Dynamic Real-Time Scheduling for Energy Conservation in I/O Devices

Rohini Krishnapura and *Steve Goddard*
Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588-0115
{*rohini, goddard*}*@cse.unl.edu*

Abstract

In real-time systems, Dynamic Power Management (DPM) techniques have traditionally centered on the CPU with less focus given to I/O. However, I/O-based DPM techniques have been popularly researched in non-real-time systems. These techniques focus on switching I/O devices to low power states based on some policy. These methods, however, are not applicable to real-time environments because of the non-deterministic nature of the policies. Recently, scheduling techniques to reduce power consumption of I/O devices in real-time systems have emerged. In this paper, we propose an online task scheduling algorithm, Slack Utilization for Reduced Energy (SURE), which utilizes slack in periodic task systems to reduce power consumption in I/O devices.

1 Introduction

Power conservation in embedded systems is traditionally implemented via efficient power management. Dynamic power management (DPM) techniques are those that are applied at run-time based on workload variation [3]. DPM techniques can be classified as CPU-based or I/O based. An example of CPU-based DPM is Dynamic Voltage Scaling (DVS) wherein the operating voltage of the CPU is varied to save energy. I/O based DPM techniques focus on switching I/O devices into low power states based on predictive, stochastic or timeout policies [4].

In real-time systems, dynamic power management has centered mainly on the CPU with little focus given to I/O. I/O-based DPM techniques used for non-real-time systems cannot be used for real-time systems because of their non-deterministic nature. For example, probabilistic power-saving policies for shutting down I/O devices cannot be implemented in hard real-time systems, as jobs are not guaranteed to meet deadlines.

Recently, a few I/O based DPM techniques for real-time systems, have emerged. A DPM algorithm, Low Energy Device Scheduler (LEDES), for hard real-time systems is presented in [5]. LEDES generates a sequence of sleep/working states for each device. This sequence is interpreted as a *device schedule*. This device schedule is generated online using a per-task device-usage list and by looking ahead a limited number of entries in a task schedule such that the energy consumed by the devices is minimized and no task misses its deadline. LEDES is similar to SURE in that devices with two power states are considered. The LEDES algorithm was extended to work with I/O devices with multiple power states in [6]. Multi-state Constrained Low Energy Scheduler (MUSCLES) generates a similar sequence of power states for devices while guaranteeing that real-time constraints are not violated. However both LEDES and MUSCLES use a pre-determined task schedule to dynamically generate the sequence of states for each device such that the total energy consumed by the devices is minimized. This is different from SURE in that both LEDES and MUSCLES do not reorder jobs either online or offline to generate a task schedule that reduces energy consumption in devices.

The pruning-based scheduling algorithm, Energy-optimal Device Scheduler (EDS), is different from LEDES in that jobs are rearranged to find the minimum energy task schedule [7]. In this respect, EDS is more similar to the work done here. EDS generates a schedule tree by selectively pruning the branches of the tree. If a resulting schedule along a branch results in a missed deadline, this infeasible schedule is removed from the tree. In addition, if a feasible schedule along a branch is determined to consume higher energy than an alternative feasible schedule, the branch leading to this schedule is pruned. The algorithm we present is different from EDS in that EDS is an offline algorithm, with schedules computed statically, whereas SURE is an online algorithm that arranges jobs at runtime to reduce energy.

The rest of this paper is organized as follows. Section 2 describes the problem and the motivation for the algorithm. We present the energy conserving algorithm in Section 3. and conclude in Section 4.

2 Problem Description

The task model that we adopt is the periodic task model as proposed by Liu and Layland [2], with deadlines equal to the periods. A periodic task set is defined with a release time, period, worst-case execution time and a deadline. Suppose that the set of devices required by each task during its execution is specified along with the above parameters. The generalized problem can now be stated as follows.

Given a periodic task set, $\{T_1, T_2, ...T_n\}$, $T_i = (\phi_i, p_i, e_i, d_i, \Lambda_i)$ *where,*

ϕ_i *is the release time or phase,*
p_i *is the period,*
e_i *is the worst case execution time,*
d_i *is the deadline and*
$\Lambda_i = \{\lambda_1, \lambda_2, ...\lambda_m\}$ *is the device requirement specification for the task* T_i*,*
is there a schedule which meets all deadlines and also reduces the energy consumed by each device λ_j*?*

Modern I/O devices have at least two power states: *idle* and *active*. The rate at which energy is consumed is different in each state with less power being used at the *idle* state. Thus to save energy, devices can be switched to the *idle* state, when it is not in use. In a real-time system, in order to guarantee that jobs will meet their deadlines, a device cannot be made *idle* without knowing when it will be requested by a job. But, the precise time at which an application requests the operating system for a device is usually not known. Predictive algorithms try to forecast the rate at which requests come in or make an estimate based on past requests. However, even without knowing the exact time at which requests are made, we can safely assume that devices are requested within the time of execution of the process or job making the request. We can also assume that in the absence of DMA or other such mechanisms, a device will be used within the execution time of a job. Thus, given these assumptions, we can determine the upper bound on the utilization of a device λ_i. We define this upper bound as the **Device Utilization Factor** U_{λ_i}. For the periodic task model as specified earlier in this section,

$$U_{\lambda_i} = \sum_{\forall T_j, \{\lambda_i\} \subseteq \Lambda_j} (e_j/p_j)$$

Thus, the Device Utilization Factor of a device is the sum of the CPU utilization of the tasks using the device. In a hyperperiod, the total time that a device λ_i will be used is $U_{\lambda_i} \cdot H$. This means that the device is not in use for at least $(1-U_{\lambda_i}) \cdot H$ time units.

For the periodic task model, consider the energy consumed over one hyperperiod. If the device remained *active* over the entire hyperperiod, the total energy consumed would be

$$E_{orig} = P_{active} \cdot H$$

where P_{active} is the rate at which energy is consumed when the device is *active*. Since the device is not in use for at least $(1 - U_{\lambda_i})H$, the device does not need to be *active* for the entire hyperperiod. However, significant cost is incurred when an I/O device switches or transitions from one power state to another. This cost is high in terms of both time and energy. The total energy consumed by a device λ_i in the hyperperiod H, is given by,

$$E_{\lambda_i} = E_{active} + E_{idle} + E_{sw} \quad (1)$$

where, E_{active} is the energy consumed when λ_i is in the *active* state and E_{idle} is the energy consumed by λ_i when it is in *idle* state and E_{sw} is the energy consumed when λ_i is in transition states.

$$E_{active} = P_{active} \cdot U_{\lambda_i} \cdot H$$

For simplification, let the time taken to switch from *active* to *idle* and vice-versa be the same. Let us call this switch time t_{sw}. In addition, let the power consumed during both transitions be the same. Let this power be P_{sw}. Then,

$$E_{sw} = \sigma_i \cdot P_{sw} \cdot t_{sw}$$

where σ_i is the total number of device state switches in a hyperperiod. So, the actual time the device is in the *idle* state is $[(1 - U_{\lambda_i})H - \sigma_i t_{sw}]$. Thus,

$$E_{idle} = P_{idle}[(1 - U_{\lambda_i})H - \sigma_i \cdot t_{sw}]$$

where P_{idle} is the rate at which energy is consumed when the device is *idle*. Substituting for E_{active}, E_{idle} and E_{sw} in Equation (1), the total energy consumed by λ_i in a hyperperiod is,

$$E_{\lambda_i} = [P_{active} U_{\lambda_i} \cdot H] + [P_{idle}(1 - U_{\lambda_i})H - P_{idle}\sigma_i t_{sw}] + [\sigma_i \cdot P_{sw} t_{sw}]$$

The energy savings incurred if the device is made *idle* whenever it is not in use, is given by,

$$\begin{aligned} E_s(\lambda_i) &= E_{orig} - E_{\lambda_i} \\ &= P_{active} \cdot H - [P_{active} \cdot U_{\lambda_i} \cdot H \\ &+ P_{idle}(1 - U_{\lambda_i})H - \sigma_i P_{idle} t_{sw} + \sigma_i \cdot P_{sw} \cdot t_{sw}] \\ &= P_{active}(1 - U_{\lambda_i})H - P_{idle}(1 - U_{\lambda_i})H \\ &- \sigma_i t_{sw}(P_{sw} - P_{idle}) \\ &= (P_{active} - P_{idle})(1 - U_{\lambda_i})H - \sigma_i t_{sw}(P_{sw} - P_{idle}) \end{aligned}$$

Thus, to increase energy savings, the time for which the device λ_i is *idle* must be increased whereas the total number of power state transitions (σ_i) must be decreased. The online energy conserving algorithm proposed in the next section does this by allowing jobs that require the same device to run in succession. Thus, if a device is in the *active* state, ready jobs requiring the device are executed in succession such that only few device state changes occur. If a device is in the *idle* state, the execution of jobs is delayed as much as possible so that the jobs do not miss their deadlines but also allows the device to be in the *idle* state for a longer duration. This results in combining small and scattered device idle times to generate device idle times of longer duration. In addition, CPU idle times are combined to produce longer intervals of CPU idle time. During these CPU idle intervals, the entire system can be switched to a low power mode to save additional power.

3 Algorithm Description

In a real-time system, there is seldom any gain in finishing jobs early. For example, in hard real-time systems, as long as deadlines are met, there is no incentive for an early response time. At any instant t, the amount of time job execution can be delayed without resulting in any job in the task set to miss its deadline is called the system slack at time t denoted as $\Omega(t)$.

If an I/O device is *idle* and a job requiring that device is released, then if there is system slack at that time, the device is allowed to stay idle till system slack becomes zero. At this point, the job has to be executed to meet its deadline. Similarly, suppose a device was *active*, and the job with the nearest deadline, i.e., the highest EDF priority job, did not require the device (henceforth, we use the term priority to mean the priority assigned by the EDF scheduling algorithm). At this time, there could be another lower priority job requiring the same device. If there is slack in the system, the higher priority job could be deferred and the lower priority job is executed till there is no more slack in the system. At this point, the higher priority job has to execute to meet its deadline.

The heuristic here is that a device state change from *active* to *idle* or vice-versa is delayed as much as possible. The overall result of the algorithm is that smaller chunks of device idle times and usage times are grouped together. This results in reducing the total number of state transitions in the hyperperiod. The algorithm is presented in Figure 1 and Figure 2.

```
scheduler( ):
Initialize at t= 0: {
    noSlack ← false ;
    J_curr ← φ;
    B_curr ← 0;
    computeSlack ← false;
    devShare ← φ;
    return;
}
If (t: instance when job is released) {
   If (J_curr == φ) // the CPU is idle
      computeSlack ← true
   else
      computeSlack ← false
   If (noSlack) { // the system has no slack
      do_EDF();
      return;
   }
}
If (t: instance when job finishes its execution budget) {
   If (job queue is empty) {
      J_curr ← φ; // make CPU idle
      B_curr ← 0;
      return;
   }
   computeSlack ← true; // need to recompute slack
}
If (computeSlack) {
   Compute Ω(t);
   If (Ω(t) > 0)
      do_SURE ();
   else
      do_EDF();
}
```

Figure 1. The SURE Scheduler.

The algorithm combines slack utilization with EDF to produce an energy conserving schedule. At $t = 0$, all devices are in the *idle* state. J_{curr} corresponds to the currently executing job and is initialized to ϕ. Each scheduled job is given an execution budget before execution. The execution budget of J_{curr} is tracked with the variable B_{curr}, which is initialized to zero. The scheduler is invoked when a job is released or when a job completes or finishes its execution budget. The boolean variable *noSlack* is used to indicate that there is no slack in the system and is initially made *false*. The boolean variable *computeSlack* determines when to compute the system slack and is made *false* at $t = 0$.

```
do_EDF(){
   If (J_curr ≠ J_high) {
      devShare ← {Λ_T(curr) ∩ Λ_T(high)}; // devShare is the
set of devices shared by J_curr and J_sh
      Make devices in {Λ_T(curr) − devShare} idle; // the set
of active devices not required by J_sh is made idle
      Make devices in {Λ_T(high) − devShare} active; // the
set of idle devices required by J_sh are made active
      J_curr ← J_high; // execute the highest priority job
      B_curr ← e_high;
      noSlack ← true;
   }
   return;
}
do_SURE () {
   If (J_curr ≠ φ) {
      // Determine the job J_sh which shares the maximum number of devices with J_curr
      devShare ← {Λ_T(curr) ∩ Λ_T(sh)};
      If (|devShare| == 0) { // no job share any device with
J_curr
         Make devices in {Λ_T(curr)} idle; // Make all devices
used by J_curr idle
         J_curr ← φ; // Make CPU idle
      }
      else {
      Make devices in {Λ_T(curr) − devShare} idle;
      Make devices in {Λ_T(sh) − devShare} active;
      J_curr ← J_sh;
      }
   }
   B_curr ← Ω(t);
   noSlack ← false;
   return;
}
```

Figure 2. do_EDF() and do_SURE () procedures.

At $t = 0$, when a job is released, since all devices are in the *idle* state, if there is slack in the system, the execution of jobs is deferred to keep the devices in the *idle* state as long as possible. Hence, *computeSlack* is made *true* and slack is computed. If system slack is greater than zero, then *do_SURE()* is invoked. Here, J_{curr} remains equal to ϕ and B_{curr} is made equal to $\Omega(t)$. When this execution budget expires, the scheduler is invoked again. Now, since all the system slack has been consumed, *do_EDF()* is invoked and the job with the nearest deadline or the highest priority job, J_{high} is executed. The devices required by this job, as specified by $\Lambda_{T(high)}$ will all be changed to the *active* state. $T(high)$ refers to the task of the highest priority job J_{high}.

J_{high} will execute till it completes, at which point the

scheduler is invoked again. Whenever a job completes or finishes its budget, slack is computed and *do_SURE ()* is invoked. Now, if there is another job, J_{sh}, which shares the maximum number of devices with the previously executed job, then J_{sh} is executed immediately. *devShare* denotes the set of devices shared by J_{curr} and J_{sh}. Some *active* devices which are not needed by J_{sh} will be made *idle* and other *idle* devices required by J_{sh} will be made *active*. J_d is executed with a budget equal to the system slack at that time. However, if none of the ready jobs require any of the *active* devices, then the CPU is idled for a time equal to the system slack.

After J_d finishes its execution budget, the SURE scheduler is invoked again. Now, if there are no more jobs to execute, the CPU is idled and B_{curr} is made zero. All *active* devices will be made *idle*. Again when a job is released, its execution is delayed as much as possible till there is no more slack.

Example

Consider the task set $\{T_1, T_2\}$, $T_1 = (0, 2, 1, 2, \{\lambda\})$, $T_2 = (0, 5, 1, 5, \{\lambda\})$ where the deadline is equal to the period and release time is 0. Both tasks require device λ. The hyperperiod is 10. Fig. 3 shows both the EDF schedule and the SURE schedule for the task set where the device λ is idle whenever the CPU is idle (since all tasks use device λ). At $t = 0$, the device λ is *idle*. With EDF the idle times in a hyperperiod are $\{1, 1, 1\}$ time units and the total number of switches is 6. Since the task sets have zero phase, the EDF schedule will be the same in all subsequent hyperperiods. With the SURE

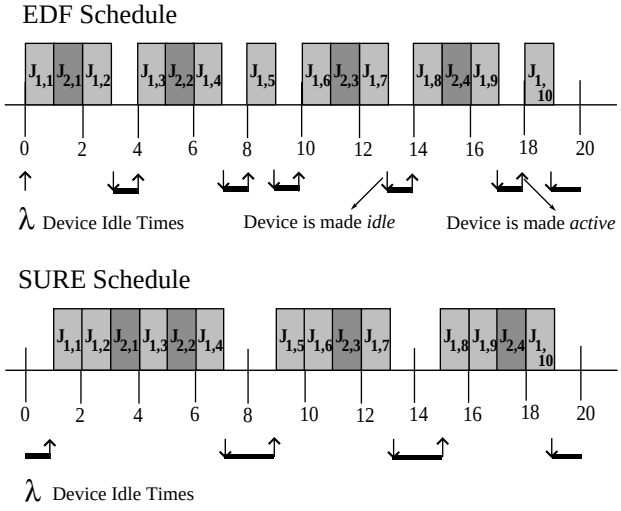

Figure 3. Schedule for $\{T_1, T_2\}$, $T_1 = (0, 2, 1, 2, \{\lambda\})$, $T_2 = (0, 5, 1, 5, \{\lambda\})$

schedule, at $t = 0$, the device λ is idled for 1 time unit. At $t = 1$, the highest EDF priority job is executed. Subsequent eligible jobs which require λ are all executed in succession. At $t = 7$, the ready job queue becomes empty and the CPU is idled. The device is changed to the *idle* state. At $t = 8$, $J_{1,5}$ is released. But, since λ is already in the *idle* state, execution of this job is delayed as much as possible. The device remains in the *idle* state till $t = 9$. The remaining jobs all execute in time and complete within their deadline. With slack utilization, the device idle time is 1 time unit in the beginning of the first hyperperiod. Subsequently, longer idle time of 2 time units are obtained. The total number of switches in a hyperperiod is reduced to 3 and the total idle time, of course, remains constant.

Temporal Correctness

Theorem: *A set of synchronous periodic tasks* $T = \{T_1, T_2, T_3, \ldots T_n\}$*, with deadlines equal to their periods and pre-emption allowed, can be feasibly scheduled on a single processor with SURE if and only if* $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$.

Proof: Proof of this theorem is provided in [1]. □

4 Conclusions and work-in-progress

The SURE, online real-time scheduling algorithm was presented in this paper. We note that there are remain many implementation details to be addressed. For instance, when we use the network card as the I/O device, the non-deterministic nature of ethernet must be taken into account. In addition, the overhead involved in switching the device states must also be addressed in the implementation. At present,we are implementing the algorithm in the microC OS-II real-time OS in the Rabbit 3000 microprocessor and are currently carrying out a preliminary evaluation of the algorithm.

References

[1] Krishnapura R. and Goddard S., A Dynamic Real-time Scheduling Algorithm for Reduced Energy Consumption in I/O Devices, TR-UNL-CSE-2003-10, Sept 2003.

[2] Liu J. W. S., Real-time Systems. Prentice-Hall. 2000.

[3] Lu Y. H., Benini L. and Micheli G., Operating-System Directed Power Reduction, *International Symposium on Low Power Electronics and Design*, 2000.

[4] Lu Y. H., Benini L. and Micheli G., Power-Aware Operating Systems for Interactive Systems, *IEEE Transactions on Very Large Scale Integration Systems*, 2000

[5] Swaminathan V. and Chakrabarty K., Dynamic I/O power management in real-time systems, *Proc. International Conference on Information Fusion (FUSION)*, 2002.

[6] Swaminathan V. and Chakrabarty K., Energy-conscious, deterministic I/O device scheduling in hard real-time systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, $July$, 2003.

[7] Swaminathan V. and Chakrabarty K., Pruning-based energy-optimal device scheduling in hard real-time systems, *Proc. International Symposium on Hardware/Software Co-Design*, 2002.