Online Energy-Aware I/O Device Scheduling for Hard Real-Time Systems*

Hui Cheng, Steve Goddard
Department of Computer Science and Engineering
University of Nebraska — Lincoln
Lincoln, NE 68588-0115
{hcheng, goddard}@cse.unl.edu

Abstract

Much research has focused on power conservation for the processor, while power conservation for I/O devices has received little attention. In this paper, we analyze the problem of online energy-aware I/O scheduling for hard real-time systems based on the preemptive periodic task model. We propose an online energy-aware I/O device scheduling algorithm: Energy-efficient Device Scheduling (EEDS). The EEDS algorithm utilizes device slack to perform device power state transitions to save energy, without jeopardizing temporal correctness. An evaluation of the approach shows that it yields significant energy savings with respect to no Dynamic Power Management (DPM) techniques.

1. Introduction

Power management is important to extend the limited battery life of portable embedded systems. In the past decade, much research work has been conducted on low-power design methodologies for real-time embedded systems. For hard real-time systems, the research has focused primarily on reducing the power consumption of the processor. The research on power conservation technologies for I/O devices, though important, has received little attention.

In practice, embedded systems are usually intended for a specific application. Such systems tend to be I/O intensive, and many of them require real-time guarantees during operation [7]. Therefore, aggressive energy conservation techniques are needed to save energy in I/O devices for real-time embedded systems.

Power management techniques can be viewed as being static or dynamic. Static power management techniques are carried out at design time whereas Dynamic Power Management (DPM) techniques are applied at run-time based on workload variation. Although static techniques can save significant energy, they are relatively inflexible to adapt to changes in the operating environment. DPM at the operating system (OS) level, on the other hand, has gained importance due to its flexibility and ease of use [7]. The focus of this paper, therefore, is to investigate I/O-based DPM techniques for real-time embedded systems.

Most I/O-based DPM techniques save energy by *resource shutdown*. That is, identifying time intervals where I/O devices are not being used and switching these devices to low-power modes during these periods. Saving energy for I/O devices in hard real-time systems incurs the following challenges: (1) Power state transitions incur significant time and power penalties for I/O devices. Therefore, devices should be put in low-power states as long as possible but still guarantee system temporal correctness; (2) Each task may use multiple devices. Power state transitions should be made per-device. Some procrastination scheduling techniques [2, 4], used in CPU-based DPM, are not suitable for I/O devices, because these techniques consider the CPU as the only shared device.

There have been efforts [5, 6, 7, 8] in developing energy-efficient device scheduling algorithms that reduce I/O device energy consumption for real-time systems. Among them, [5, 6, 7] can only support non-preemptive task scheduling. The only known published energy-aware algorithm for preemptive schedules, Maximum Device Overlap (MDO), is an offline method proposed by the same authors in [8]. A deficiency of the offline method is that it is hard to adapt to changes in the operating environment, such as dynamic task join/leave or early job completion.

In this paper, we propose an online energy-aware I/O device scheduling algorithm, Energy-efficient Device Scheduling (EEDS), for hard real-time systems based on the preemptive periodic task model. This algorithm uses preemptive Earliest Deadline First (EDF) [3] to schedule jobs. As with [8], EEDS performs inter-task device scheduling rather than intra-task device scheduling. That is, the scheduler does not put devices to sleep while tasks that require them are being executed, even though there are no pending I/O requests at that time. Intra-task device scheduling is generally not advisable for hard real-time systems. To the best of our

*Supported, in part, by grants from the National Science Foundation (CNS-0409382, and CCF-0429149).

knowledge, EEDS is the first online energy-efficient device scheduling algorithm for hard real-time systems.

The rest of this paper is organized as follows. The problem of energy-aware I/O device scheduling is analyzed in Section 2. Section 3 describes the proposed algorithms. Section 4 describes how we evaluated our system and presents the results. Section 5 presents our conclusions and describes future work.

2. Problem description

Modern I/O devices usually have at least two power modes: *active* and *sleep*. I/O operations can be only performed on a device in active state, and a transition delay is incurred to switch a device between power modes. In a real-time system, in order to guarantee that jobs will meet their deadlines, a device cannot be put in sleep mode 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. 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 job making the request. As a result, our method is based on inter-task device scheduling.

Given a periodic task set with deadlines equal to periods, $\tau = \{T_1, T_2, ...T_n\}$, let task T_i be specified by the three tuple $(P(T_i), W(T_i), Dev(T_i))$ where, $P(T_i)$ is the period, $W(T_i)$ is the Worst Case Execution Time (WCET), $Dev(T_i) = \{\lambda_1, \lambda_2, ..., \lambda_m\}$ is the set of required devices for the task T_i. Let i be the index of T_i. We refer to the j^{th} job of a task T_i as $J_{i,j}$. The release time of $J_{i,j}$ is denoted by $R(J_{i,j})$. We let $Dev(J_{i,j})$ denote the set of devices that are required by $J_{i,j}$. Throughout this paper, we have $Dev(J_{i,j})$=$Dev(T_i)$.

The priorities of all jobs are based on EDF. For any two jobs, the job with the earlier deadline has a higher priority. If two jobs have equivalent deadlines, the job with the earlier release time has a higher priority. In case that both deadline and release times are equal, the job belonging to the task with a smaller index has the higher priority. In this way, the priority of any job is unique. The priority of a job $J_{i,j}$ is denoted by $Pr(J_{i,j})$.

Associated with a device λ_i are the following parameters: the transition time from the *sleep* state to the *active* state represented by $t_{wu}(\lambda_i)$; the transition time from the *active* state to the *sleep* state represented by $t_{sd}(\lambda_i)$; the energy consumed per unit time in the *active*/*sleep* state represented by $P_a(\lambda_i)$/$P_s(\lambda_i)$ respectively; the energy consumed per unit time during the transition from the *active* state to the *sleep* state represented by $P_{sd}(\lambda_i)$; and the energy consumed per unit time during the transition from the *sleep* state to the *active* state represented by $P_{wu}(\lambda_i)$. We assume that for any device, the state switch can only be performed when the device is in a stable state, *i.e.* the sleep state or the active state.

```
1  Preprocessing:
2    Compute break-even time BE(λ_k) (1 ≤ k ≤ m) for each device.
3  Schedule jobs at time t when a job is put in the ready queue or is
   completed
4  // A job can join the ready queue when all needed devices are active.
5    J_run ← the job with the highest EDF priority in the ready queue.
6    Dispatch J_run;
7  Perform device state transitions at time t when a job is released,
   completed or the timer to reactivate a device is reached.
8    If (t: ∃λ_k, λ_k ∉ Dev(J_run) && λ_k = active
     && DS(λ_k, t) > BE(λ_k))
9      λ_k ← sleep;
10     // Up(λ_k) is the timer set to reactivate λ_k.
11     Up(λ_k) ← t + DS(λ_k, t) − t_wu(λ_k);
12   End If
13   // Device slack may increase; update Up(λ_k) for sleeping devices
     // in this case. see Section 3.1 for detailed discussion.
14   If (t: ∃λ_k, λ_k = sleep && t+DS(λ_k, t)−t_wu(λ_k) > Up(λ_k))
15     Up(λ_k) ← t + DS(λ_k, t) − t_wu(λ_k);
16   End If
17   // Reactivate λ_k when the timer is reached.
18   If (t: ∃λ_k, λ_k = sleep && Up(λ_k) = t)
19     λ_k ← active;
20   End If
21 End
```

Figure 1. The EEDS algorithm.

3. Algorithm

As discussed in Section 2, an energy-aware I/O device scheduler needs to identify and even create idle intervals where I/O devices can be put in the sleep mode while not violating temporal correctness. For example, if all pending jobs requiring device λ_k during time interval $[t, t']$ can start/resume their executions as late as t' without causing any job to miss its deadline, then λ_k can sleep during $[t, t']$ to save energy. Before we explain our approach in more detail, we first introduce several definitions.

Definition 3.1. *Job slack.* The *job slack* of a job $J_{i,j}$ is the available time for $J_{i,j}$ to suspend its execution without causing any job to miss its deadline. The job slack of $J_{i,j}$ at time t is denoted by $JS(J_{i,j}, t)$. The computation of job slack is given in Section 3.1.

Obviously, only unfinished jobs need to be considered. The following definition identifies the current job of a task.

Definition 3.2. *Current job.* Let $CurJob(T_i, t)$ denote the current job of task T_i at time t. Suppose job $J_{i,j}$ is the last released job of task T_i at time t. The current job of T_i is $J_{i,j}$ if $J_{i,j}$ is not finished at or before time t; otherwise the current job of T_i is $J_{i,j+1}$.

With EEDS, each device is associated with a *device slack*, which represents the available time for a device to sleep. The device slack is defined as follows.

Definition 3.3. *Device slack.* The *device slack* is the length of time that a device λ_k can be inactive[1] without causing any

[1] *inactive* means that a device is either in the sleep mode or is in the middle of a power mode transition.

job to miss its deadline. We let $DS(\lambda_k, t)$ denote the device slack for a device λ_k at time t. With the definition of job slack and current job, $DS(\lambda_k, t)$ can be given by

$$DS(\lambda_k, t) = min(JS(CurJob(T_i, t), t)) \quad (1)$$

where T_i is any task that requires λ_k.

Because of the energy penalty associated with the power state transition, a device needs to be put in the sleep mode long enough to save energy. *Break-even time* represents the minimum inactivity time required to compensate for the cost of entering and exiting the idle state. The computation of break-even time can be found in [1]. It is clear that if a device is idle for less than the break-even time, it is not worth performing the state switch. Therefore, our approach makes decisions of device state transition based on the break-even time rather than device state transition delay.

Algorithm EEDS (Figure 1) sketches the general idea of our approach. The scheduler keeps track of device slack for each device. Once a device is not required by the current running job and the device slack is larger than the break-even time, the scheduler puts the device in the sleep mode to save energy. At the same time, a timer is set to reactivate the device in the future. In case that the device slack for a sleeping device is increased, the timer is updated accordingly. We will discuss it at the end of Section 3.1. All released but unfinished jobs are put in the waiting queue if needed devices are not active. Other jobs are put in the ready queue and scheduled according to the EDF algorithm. In case that a new task joins the system during runtime, all sleeping devices are reactivated and the device slack of each device is updated.

3.1. Computing the job slack

The job slack of a job comes from two sources. The first source is the *run-time*. The concept of run-time comes from known techniques [2, 9], denoting the time budget allocated to each job. Generally this time budget is larger than the actual need of the job. Therefore, over-provisioned run-time can be used to prolong job slack. Figure 3(a) shows an example of this kind of job slack. We will discuss it in detail shortly.

The straightforward run-time assigned to a job $J_{i,j}$ is its WCET, *i.e.*, $W(J_{i,j})$. However, since the system utilization U is generally less than 1, the initial run-time can be further extended to $W(J_{i,j})/U$, without overloading the system. The run-time of a job has a priority and a deadline which are set equal to the job's priority and deadline for the purpose of computing job slack. Recall that in Section 2, we described rules to assign each job an unique priority to break the tie when two jobs have the same deadline.

When a job $J_{i,j}$ is released, the associated initial run-time is inserted into a *run-time list* (RT-list), in which run-times are sorted by their priorities with the highest priority

```
1  At any time t:
2    If (t: a new job J_{i,j} arrives)
3      Insert_to_RT(W(J_{i,j})/U, Pr(J_{i,j}));
4    End If
5    rt_0 ← rt_0 − 1;
6    If (rt_0 = 0);
7      Remove_from_RT(rt_0);
8      rt_0 ← the head of RT-list;
9    End If
10 End
```

Figure 2. The algorithm to update run-time list.

run-time at the head of the RT-list. The running job always consumes the run-time from the head of the RT-list when it executes. When the CPU is idle, the run-time is consumed in the same way. If the run-time at the head of the RT-list is depleted, the item is removed and the next run-time becomes the head.

Let the *available run-time* for a job $J_{i,j}$ denote the sum of all higher priority run-times in the RT-list and the run-time associated with $J_{i,j}$ itself. Note that a run-time is inserted into the RT-list only when the associated job is released. Therefore, the run-time associated with $J_{i,j}$ is available for only $J_{i,j}$ before the release of $J_{i,j}$; and is available for all jobs with priorities no higher than $Pr(J_{i,j})$ when $J_{i,j}$ is released.

In Section 3.2, we will prove that a job can suspend its execution as long as the available run-time is larger than its residual execution time, without causing any job to miss its deadline. Therefore, the job slack that comes from the run-time is the available run-time for the job minus the residual execution time. Before proceeding with the discussion, we introduce following notation.

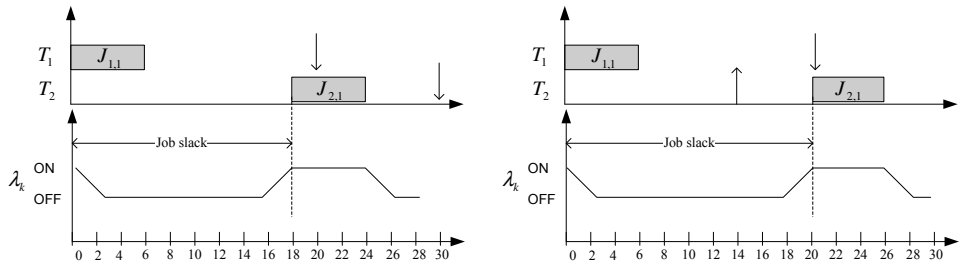
- rt_i: the i^{th} run-time in the RT-list, with rt_0 representing the head of the RT-list.
- $rt(J_{i,j})$: the run-time associated with $J_{i,j}$.
- $Pr(rt_i)$: the priority of the run-time rt_i.
- $R^e(J_{i,j}, t)$: the worst case residual execution time of job $J_{i,j}$.
- $R^r(J_{i,j}, t)$: the *available run-time* for job $J_{i,j}$, which is given by $\sum_{Pr(rt_i) > Pr(J_{i,j})} rt_i + rt(J_{i,j})$.

With this notation, the algorithm to update RT-list is described in Figure 2. The job slack coming from the run-time is given by

$$JS(J_{i,j}, t) = R^r(J_{i,j}, t) - R^e(J_{i,j}, t) \quad (2)$$

The example shown in Figure 3(a) illustrates how run-time contributes to job slack. Both jobs are released at time 0. The initial run-time for both jobs is 12 because $U = 0.5$. The job slack for $J_{2,1}$ is the sum of available run-time minus the residual execution time of $J_{2,1}$, which is $12+12-6 = 18$ according to Equation (2).

However, there is another source of job slack. A job can only start execution when it is released. So if the current



(a) Job slack coming from the run-time (b) Job slack coming from the latest eligible time.

Figure 3. Job slack examples. $T_1 = \{20, 6, \emptyset\}$**;** $T_2 = \{30, 6, \{\lambda_k\}\}$**. That is,** $\lambda_k \in Dev(T_2)$**. (a)** $J_{1,1}$ **and** $J_{2,1}$ **both are released at time** 0**; (b)** $J_{1,1}$ **is released at time** 0 **and** $J_{2,1}$ **is released at time** 14**.**

time t is less than the job release time $R(J_{i,j})$, then the time interval $[t, R(J_{i,j})]$ can be seen as job slack. Furthermore, a job is assigned an initial run-time of $W(J_{i,j})/U$, which will produce at least $W(J_{i,j}) \times (1/U - 1)$ unused run-time (free run-time or slack). Therefore, it is known that a job can start its execution as late as $R(J_{i,j}) + W(J_{i,j}) \times (1/U - 1)$, which is called its *latest eligible time* and is defined as follows

Definition 3.4. *Latest eligible time.* The *latest eligible time* for a job $J_{i,j}$ is given by

$$LT(J_{i,j}) = R(J_{i,j}) + W(J_{i,j}) \times (1/U - 1) \tag{3}$$

A job can become eligible for execution as late as its latest eligible time without causing any job to miss its deadline. The job slack coming from the latest eligible time is given by

$$JS(J_{i,j}, t) = LT(J_{i,j}) - t \tag{4}$$

Finally, considering both Equation (2) and Equation (4), the job slack of a job $J_{i,j}$ is given by

$$JS(J_{i,j}, t) = max(LT(J_{i,j}) - t, R^r(J_{i,j}, t) - R^e(J_{i,j}, t)) \tag{5}$$

Figure 3(b) shows an example of the job slack coming from the latest eligible time. $J_{1,1}$ is released at 0 and $J_{2,1}$ is released at 14. At time 0, the job slack of $J_{2,1}$ coming from run-time is 18 according to Equation (2). However, the job slack coming from its latest eligible time is 20 according to Equation (4). Therefore, the job slack of $J_{2,1}$ is the larger of the two, which is 20.

It can be seen that the job slack can be reduced at most 1 per system unit; and is increased only when a new job is released. According to Equation (1), the device slack can also be increased when a new job is released. Therefore, the timer set to reactivate a sleeping device needs to be updated in this case, as described in Figure 1 (line 13-16).

3.2. Schedulability

Theorem 3.1. *A set of periodic tasks* $T = T_1; T_2; T_3; \ldots T_n$*, with deadlines equal to their periods, can be feasibly scheduled on a single processor with EEDS if and only if*

$$\sum_{i=1}^{n} \frac{W(T_i)}{P(T_i)} \leq 1 \tag{6}$$

However, the following lemmas are required before we can actually prove Theorem 3.1.

Lemma 3.2. *With the EEDS algorithm, the run-time available to a job* $J_{i,j}$ *is depleted at or before its deadline.*

Proof: We first show that any run-time must be depleted at or before its own deadline. Suppose the claim is false. Let t be the first time that a run-time rt_k is not depleted at its deadline. Let t_0 be the last instance before t at which there is no run-time with the priority higher than or equal to $Pr(rt_k)$ in the RT-list. Since there is no run-time before the system start time 0, t_0 is well defined. Recall that the run-time is always consumed with the highest priority run-time first, the sum of generated run-times with priorities higher than or equal to $Pr(rt_k)$ must be greater than the run-time consumed in $[t_0, t]$. Therefore,

$$\begin{aligned} &\textstyle\sum_{i=1}^{n} \lfloor (t - t_0)/P(T_i) \rfloor \times W(T_i)/U > t - t_0 \\ &\implies \textstyle\sum_{i=1}^{n} W(T_i)/P(T_i) > U \implies U > U \end{aligned} \tag{7}$$

Which is a contradiction. Therefore, a run-time is depleted at or before its deadline.

Since any run-time available to a job $J_{i,j}$ has earlier or equal deadlines, they are all depleted at $D(J_{i,j})$. □

Lemma 3.3. *With the EEDS algorithm, there is always available run-time for any released and unfinished job* $J_{i,j}$ *if* $U \leq 1$*. That is,* $R^r(J_{i,j}, t) > 0$ *if* $R^e(J_{i,j}, t) > 0$ *and* $t \geq R(J_{i,j})$*.*

Proof: Suppose the claim is false. Let t be the first time that there is no available run-time for any pending job. Let $J_{i,j}$ be the job with the highest priority.

Let $rt(J_{i,j})$ be the run-time associated with $J_{i,j}$. It is easy to see that $rt(J_{i,j}) = 0$ at time t, while $rt(J_{i,j}) = W(J_{i,j})/U$ at time $R(J_{i,j})$. Therefore, there must be a time instance $t' \in (R(J_{i,j}), t]$ at which $rt(J_{i,j}) < R^e(J_{i,j}, t')$. Let t' be the first of such time instances, it follows that

Device[1]	$P_a, P_s, P_{wu}(P_{sd})$[2]	$t_{wu}(t_{sd})$
Realtek Ethernet Chip	$0.19, 0.085, 0.125(W)$	$10(ms)$
MaxStream wireless module	$0.75, 0.005, 0.1(W)$	$40(ms)$
IBM Microdrive	$1.3, 0.1, 0.5(W)$	$12(ms)$
SST Flash SST39LF020	$0.125, 0.001, 0.05(W)$	$1(ms)$
SimpleTech Flash Card	$0.225, 0.02, 0.1(W)$	$2(ms)$
Fujitsu 2300AT Hard disk	$2.3, 1.0, 1.5(W)$	$20(ms)$

Table 1. Device Specifications.

$rt(J_{i,j}) = R^e(J_{i,j}, t'-1)$ at time $t'-1$. Since $rt(J_{i,j})$ is consumed during $[t'-1, t']$, it must be the at head of RT-list at time $t'-1$. Let J_{exec} be the job that executes during $[t'-1, t']$, then J_{exec} can only be one of following three cases:

1. J_{exec} is $J_{i,j}$. In this case, $R^e(J_{i,j}, t'-1) = R^e(J_{i,j}, t') - 1$. Therefore, $rt(J_{i,j}) = R^e(J_{i,j}, t')$ at time t'. It contradicts our assumption of t'.
2. J_{exec} is a lower priority job or the CPU is idle. Since $rt(J_{i,j})$ is at the head of RT-list and $rt(J_{i,j}) = R^e(J_{i,j}, t'-1)$ at time $t'-1$, the job slack of $J_{i,j}$ is 0 at time $t'-1$. It follows all devices needed by $J_{i,j}$ are active at $t'-1$. Thus $J_{i,j}$ is in the ready queue at time $t'-1$. The execution of J_{exec} contradicts the scheduling rule of EEDS algorithm.
3. J_{exec} is a higher priority job. In this case, the execution of J_{exec} consumes the run-time with a lower priority. This contradicts the assumption that $J_{i,j}$ is the first job with no available run-time.

Thus each case leads to a contradiction. That completes our proof of Lemma 3.3.

□

Proof of Theorem 3.1:

For the proof of necessity of the Theorem, we need to show that the EEDS scheduler cannot find a schedule if $U > 1$. The proof is trivial and is omitted here.

For sufficiency, suppose the claim is false. Let $J_{i,j}$ be the first job that misses its deadline at $D(J_{i,j})$. According to Lemma 3.3, the available run-time to $J_{i,j}$ at time $D(J_{i,j})$ must be larger than 0. However, this contradicts Lemma 3.2 because the run-time available to $J_{i,j}$ should be depleted at $D(J_{i,j})$. That completes our proof.

4 Evaluation

We evaluated the EEDS algorithms using an event-driven simulator. This approach is consistent with evaluation approaches adopted by other researchers for energy-aware I/O scheduling [5, 7, 6]. To better evaluate EEDS, we compared EEDS with the MDO algorithm for each simulation. MDO is an offline method proposed in [8]. The MDO algorithm uses a real-time scheduling algorithm, e.g., EDF, to generate a feasible real-time job schedule, and then iteratively swaps job segments to reduce energy consumption in device power state transitions. After the heuristic-based job schedule is generated, the device schedule is extracted. That is, device power state transition actions and times are recorded prior to runtime and used at runtime. Although MDO can achieve nearly optimal energy savings (when job execution times are equal to their WCETs), the computation overhead of MDO can be huge and cannot be used as an online method. It is reported in [8] that the computational complexity of MDO algorithm is $O(pH^2)$, where p is the number of devices used and H is the hyperperiod of task set.

The devices used in experiments are listed in Table 1. The data were obtained from data sheets provided by the manufacturer. The sources of the data can be found in [1]. We evaluated the energy savings by the *normalized energy savings*, which is the amount of device energy saved under a DPM algorithm relative to the case when no DPM technique is used, wherein all devices remain in the active state over the entire simulation. The normalized energy savings is computed using Equation (8).

$$\textit{Normalized Energy Savings} = 1 - \frac{\textit{Energy with DPM}}{\textit{Energy with No DPM}} \quad (8)$$

Task sets were randomly generated in all experiments. Each generated task set contained $1 \sim 8$ tasks. Each task required a random number ($0 \sim 2$) of devices from Table 1. The periods of tasks were randomly chosen in the range of $[50, 2000]$. Task WCETs were randomly selected such that the system utilization $U \leq 1$. We repeated each experiment 500 times and present the mean value.

We compared the scheduling overhead of EEDS with respect to EDF in our simulations. The *relative scheduling overhead* was used to evaluate the scheduling overhead of EEDS, which is given by

$$\textit{relative scheduling overhead} = \frac{\textit{sched overhead with EEDS}}{\textit{sched overhead with EDF}} - 1$$

The mean value of the relative scheduling overhead of EEDS is 5.2%. Considering that the scheduling overhead of EDF is very low, a relative overhead of 5.2% is very affordable.

4.1. Average energy savings

The first experiment measured the overall performance of EEDS. The best/worst case execution time ratio was set to 1. Figure 4 shows the mean normalized energy saving for EEDS and MDO under different system utilizations.

On average, MDO performs slightly better than EEDS. This is consistent with our expectations. The reason comes from the fact that this experiment assumed the runtime job execution is exactly as computed with MDO at the offline phase, i.e., job execution times were equal to their WCETs and job arrival times were known at the offline phase. In this case, MDO saves more energy by swapping job segments to

[2]Most vendors report only a single switching time and energy overhead. We used this time for t_{wu}, t_{sd} and this energy overhead for P_{wu}, P_{sd}.

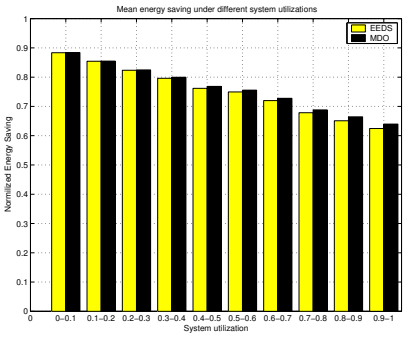


Figure 4. Normalized energy savings with various system utilizations.

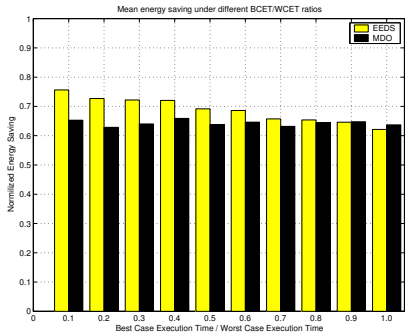


Figure 5. Normalized energy savings for various ratios of the BCET/WCET.

reduce energy consumption in device power state transitions. Therefore, MDO performs nearly optimal in this case.

With more flexibility and much less overhead, EEDS performs comparable to MDO. As shown in Figure 4, MDO has additional average energy savings of less than 1.53% over EEDS for all system utilizations. And when the system utilization is less than 80%, EEDS performs almost the same as MDO (the additional energy saving is less than 1%).

4.2. Reclaiming unused WCETs to save energy

In practice, job actual execution times can be less than their WCETs. Unused WCETs can be reclaimed to save energy under EEDS. In this experiment, we evaluate the ability of EEDS to save energy by utilizing the slack coming from unused WCETs. Recall that in our method to compute job slack, unused WCETs are kept in the RT-list and can be used to increase the job slack of lower priority jobs.

Figure 5 shows the normalized energy savings for EEDS and MDO with increasing best/worst case execution time ratios. In this experiment, the actual execution time of a job was randomly generated between the best case execution time and the worst case execution time. The worst case system utilization is set between 90% and 100%. As shown in Figure 5, EEDS saves more energy when the ratio of the best/worst case execution time is smaller, showing that it can dynamically reclaim unused WCETs to save energy.

As an offline scheduling method, MDO computed device schedules at the offline phase and applied at runtime, making it unable to effectively adapt to changes at the runtime. As shown in Figure 5, MDO saves less energy than EEDS when the best/worst case execution time is less than 90%.

5 Conclusion

EEDS is a hard real-time scheduling algorithm for conserving energy in device subsystems. This algorithm supports the preemptive scheduling of periodic tasks. As an online scheduling algorithm, EEDS is flexible enough to adapt to changes in the operating environment, and still achieves significant energy savings. Although not addressed in this paper, our work can be applied to the sporadic task model without any modification.

The problem of finding a feasible schedule that consumes minimum I/O device energy is NP-hard. EEDS is not a panacea for all systems. Instead, this work provides the foundation for a family of general, online energy saving algorithms that can be applied to systems with hard temporal constraints. In the EEDS algorithm, we do not address the issue of resource sharing and blocking. We plan to integrate resource accessing policies with EEDS in future work.

References

[1] Cheng, H., Goddard, S., "Online Energy-Aware I/O Device Scheduling Algorithms",Technical Report, 2005

[2] Jejurikar., R. and Gupta, R., "Dynamic Slack Reclamation with Procrastination Scheduling in Real-Time Embedded Systems", DAC 2005.

[3] Liu and Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the ACM*, 20(1), January, 1973.

[4] Niu., L and Quan., G., "Reducing both dynamic and leakage energy consumption for hard real-time systems", CASE, 2004.

[5] Swaminathan, V., Chakrabarty, K., and Iyengar, S.S., "Dynamic I/O Power Management for Hard Real-time Systems" CODES, 2001.

[6] Swaminathan, V., Chakrabarty, K., "Pruning-based energy-optimal device scheduling for hard real-time systems", CODES, 2002.

[7] 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*, vol 22, pages 847–858, July 2003.

[8] Swaminathan, V., and Chakrabarty, K., "Pruning-based, Energy-optimal, Deterministic I/O Device Scheduling for Hard Real-Time Systems", *ACM Transactions on Embeded Computing Systems*, 4(1):141-167, February 2005.

[9] Zhang, F., Chanson, S., "Processor Voltage Scheduling for Real-Time Tasks with Non-Preemptible Sections",RTSS,2002.