at a speed s_{crit} or faster is run at the same time and speed as in $\mathcal{S}_S$.*

Proof. Omitted from this version.

At this point, it is useful to discuss the optimal offline algorithm for DSS-NS given by Yao *et al.* in [16]. We will call that algorithm OPTIMAL-SCHEDULE-NO-SLEEP (OSNS). They show that each job in the optimal solution is executed at a uniform speed although not necessarily in a contiguous block of time. Their algorithm decides on the time interval and speed in which each job will be run in decreasing order of speed. Using their algorithm and the lemma above, we can run the OSNS algorithm until a job is found which will be run more slowly than s_{crit}. Let $\mathcal{J}_{fast}$ be the set of jobs which are determined to run at a speed s_{crit} or greater. Let $\mathcal{I}_{fast}$ be the set of intervals during which these jobs are run. Because of Lemma 5.1 above, an optimal offline algorithm can first determine the jobs in $\mathcal{J}_{fast}$. The running times and speeds for these jobs is then determined according to OSNS. This will produce a set of intervals for which the system is already scheduled. We will call these *scheduled* intervals. For the description of the algorithm we will assume that all the jobs in $\mathcal{J}_{fast}$ have been removed from $\mathcal{J}$. The remainder of this section 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 arrival times and deadlines for these remaining jobs so that they do not occur during a scheduled interval. Arrival times will be moved to the end of the scheduled interval and deadlines will be adjusted to the beginning of the scheduled interval.

Now we must decide at what speed and at what time to run the jobs which would run more slowly 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}. Given this decisions, every algorithm 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 arrival 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 $1/\alpha$). 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 rerunning OSNS 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 arrival 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 arrival 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}.

THEOREM 5.1. *If the power function $P(s)$ and $P(s)/s$ are both convex, then the algorithm Left-To-Right achieves an approximation ratio of* 3.

Before proving Theorem 5.1, we prove the following

```
LeftToRight:FindIdleIntervals(J)
(1) if a new job j arrives
(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
(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 wake-up time is reached
(12)    Start working on pending job
        with earliest deadline.
(13)if the beginning of a scheduled
    interval is reached
(14)    Process the jobs from I_fast which
        were scheduled for this interval
(15)if the end of a scheduled interval
    is reached
(16)    Complete lines (7)-(10)
```

useful lemma. In the proof of the 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. $\mathcal{S}_{LTR}$).

Lemma 5.2. *There is no single interval in $\mathcal{P}_{OPT}$ which contains two intervals in $\mathcal{D}_{LTR}$.*

```
SetWakeUpTime()
    Find the largest time t_w such that
    it is feasible to keep the schedule
    determined so far, have the system
    asleep for the interval [t, t_w] and
    complete all jobs in J - J_fast
    by their deadlines at a speed of s_crit
    or less.
    Set the wake-up time to be t_w .
```

Proof. Suppose that there is an interval $I \in \mathcal{P}_{OPT}$ which contains two intervals $A_1, A_2 \in \mathcal{D}_{LTR}$. Suppose without loss of generality that A_1 is to the left of A_2. Consider the first job j which is run after A_1 and before A_2. It must be the case that either j's release time is before I begins or its deadline is after I completes. This is because $\mathcal{S}_{OPT}$ is a feasible schedule and can not be in the sleep state during all of j's execution interval. If j's release time is before I began, then its release time is also before the left endpoint of A_1. This means that it would have been run at the beginning of interval A_1 in $\mathcal{S}_{LTR}$ since the algorithm never idles as long as there are pending jobs in the system. Similarly, if j's deadline is after the end of I, its deadline is also after the end of A_2. This means that Left-To-Right would have waited to run the task so that it completes just before the right endpoint of A_2 due to the fact that it delays becoming active until it is necessary in order to have a feasible schedule. Thus, it is impossible that any job is run after A_1 and before A_2. $\square$

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 in running jobs aside from the α per time unit keeping the system in the *on* state:
$$\texttt{run}(\mathcal{S}) = \int_{t_0}^{t_1} P(s(t)) - \alpha dt.$$
2. The cost to keep the system in the *on* state while the system is active. Let $\delta_s(t) = 1$ if $s(t) > 0$ and 0 otherwise.
$$\texttt{active}(\mathcal{S}) = \int_{t_0}^{t_1} \alpha\delta_s(t)dt.$$
3. 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 $\mathcal{S}$.
$$\texttt{idle}(\mathcal{S}) = \sum_{I \in \mathcal{D}} \min(\alpha|I|, 1).$$
4. The cost to keep the system in the *on* state while the system is on
$$\texttt{on}(\mathcal{S}) = \int_{t_0}^{t_1} \alpha\delta(\phi(t), on\ )dt.$$
5. 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 $\mathcal{I}$.

Fix a problem instance. We will prove the following three lemmas from which Theorem 5.1 follows easily.

LEMMA 5.3. $\texttt{active}(\mathcal{S}_{LTR}) \leq \texttt{active}(\mathcal{S}_{OPT})$.

LEMMA 5.4.

$$\texttt{run}(\mathcal{S}_{LTR}) \leq \texttt{run}(\mathcal{S}_{OPT}) + \texttt{active}(\mathcal{S}_{OPT}).$$

LEMMA 5.5.

$$\texttt{idle}(\mathcal{S}_{LTR}) \leq \texttt{on}(\mathcal{S}_{OPT}) + 3\texttt{sleep}(\mathcal{S}_{OPT}).$$

Proof of Theorem 5.1

$$\begin{aligned}
\texttt{cost}(\mathcal{S}_{LTR}) &= \texttt{active}(\mathcal{S}_{LTR}) + \texttt{run}(\mathcal{S}_{LTR}) \\
&\quad + \texttt{idle}(\mathcal{S}_{LTR}) \\
&\leq \texttt{run}(\mathcal{S}_{OPT}) + 2\texttt{active}(\mathcal{S}_{OPT}) \\
&\quad + \texttt{on}(\mathcal{S}_{OPT}) + 3\texttt{sleep}(\mathcal{S}_{OPT}) \\
&\leq \texttt{run}(\mathcal{S}_{OPT}) + 3\texttt{on}(\mathcal{S}_{OPT}) \\
&\quad + 3\texttt{sleep}(\mathcal{S}_{OPT}) \\
&\leq 3\texttt{cost}(\mathcal{S}_{OPT})
\end{aligned}$$

The first inequality uses the fact that for any schedule $\mathcal{S}$, $\texttt{cost}(\mathcal{S}) = \texttt{active}(\mathcal{S}) + \texttt{run}(\mathcal{S}) + \texttt{idle}(\mathcal{S})$. The next inequality comes from applying Lemmas 5.3, 5.4 and 5.5. The next inequality follows from the fact that for any schedule $\mathcal{S}$, $\texttt{active}(\mathcal{S}) \leq \texttt{on}(\mathcal{S})$. This just follows from the fact that if a system is active then it has to be on. The final inequality follows from the fact that for any schedule $\mathcal{S}$, $\texttt{cost}(\mathcal{S}) = \texttt{on}(\mathcal{S}) + \texttt{run}(\mathcal{S}) + \texttt{sleep}(\mathcal{S})$. □

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

Proof of Lemma 5.3. We must establish that the optimal algorithm will never run any job faster than s_{crit}. At this point, it is necessary to review the optimal offline algorithm OSNS for the no-sleep version of the problem. They define the intensity of an interval $I = [z, z']$ to be $g(I) = \sum R_j/(z' - z)$, where the sum is taken over all jobs j with $[a_j, b_j] \subseteq [z, z']$. They prove that the optimal algorithm is to identify the interval I of maximal intensity called the *critical* interval. The algorithm schedules those jobs whose interval of execution is contained in I during this interval at a speed of $g(I)$. This is feasible using the Earliest-Deadline-First policy or else there must be an interval of greater intensity. The interval I is blacked out so that no other jobs can be run at this time, all scheduled jobs are removed from $\mathcal{J}$ and the algorithm iterates by picking the next interval of maximum intensity.

Recall that we can assume that all the jobs which would run at a greater speed than s_{crit} in the no-sleep version of the problem have been scheduled and removed. Thus, we are only referring to the jobs that would run at a speed of s_{crit} or less in the no-sleep version of the problem. Suppose that the optimal schedule has a job which runs at a speed greater than s_{crit}. We can think of the optimal algorithm as running in two parts: first determine the optimal *sleep* intervals for the system. Then run OSNS with these periods blacked out. Since OSNS decides on the execution time and speed for the jobs in reverse order of speed, we know that the first critical interval chosen has intensity greater than s_{crit}. Let this critical interval be $[a, b]$, where a is the release time of some job and b is the deadline of some job. It must be the case that $[a, b]$ has a non-zero intersection with some sleep interval because these jobs are run at a speed at most s_{crit} in the absence of sleep intervals. Suppose we decrease this sleep interval by ϵ and use this extra time to run one of the jobs. Since it is possible to schedule jobs using EDF at a uniform speed in $[a, b]$, it must be the case that that the interval of execution of some job contains the period of length ϵ that we have just freed up. The new speed of the job will be $s' - \epsilon'$. Call this job j. The algorithm will spend less energy in runnin j but may have to spend some extra energy in keeping the system on for an additional ϵ. This extra energy will be at most $\alpha\epsilon$. In the worst case, the total energy will decrease by at least

$$R_j \left[\frac{P(s')}{s'} - \frac{P(s' - \epsilon')}{s' - \epsilon'} \right].$$

Since $P(s)/s$ is convex and s' is larger than the minimum value, this energy savings will be positive. □

Proof of Lemma 5.4. We can think of the optimal algorithm as running in two parts: first determine the optimal *sleep* intervals for the system. Then run OSNS with these periods blacked out. Since OSNS runs each individual job at a uniform speed (although possibly not in a contiguous interval), we will assume that the optimal algorithm does not vary the speed of the system while it is running a single job. Consider a job j with workload R_j. Suppose that the optimal algorithm runs this job at speed s_j. The power expended while the job is running is $P(s_j)(R_j/s_j)$. The speed at which Left-To-Right runs the job is s_{crit} which minimizes this value. Thus we have that

$$\texttt{run}(\mathcal{S}_{LTR}) = \sum_{j \in \mathcal{J}} [P(s_{crit}) - P(0)] \frac{R_j}{s_{crit}}$$

$$
\begin{aligned}
&\leq \sum_{j\in\mathcal{J}} P(s_{crit})\frac{R_j}{s_{crit}} \\
&\leq \sum_{j\in\mathcal{J}} P(s_j)\frac{R_j}{s_j} \\
&= \texttt{run}(\mathcal{S}_{OPT}) + \texttt{active}(\mathcal{S}_{OPT}).
\end{aligned}
$$

□

Proof of Lemma 5.5. 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 α times its length, the cost of all these intervals is at most $\texttt{on}(\mathcal{S}_{OPT})$.

Next consider the intervals in $\mathcal{D}_{LTR}$ which have a non-zero intersection with some interval in $\mathcal{P}_{OPT}$. By Lemma 5.2, no more than one interval from $\mathcal{D}_{LTR}$ can be contained a single interval from $\mathcal{P}_{OPT}$. Thus, each interval from $\mathcal{P}_{OPT}$ can intersect at most three 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 three times the number of intervals in $\mathcal{P}_{OPT}$ which is exactly $3\texttt{sleep}(\mathcal{S}_{OPT})$. □

6 An Online Algorithm

The online algorithm for DSS-S which we present here makes use of an monotonic online algorithm A for DSS-NS. 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 the property of being monotonic. However, we will phrase our algorithm so that it can make use of any competitive monotone online algorithm for DSS-NS. We will use $s_A(t)$ to denote the speed of the system as a function of time chosen by A on a given input.

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}. It 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 chosen as follows:

$$
s_{fast}(t) = \begin{cases} s_A(t) & \text{when in fast mode} \\ 0 & \text{when in slow mode} \end{cases}
$$

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 becomes active and remains in slow mode.

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

In each case $s_{slow}(t)$ is set as follows. Let R denote the remaining work of all pending jobs in the system.

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

We define the notion of the *excess at time t* to help in determining when the algorithm needs to switch modes. 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}. This can easily be computed by simulating the system at speed s_{crit} until all the deadlines of pending jobs have been reached. 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 stystem can transition to slow mode unless new jobs arrive. This is the smallest value t_s such that

$$
\int_{t_{current}}^{t_s} s_A(t) \geq \quad \text{excess at the current time.}
$$

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 $1/\alpha$. 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 projected version of these functions and then periodically updates it decision. 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)         SETSLOWDOWNTIME()
(4)     if the system is in slow mode
        or not currently running a job
(5)         if pending jobs can be
            completed at rate $s_{crit}$,
(6)             if system is not currently
                running a job,
(7)                 SETWAKEUPTIME()
(8)         else if pending jobs can
            not be completed at rate $s_{crit}$
(9)             Set wake-up time to the
                current time $t$
(10)            Change to fast mode.
(11)            SETSLOWDOWNTIME()
(12)            $s_{fast}(t) = s_A(t)$
                for all $t > t_{current}$.
(13)if the system completes a job
(14)     if there are no pending jobs,
(15)        Set timer to $1/\alpha$.
(16)if wake-up time is reached
(17)    if system is in sleep state
(18)        Transition to on state.
(19)    Start working on pending job
        with earliest deadline.
(20)    Clear timer.
(21)if timer expires,
(22)    Transition to sleep state.
(23)if the slowdown time is reached,
(24)    Transition to slow mode.
(25)    Set $s_{fast}(t) = 0$ for all $t > t_{current}$.
(26)    RESET()
```

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 $\mathcal{S}_P$.

LEMMA 6.1. *There is no single interval in $\mathcal{P}_{OPT}$ which contains two intervals in $\mathcal{D}_P$.*

Proof. Similar to the proof of Lemma 5.2 except for one case. This is the case where job j's deadline is after I_2. If the algorithm is in slow mode when it wakes up and starts work on j, the argument is the same as in Lemma 5.2. The only case that needs to be addressed is if the arrival of job j causes the algorithm to wake-up in fast mode. We will argue that in this case, the

```
SETWAKEUPTIME()
    Find the largest time $t_w$ such that
    it is feasible to have the system
    sleep until time $t_w$ complete all
    pending jobs by their deadlines at a
    speed of $s_{crit}$ or less.
    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
        $\int_{t_{current}}^{t_s} s_A(t) \geq E$.
```

```
RESET()
    Let $R$ be the total amount of work
    left on pending jobs
    Set $s_{slow}(t) = s_{crit}$
        for $t_{current} \leq t \leq t_{current} + R/s_{crit}$
    Set $s_{slow}(t) = 0$ for
        $t > t_{current} + R/s_{crit}$
```

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 $\hat{t} \in [a_j, b_j]$. If there is an idle period anywhere in $[\hat{t}, b_j]$ 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 6.2. $\texttt{active}(\mathcal{S}_P) \leq \texttt{active}(\mathcal{S}_{OPT})$.

Proof. Consider the the schedule $\mathcal{S}_{NS}$ produced by the optimal offline algorithm for the no-sleep version of the problem. Recall that $\mathcal{I}_{fast}$ is defined to be all those intervals in $\mathcal{S}_{NS}$ in which the system is running at a speed of s_{crit} or higher. For an interval $I \in \mathcal{I}_{fast}$, let R_I denote the total work for those jobs whose execution interval is contained in I. That is,

$$R_I = \sum_{j|[a_j,b_j]\subseteq I} R_j.$$

Any algorithm must get at least R_I work done in interval I. As established in Lemma 5.1 $\mathcal{S}_{OPT}$ gets exactly R_I work done in interval I.

During the remaining time, whenever Procrastinator is active, it is running at a speed at least s_{crit}. In the proof of Lemma 5.3, we established that outside intervals in $cali_{fast}$, the optimal never runs faster than s_{crit}. Therefore, Procrastinator spends less time running jobs than the optimal algorithm. □

LEMMA 6.3. $\texttt{idle}(\mathcal{S}_P) \leq 2\texttt{on}(\mathcal{S}_{OPT}) + 6\texttt{sleep}(\mathcal{S}_{OPT})$.

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 6.1 instead of Lemma 5.2, and an identical argument to that used in Lemma 5.5, we get that $\texttt{idle}(\mathcal{S}_{P-OPT}) \leq \texttt{on}(\mathcal{S}_{OPT}) + 3\texttt{sleep}(\mathcal{S}_{OPT})$.

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 6.1. *Assume that $P(s)$ and $P(s)/s$ are convex functions. Let c_1 be the competitive ratio for A, a monotonic 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* $\max\{c_2(c_1+1), c_2+3, 6\}$.

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 above and beyond the α power rate to keep the system on. Since PROCRASTINATOR waits at least as long to wake-up as Left-to-Right and it runs at least as fast as Left-To-Right while it is active, we know that $s_{slow}(t) \leq s_{LTR}(t)$. By definition, we know that $s_{fast}(t) \leq s_A(t)$ for all t.

$$\begin{aligned}
\texttt{run}(\mathcal{S}_P) &= \int_{t_0}^{t_1} f(s(t))dt \\
&\leq \int_{t_0}^{t_1} c_2(f(s_{fast}(t) + f(s_{slow}(t)))dt \\
&\leq c_2 \int_{t_0}^{t_1} f(s_A(t))dt + c_2 \int_{t_0}^{t_1} f(s_{LTR}(t))dt \\
&\leq c_2\texttt{run}(\mathcal{S}_A) + c_2\texttt{run}(\mathcal{S}_{LTR}) \\
&\leq c_1c_2\texttt{run}(\mathcal{S}_{OPT}) \\
&\quad + c_2(\texttt{run}(\mathcal{S}_{OPT}) + \texttt{active}(\mathcal{S}_{OPT}))
\end{aligned}$$

Now we turn to the energy spent in keeping the system on and in waking it up after sleep intervals. Using Lemmas 6.2 and 6.3,

$$\begin{aligned}
& \texttt{active}(\mathcal{S}_P) + \texttt{idle}(\mathcal{S}_P) \\
\leq\ & \texttt{on}(\mathcal{S}_{OPT}) + 2\texttt{active}(\mathcal{S}_{OPT}) + 6\texttt{sleep}(\mathcal{S}_{OPT})) \\
\leq\ & 3\texttt{on}(\mathcal{S}_{OPT}) + 6\texttt{sleep}(\mathcal{S}_{OPT})
\end{aligned}$$

Combining with the above bound we get that

$$\begin{aligned}
\texttt{cost}(\mathcal{S}_P) &= \texttt{run}(\mathcal{S}_P) + \texttt{active}(\mathcal{S}_P) + \texttt{idle}(\mathcal{S}_P) \\
&\leq c_1(c_2+1)\texttt{run}(\mathcal{S}_{OPT}) \\
&\quad + (c_2+3)\texttt{on}(\mathcal{S}_{OPT}) + 6\texttt{sleep}(\mathcal{S}_{OPT}) \\
&\leq \max\{c_1(c_2+1), c_2+3, 6\}\texttt{cost}(\mathcal{S}_{OPT})
\end{aligned}$$

□

References

[1] 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.

[2] 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.

[3] 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.

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

[5] S. Irani and S. Shukla and R. Gupta, "Competitive analysis of dynamic power management strategies for systems with multiple power saving states," in *Proceedings of the Design Automation and Test Europe Conference*, 2002.

[6] S. Irani and S. Shukla and R. Gupta, "Online Strategies for Dynamic Power Management in Systems with Multiple Power Saving States," submitted for publication.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.