2005

Utilizing Device Slack for Energy-Efficient I/O Device Scheduling in Hard Real-Time Systems with Non-preemptible Resources

Hui Cheng
University of Nebraska-Lincoln, hcheng@cse.unl.edu

Steve Goddard
University of Nebraska-Lincoln, goddard@cse.unl.edu

Utilizing Device Slack for Energy-Efficient I/O Device Scheduling in Hard Real-Time Systems with Non-preemptible Resources

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

Technical Report TR-UNL-CSE-2005-0010
Dec 16, 2005

Abstract

The challenge in conserving energy in embedded real-time systems is to reduce power consumption while preserving temporal correctness. 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 with non-preemptible shared resources. We extend the concept of device slack *proposed in [2] to support non-preemptible shared resources; and propose an online energy-aware I/O scheduling algorithm: Energy-efficient Device Scheduling with Non-preemptible Resources (EEDS_NR). The EEDS_NR 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.*

1. Introduction

In recent years, many embedded real-time systems have emerged with energy conservation requirements. Most of these systems consist of a microprocessor with I/O devices and batteries with limited power capacity. Therefore, aggressive energy conservation techniques are needed to extend their lifetimes. Traditionally, the research community has focused on processor-based power management techniques, with many articles published on processor energy conservation. On the other hand, research of energy conservation with I/O devices 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 [21]. I/O devices have been identified as the one of the major energy consuming components in many real-time embedded systems. The focus of this paper, therefore, is to investigate I/O-based energy conservation techniques for real-time embedded systems.

At the operating system (OS) level, I/O device energy can be saved 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. Much previous work (e.g., [17, 4, 18]) has been done on scheduling tasks to maximize system idle time, also known as *system slack*. However, none of the previous work studied slack for scheduling I/O devices. Our study shows that the concept of system

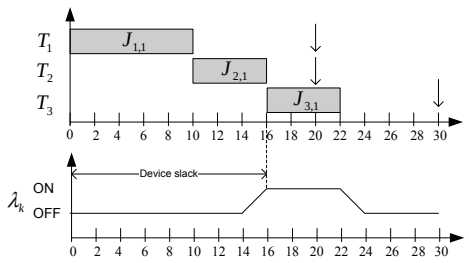


Figure 1. Device slack example. The periods of task T_1, task T_2 and task T_3 are $20, 20, 30$ respectively; and the worst case execution times for these tasks are $10, 6, 6$ respectively. λ_k is used by T_3.

slack is not well suited for energy-efficient I/O device scheduling. Figure 1 shows an example to illustrate this problem. In this example, the system utilization is 1, leaving no system slack for any job. However, the device λ_k can still be put into sleep from time 0 to 14 without causing deadline misses.

In [2], we proposed a concept called *device slack* to represent the length of time that a device can be put in the low power state without causing any job to miss its deadline. Based on the concept of device slack, we developed an online energy-aware I/O device scheduling algorithm, *i.e.*, Energy-efficient Device Scheduling (EEDS), for hard real-time systems based on the fully preemptive periodic task model. However, as with other energy-efficient device scheduling algorithms for the preemptive task model, EEDS does not support non-preemptible shared resources.

When performing preemptive scheduling with I/O devices, some I/O devices become important shared resources whose access needs to be carefully managed. For example, a job that performs an uninterruptible I/O operation can block the execution of all jobs with higher priorities. Thus the time for the uninterruptible I/O operation needs to be treated as a non-preemptive resource access. Resource accessing policies need to be integrated into device scheduling.

The research of shared resources accessing policies is a mature field. Successful policies such as Basic Preemption-Ceiling Protocol (BPCP) [7] and Stack Resource Policy (SRP) [1] have been well studied and applied in real-time systems. However, the integration of existing resource accessing policies and device scheduling is not a trivial task. Directly applying such policies can cause schedule anomalies. For example, with some needed device in the sleep state, a job may suspend itself when it holds a non-preemptible resource, such as read/write buffers. This can enlarge the blocking time for higher priority jobs and cause deadline misses.

In this paper, we extend the EEDS algorithm [2] to support non-preemptible shared resources. The new algorithm, *i.e.*, Energy-efficient Device Scheduling with Non-preemptible Resources (EEDS_NR), uses Earliest Deadline First (EDF) [6] to schedule jobs, and BPCP [7][1] to control access to shared resources. Our study shows that BPCP is preferable to SRP in terms of energy savings. To the best of our knowledge, EEDS_NR is the first energy-efficient device scheduling algorithm for preemptive task sets with non-preeptible shared resources.

[1]BPCP is different from the Priority-Ceiling Protocol (PCP), which uses the priority ceiling rather than preemption ceiling.

The rest of this paper is organized as follows. Section 2 discusses related work. The system model is presented in Section 3. Section 4 reviews how BPCP works with EDF. Section 5 describes the proposed algorithms. Section 6 describes how we evaluated our system and presents the results. Section 7 presents our conclusions and describes future work.

2. Related Work

Most Dynamic Power Management (DPM) techniques for devices are based on switching a device to a low power state (or shutdown) during an idle interval. DPM techniques for I/O devices in non-real-time systems focus on switching the devices into low power states based on various policies (e.g., [10, 8]). These strategies cannot be directly applied to real-time systems because of their non-deterministic nature.

In [5, 26], researchers try to achieve system-wide energy savings including both processor energy and I/O device energy. They made a simplified assumption that there is no energy penalty and delay for power state transition. Therefore, they used an aggressive device scheduling algorithm, which turns off devices whenever they are not in use. However, this assumption is not true for real-world I/O devices and the algorithm cannot be applied to hard real-time systems if devices that have non-zero transition delays are used.

Some practical energy-aware I/O scheduling algorithms [19, 20, 21, 22] have been developed for hard real-time systems. Among them, [19, 20, 21] are non-preemptive methods, which are known to have limitations. With non-preemptive scheduling, a higher priority task that has been released might have to wait a long time to run (until the current task gives up the CPU). This reduces the set of tasks that the scheduler can support with hard temporal guarantees. For this reason, most commercial real-time operating systems support preemptive task scheduling.

Maximum Device Overlap (MDO) [22] is an offline method for preemptive schedules. The MDO algorithm uses a real-time scheduling algorithm, e.g., EDF or RM, 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.

A deficiency of the MDO algorithm is that it does not explicitly address the issue of resource blocking. It is usually impossible to estimate when a resource blocking will happen at the offline phase. Thus it is hard to integrate a resource accessing policy into MDO. Another problem with MDO is that it does not consider the situation when job executions are less than their WCET; the schedule is generated with jobs' WCET. Even without resource blocking, the actual job executions can be very different from the pre-generated job schedule. A fixed device schedule cannot effectively adapt to actual job executions.

In [2], we proposed an online energy-aware I/O device scheduling algorithm, *i.e.*, EEDS, for hard real-time systems based on the fully preemptive periodic task model. EEDS provides more flexibility when compared to MDO. However, EEDS does

not support non-preemptible shared resource either.

The EEDS_NR algorithm proposed in this paper removes these drawbacks. As an online scheduling algorithm, EEDS_NR is flexible enough to adapt to changes in the operating environment, such as early job completions. By deliberately scheduling the task execution as well as device power state changes, EEDS_NR achieves significant energy savings for task sets that have feasible preemptive schedules with blocking for shared resources. To the best of our knowledge, no previous publication has addressed this problem.

3. System model

In this section, we briefly discuss the device and task models that we have used in our work.

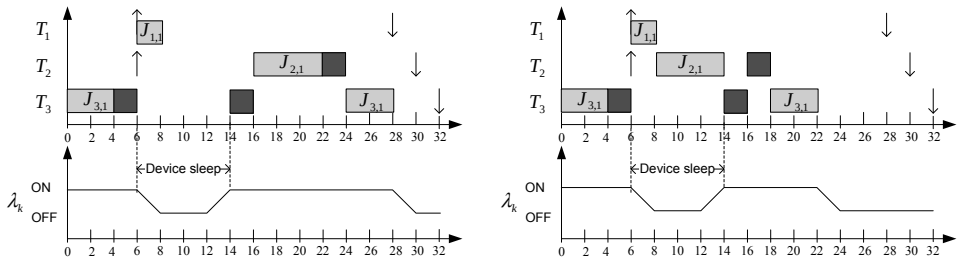
3.1. Device model

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. 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.

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.

3.2. Task model

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 four tuple $(P(T_i), W(T_i), Dev(T_i), Res(T_i))$ 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, and $Res(T_i) = \{res_1, res_2, ...res_n\}$ is the set of resources required by the task. Note that $Dev(T_i)$ specifies physical devices required by a task T_i, while $Res(T_i)$ specifies how these devices appear as shared resources to task T_i. A non-preemptive device may appear as a shared resource with different access times to different tasks; and a preemptive device may be included in $Dev(T_i)$ but not in $Res(T_i)$.



(a) The job scheduling with SRP. (b) The job scheduling with BPCP.

Figure 2. Device λ_k is required by $J_{3,1}$ and is in the sleep state during time $[6, 14]$. $J_{2,1}$ and $J_{3,1}$ require a non-preemptible shared resource res, which is represented by the dark section.

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 and the deadline of $J_{i,j}$ is denoted by $R(J_{i,j})$ and $D(J_{i,j})$. 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 $Pr(J_{i,j})$. Note that $Pr(J_{i,j})$ is not changed during execution, though the actual priority may change due to priority inheritance with BPCP. Let $Pr_{dyn}(J_{i,j}, t)$ be the actual priority of job $J_{i,j}$ at time t. $Pr_{dyn}(J_{i,j}, t)$ is equal to $Pr(J_{i,j})$ at the time $J_{i,j}$ is released.

With BPCP, each task T_i is assigned a *preemption level* $PL(T_i)$, which can be the reciprocal of the period of the task. All jobs of a task have the same preemption level as that of the task. A job with a lower preemption level cannot preempt a job with a higher preemption level. The preemption ceiling of any resource res_i is the highest preemption level of all the tasks that require res_i. Let $PL_{high}(J_{i,j}, t)$ be the highest preemption level ceiling of all the resources that are held by job $J_{i,j}$ at time t. The maximal length that a job of task T_k can be blocked is represented by $B(T_k)$.

4. Review of BPCP

BPCP [7] is used in EEDS_NR to control access to shared resources. BPCP is a non-stack version of the well known SRP [1]. In systems without device scheduling, SRP is preferable to BPCP because it can reduce the number of context switches. However, BPCP is chosen in this work because it is more energy efficient than SRP. The reason will be discussed shortly.

As discussed before, $PL(T_i)$ represents the *preemption level* of T_i. The preemption ceiling of any resource res_i is the highest preemption level of all the tasks that require res_i. Let $\Pi(t)$ be the current ceiling of the system, which is the highest-preemption level ceiling of all the resources that are in use at time t. The rules of BPCP were stated in [7] as follows.

1. "Scheduling Rule: At its release time t, the current priority of every job J is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3. Every ready job J is scheduled preemptively and in

a priority-driven manner at its current priority."

2. "Allocation Rule: Whenever a job J requests resource res at time t , if res is held by another job, J's request fails, and J becomes blocked. On the other hand, if res is free, one of the following two conditions occurs:

   (i) If J's preemption level $PL(J)$ is higher than the current preemption ceiling $\Pi(t)$ of the system, res is allocated to J.

   (ii) If J's preemption level $PL(J)$ is not higher than the ceiling $\Pi(t)$ of the system, res is allocated to J only if J is the job holding the resource(s) whose preemption ceiling is equal to $\Pi(t)$; otherwise, J's request is denied, and J becomes blocked."

3. "Priority-Inheritance Rule: When J becomes blocked, the job J_{low} which blocks J inherits the current priority of J. J_{low} executes at its inherited priority until the time when it releases every resource whose preemption ceiling is equal to or higher than $PL(J)$; at that time, the priority of J_{low} returns to its priority at the time when it was granted the resource(s)."

A major difference between SRP and BPCP is: with SRP, a job is blocked from starting execution until its preemption level is higher than the current ceiling $\Pi(t)$ of the system. With BPCP, a job is blocked only when it actually requires resources. That is, BPCP delays the blocking as late as possible. In a system without device scheduling, the response time of individual jobs is similar for both BPCP and SRP. But this is not true with device scheduling, in which jobs can be suspended because needed devices are in the sleep state.

To better illustrate this problem, consider the example shown in Figure 2. Both $J_{2,1}$ and $J_{3,1}$ require a non-preemptible shared resource res. Device λ_k is required by $J_{3,1}$. $J_{1,1}$ and $J_{2,1}$ are released at time 6. Moreover, $J_{1,1}$ preempts $J_{3,1}$ at time 6. Therefore, device λ_k is put in the sleep state to save energy, while all jobs can still meet their deadlines. In this example, BPCP reduces the response time of both jobs by utilizing the time that job $J_{2,1}$ suspends. It follows that more device energy is also saved with BPCP than with SRP.

5. Algorithm

As discussed before, EEDS_NR performs inter-task device scheduling rather than intra-task device scheduling. This approach is adopted by most real-time I/O device scheduling algorithms.

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. The idle interval is called *device slack*. We first give a brief review of device slack, as proposed in [2]; then extend the computation of device slack to support non-preemptible shared resources.

Definition 5.1. *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 $JS(J_{i,j}, t)$.

Definition 5.2. *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 $J_{i,j+1}$.

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 5.3. *Device slack.* The *device slack* is the length of time that a device λ_k can be inactive[2] without causing any 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

$$DS(\lambda_k, t) = min(JS(CurJob(T_i, t), t)) \tag{1}$$

where T_i is any task that requires λ_k.

5.1. Computing job slack

It can be seen from Equation (1) that the computation of device slack is straightforward if the job slack of all jobs are known. In [2], the *run-time* is used to compute the job slack. The concept of run-time comes from known techniques [4, 25], denoting the time budget allocated to each job. The core idea is that a job is allowed to use its own run-time as well as the run-time from higher priority jobs. Therefore, over-provisioned run-time can be used to prolong job slack. But this algorithm is not applicable when non-preemptible shared resources are present.

In the remainder of this section, we present the computation of job slack for task sets with non-preemptible shared resources. The method is based on [2] with important enhancements to guarantee temporal correctness when non-preemptible shared resources are present. As with [2], there are two sources for the job slack: (1) the job slack from the *available run-time*; and (2) the job slack from the *latest eligible time*.

5.1.1. Job slack from the available run-time

Let $init\_rt(T_i)$ denote the initial run-time assigned to each job of task T_i. Suppose a set of periodic tasks $T = T_1; T_2; T_3; \ldots T_n$ are sorted by their periods, and

$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{W(T_i)}{P(T_i)} + \frac{B(T_k)}{P(T_k)} \leq 1, \tag{2}$$

where $B(T_k)$ is the maximal length that a job in T_k can be blocked. In Section 5.3, we will prove that Equation (2) is a sufficient schedulability condition for EEDS_NR. The initial run-time assigned to each task is subject to

$$init\_rt(J_{i,j}) \geq W(J_{i,j}) \tag{3}$$

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

```
1  At any time t:
2    //J_exec is the current running job at time t.
3    If (t: a new job J_{i,j} arrives)
4      Insert_to_RT(init_rt(J_{i,j}), Pr(J_{i,j}));
5    End If
6    If (t: Pr_dyn(J_exec, t) > Pr(J_exec) &&
           Pr(rt_0) = Pr_dyn(J_exec, t))
7      rt(J_exec) ← rt(J_exec) − 1;
8      If (rt(J_exec) = 0)
9        Remove_from_RT(rt(J_exec));
10     End If
11   Else
12     rt_0 ← rt_0 − 1;
13     If (rt_0 = 0)
14       Remove_from_RT(rt_0);
15     End If
16   End If
17   rt_0 ← the head of RT-list;
18 End
```

Figure 3. The algorithm to update run-time list with non-preemptible shared resources.

$$\sum_{i=1}^{n} \frac{init\_rt(T_i)}{P(T_i)} = 1 \tag{4}$$

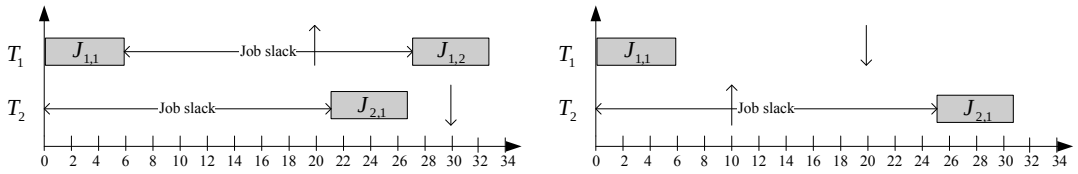
$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{init\_rt(T_i)}{P(T_i)} + \frac{B(T_k)}{P(T_k)} \leq 1, \tag{5}$$

Equation (3) makes sure that each job is assigned a sufficient initial run-time; Equation (4) fully utilizes the system capability without overloading the system; and Equation (5) guarantees that this initial assignment is consistent with the sufficient schedulablility condition for EEDS_NR.

There is at least one solution for the above equations. It is known that with EDF (BPCP), a job of the task with the longest period cannot be blocked by jobs of other tasks [7]. It follows that the blocking time for any job of the task with the longest period is 0. Therefore, a solution for the above equations can be: $\forall i, 1 \leq i \leq n-1, init\_rt(T_i) = W(T_i)$; and $init\_rt(T_n) = P(T_n) \times (1 - \sum_{i=1}^{n-1} (init\_rt(T_i)/P(T_i)))$. Usually more than one solution exists. Systems may achieve different energy savings with different initial run-time assignments to tasks. Generally speaking, enlarging the initial run-time for short-period tasks will result in frequent short job slacks; while enlarging the initial run-time for long-period tasks will result in fewer but longer job slacks. The latter initial run-time assignment strategy is adopted in the evaluation of this work.

As with [2], the run-time of a job $J_{i,j}$ has a priority and a deadline which are set equal to the job's priority and deadline, *i.e.*, $Pr(J_{i,j})$ and $D(J_{i,j})$. The priority of each job is unique as discussed in Section 3. 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 run-time at the head of the RT-list. The *available run-time* for a job $J_{i,j}$ denotes 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.

The following notation is used in the algorithm, as in [2]:



(a) Job slack coming from the run-time

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

Figure 4. Job slack examples. $T_1 = \{20, 6, \emptyset, \emptyset\}$**;** $T_2 = \{30, 6, \emptyset, \emptyset\}$**. (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** 10**.**

- 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 $J_{i,j}$ at time t.
- $R^r(J_{i,j}, t)$: the *available run-time* for job $J_{i,j}$, which is given by $\sum_{Pr(rt_i) > Pr(J_{i,j})} rt_i + rt(J_{i,j})$.

The algorithm to update the run-time list with non-preemptible shared resources is presented in Figure 3. In general, the run-time in the run-time list is reduced by 1 from the head of run-time list (rt_0) at each system tick. However, when a job J_{exec} occupies the CPU by blocking a higher priority job J_{high}, it inherits the priority of J_{high}. Accordingly, the run-time with the length of the blocking section should be moved from the run-time associated with J_{exec} ($rt(J_{exec})$) to the run-time associated with J_{high} ($rt(J_{high})$). However, the length of the actual blocking section is usually unknown, and it could be more complex when J_{exec} blocks a higher priority job that is released later or more than one resource is used. Therefore, the algorithm works as follows:

1. When the current running job J_{exec} blocks some higher priority job at time t, J_{exec} inherits the priority of the blocked job, say J_{high}. It follows that $Pr_{dyn}(J_{exec}, t) = Pr(J_{high}) > Pr(J_{exec})$. If rt_0 is the run-time associated with job J_{high} (line 6), then the execution of J_{exec} should consume the run-time associated with J_{exec} (line 7). Otherwise the run-time is consumed from the head of the RT-list (line 12).
2. When J_{exec} does not block any job at time t, the run-time is always consumed from the head of the RT-list (line 12). Note that J_{exec} can be the *idle* job.

If a run-time is depleted, the item is removed from the run-time list (lines 9,14).

Now we are ready to present the computation of the job slack from the available run-time. At time t, a job $J_{i,j}$ is in a *fully-preemptible section* if it holds no resource; otherwise it is in a *resource accessing section*. We have shown in [2] that a job can safely suspend its execution as long as its available run-time is larger than its residual execution time when it is in a fully-preemptible section. On the other hand, when a job is in a resource accessing section, the delay of its execution may block higher priority jobs and cause deadline misses. The solution to this problem is to treat the job execution in the resource accessing region as a high priority job and compute the job slack accordingly.

Therefore, the job slack from the available run-time of a job is given in two cases:

1. Job $J_{i,j}$ is in a fully-preemptible section at time t. This is the case presented in [2]. The job slack is the available run-time for the job minus the residual execution time. That is,

$$JS(J_{i,j}, t) = R^r(J_{i,j}, t) - R^e(J_{i,j}, t) \tag{6}$$

2. Job $J_{i,j}$ is in a resource accessing section at time t. Let $PL_{high}(J_{i,j}, t)$ be the highest preemption level ceiling of all the resources that are held by job $J_{i,j}$ at time t. According to EDF (BPCP), a job J_{high} can be blocked by $J_{i,j}$ only if $PL(J_{high}) \leq PL_{high}(J_{i,j}, t)$. To ensure system schedulability, the job slack of job $J_{i,j}$ at time t is the minimal job slack of all jobs that could be blocked by $J_{i,j}$. Therefore, the job slack of a job when it is in a resource accessing section can be given by

$$\begin{aligned} JS(J_{i,j}, t) &= min(JS(J_{x,y}, t)), \\ \forall J_{x,y}, PL(J_{i,j}) &\leq PL(J_{x,y}) \leq PL_{high}(J_{i,j}, t) \end{aligned} \tag{7}$$

   Note the preemption level of a task is fixed and is the same for all of its jobs. With pre-computation at the offline phase, the search for $J_{x,y}$ in Equation (7) can be done in $O(1)$ time online. Here we need to clarify that the above method identifies a superset of all jobs that can be blocked by a job $J_{i,j}$. A more accurate method to identify such a job set (and hence identify more available job slack) exists, but is more complex.

The example shown in Figure 4(a) illustrates how run-time contributes to job slack. Both jobs are released at time 0. The initial run-time assigned to $J_{1,1}$ is 6 and the initial run-time assigned to $J_{2,1}$ is 21. 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 $6 + 21 - 6 = 21$ according to Equation (6).

5.1.2. Job slack from the latest eligible time

There is another source of job slack, as introduced in [2]. A job can only start execution when it is released. So if the current 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 $init\_rt(J_{i,j})$, which will produce at least $init\_rt(J_{i,j}) - W(J_{i,j})$ unused run-time. Therefore, it is known that a job can start its execution as late as $R(J_{i,j}) + init\_rt(J_{i,j}) - W(J_{i,j})$, which is called its *latest eligible time* and is defined as follows:

Definition 5.4. *Latest eligible time.* The *latest eligible time* for a job $J_{i,j}$ is given by

$$LT(J_{i,j}) = R(J_{i,j}) + init\_rt(J_{i,j}) - W(J_{i,j}). \tag{8}$$

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

$$JS(J_{i,j}, t) = LT(J_{i,j}) - t \tag{9}$$

A unreleased job cannot hold any resources. Therefore, considering Equations (6) and (9), the job slack of a job $J_{i,j}$ that is in a fully preemptible section is given by

$$JS(J_{i,j}, t) = max(LT(J_{i,j}) - t, R^r(J_{i,j}, t) - R^e(J_{i,j}, t)) \tag{10}$$

Figure 4(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 10. At time 0, the job slack of $J_{2,1}$ coming from run-time is 21 according to Equation (6). However, the job slack coming from its latest eligible time is 25 according to Equation (9). Therefore, the job slack of $J_{2,1}$ is the larger of the two, which is 25.

In summary, the job slack of a job can be computed by Equation (10) if the job is in a fully-preemptible section at the time of computation; otherwise it should be computed by Equation (7). The job slack is increased when a new job is released or a resource is released.

5.2. The EEDS_NR algorithm

Now we are ready to present the EEDS_NR algorithm. Algorithm EEDS_NR consists of two parts: job scheduling and device scheduling. The job scheduling is presented in Section 5.2.1 and device scheduling is presented in Section 5.2.2.

5.2.1. Job scheduling

The job scheduling in EEDS_NR becomes a big challenge when cooperating with device scheduling. Device scheduling can cause basic characteristics of the BPCP policy being violated, resulting in deadline misses.

To better illustrate this problem, consider the example shown in Figure 5. In this example, task $T_1 = \{16, 10, \emptyset, \{res_1\}\}$ and task $T_2 = \{24, 2, \{\lambda_k\}, \{res_1\}\}$, both need the non-preemptible resource res_1 during their execution. The first jobs of T_1 and T_2 are released at time 0. According to their deadlines, we have $Pr(J_{1,1}) > Pr(J_{2,1}) > Pr(J_{1,2})$. T_1 is assigned an initial run-time of 10 and T_2 is assigned an initial run-time of 9. Device λ_k is required by T_2. According to the device slack computation discussed before, the device slack of λ_k is 17 at time 0. Thus the device is put in the sleep state during $[0, 17]$. At time 16, $J_{1,2}$ is released and is allocated the resource res_1 since the resource res_1 is free and the preemption level of $J_{1,2}$ is higher than the current ceiling of the system. As shown in the example, $J_{2,1}$ is blocked and misses its deadline. On the other hand, if the request of $J_{1,2}$ for resource res_1 is denied at time 16, then both jobs can meet their deadlines.

The root of this problem is that device scheduling can cause a low priority job to acquire unexpected resources. For the above example, $J_{1,2}$ cannot get the resource res_1 before $J_{2,1}$ without device scheduling. Therefore, the resource allocation rule of BPCP, which is discussed in Section 4, is modified as:

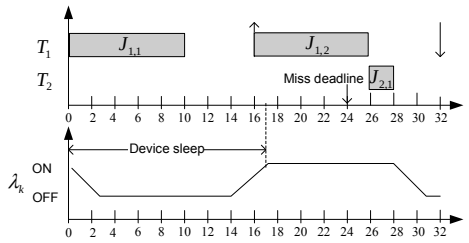


Figure 5. An example of the scheduling anomalies caused by the device scheduling. $T_1 = \{16, 10, \emptyset, \{res_1\}\}$**;** $T_2 = \{24, 2, \{\lambda_k\}, \{res_1\}\}$**.**

Allocation Rule: Whenever a job J requests resource res at time t, if res is held by another job, J's request fails, and J becomes blocked. On the other hand, if res is free, one of the following two conditions occurs:

(i) If J's preemption level $PL(J)$ is higher than the current preemption ceiling $\Pi(t)$ of the system, res is allocated to J only if there is no run-time in the run-time list with a priority higher than $Pr_{dyn}(J, t)$; otherwise, J's request is denied, and J becomes blocked.

(ii) If J's preemption level $PL(J)$ is not higher than the ceiling $\Pi(t)$ of the system, res is allocated to J only if J is the job holding the resource(s) whose preemption ceiling is equal to $\Pi(t)$; otherwise, J's request is denied, and J becomes blocked.

The modification was made in case (i) of the allocation rule to make sure that a lower priority job cannot create unexpected resource blocking. In the above example, $J_{1,2}$ cannot acquire the resource with the new allocation rule because the run-time associated with $J_{2,1}$ has a higher priority.

The job scheduler of EEDS_NR chooses the running job J_{exec} with the highest priority ($Pr_{dyn}(J_{exec}, t)$) from all jobs that are not blocked and with all needed devices being active. The temporal correctness of this scheduling algorithm is proved in Section 5.3.

5.2.2. Device scheduling

Because of the energy penalty associated with a 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. Let $BE(\lambda_i)$ denote the break-even time of device λ_i. By knowing the energy expended for transitions, $E_{wu}(\lambda_k) = P_{wu}(\lambda_k) \times t_{wu}(\lambda_k)$ and $E_{sd}(\lambda_k) = P_{sd}(\lambda_k) \times t_{sd}(\lambda_k)$, as well as the transition delay $t_{sw} = t_{wu}(\lambda_k) + t_{sd}(\lambda_k)$, we can calculate the break-even time, $BE(\lambda_k)$, as

$$P_a \times BE(\lambda_k) = E_{wu}(\lambda_k) + E_{sd}(\lambda_k) + P_s \times (BE(\lambda_k) - t_{sw}(\lambda_k))$$

$$\Longrightarrow BE(\lambda_k) = \frac{E_{wu}(\lambda_k) + E_{sd}(\lambda_k) - P_s(\lambda_k) \times t_{sw}(\lambda_k)}{P_a(\lambda_k) - P_s(\lambda_k)}$$

1 **Schedule devices at time t: (1)job release; (2)job completion; (3) new resource allocation; (4) old resource release; (5) the timer to reactivate a device is reached.**
// J_{exec} is the currently running job.
2 If (t: $\exists \lambda_k, \lambda_k \notin Dev(J_{exec})$ && $\lambda_k = active$ && $DS(\lambda_k, t) > BE(\lambda_k)$)
3 $\lambda_k \leftarrow sleep$;
4 // $Up(\lambda_k)$ is the timer set to reactivate λ_k.
5 $Up(\lambda_k) \leftarrow t + DS(\lambda_k, t) - t_{wu}(\lambda_k)$;
6 End If
7 // Update $Up(\lambda_k)$ for sleeping devices.
8 If (t: $\exists \lambda_k, \lambda_k = sleep$ && $t + DS(\lambda_k, t) - t_{wu}(\lambda_k) > Up(\lambda_k)$)
9 $Up(\lambda_k) \leftarrow t + DS(\lambda_k, t) - t_{wu}(\lambda_k)$;
10 End If
11 // Reactivate λ_k when the timer is reached.
12 If (t: $\exists \lambda_k, \lambda_k = sleep$ && $Up(\lambda_k) = t$)
13 $\lambda_k \leftarrow active$;
14 End If
15 End

Figure 6. Device scheduling of EEDS_NR.

Note that the break-even time has to be larger than the transition delay, i.e., $t_{sw}(\lambda_k)$. So the break-even time is given by

$$BE(\lambda_k) = Max(t_{sw}(\lambda_k), \frac{E_{wu}(\lambda_k) + E_{sd}(\lambda_k) - P_s(\lambda_k) \times t_{sw}(\lambda_k)}{P_a(\lambda_k) - P_s(\lambda_k)}) \quad (11)$$

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.

The device scheduling portion of EEDS_NR is actually the same as EEDS in [2], as shown in Figure 6. EEDS_NR keeps track of device slack for each device. Once a device is not required by the currently 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 (lines 2-3). At the same time, a timer is set to reactivate the device in the future (line 5). In case that the device slack for a sleeping device is increased (lines 8-10), the timer is updated accordingly. This situation could happen when a new job releases. A sleeping device is reactivated in case the wake up timer is reached (lines 12-14).

5.3. Schedulability

This section presents a sufficient schedulability condition for the EEDS_NR scheduling algorithm. The condition is the same condition used for the EDF algorithm with SRP [1]:

Theorem 5.1. *Suppose a set of periodic tasks $T = T_1; T_2; T_3; \ldots T_n$ are sorted by their periods. They are schedulable by EEDS_NR if*

$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{W(T_i)}{P(T_i)} + \frac{B(T_k)}{P(T_k)} \leq 1, \quad (12)$$

where $B(T_k)$ is the maximal length that a job in T_k can be blocked, which is caused by accessing non-preemptive resources including I/O device resources and non I/O device resources. Note that device state transition delay is not included. That is, for each task T_k, $B(T_k)$ is the same for EEDS_NR and EDF (BPCP).

The following lemmas are required before we can actually prove Theorem 5.1.

Lemma 5.2. *With EEDS_NR, a device λ_k is in the active state when the device slack of λ_k is* 0.

Proof: Suppose that the lemma is false. Let t be the first time that device λ_k is inactive while the device slack of λ_k is 0. That is, $DS(\lambda_k, t) = 0$. According to the device scheduling algorithm of EEDS_NR, there must be at least one time instance before t at which a timer is set to reactivate λ_k. Let t_0 be the last instance of such time instances. It follows that the timer to reactivate λ_k is $t_0 + DS(\lambda_k, t_0) - t_{wu}(\lambda_k)$. Since λ_k is inactivate at time t, we have

$$t_0 + DS(\lambda_k, t_0) > t \Longrightarrow DS(\lambda_k, t_0) - DS(\lambda_k, t) > t - t_0$$

Therefore, at some time during $[t_0, t)$, the device slack of λ_k must be reduced faster than 1 per system tick. Let $t_1, t_0 \leq t_1 < t$ be such a time instance. That is, $DS(\lambda_k, t_1 + 1) < DS(\lambda_k, t_1) - 1$.

According to the definition of device slack (Definition 5.3), the device slack of λ_k is the minimal job slack of all jobs requiring λ_k. Thus the job slack of a job that requires λ_k must be decreased more than 1 at time t_1. Let $J_{i,j}$ be such a job. That is, $JS(J_{i,j}, t_1+1) < JS(J_{i,j}, t_1) - 1$. It is easy to see that $J_{i,j}$ cannot execute during $[t_1, t_1+1]$ because λ_k is inactive. Therefore, $J_{i,j}$ can be either in a fully-preemptible section or in a resource accessing section during $[t_1, t_1 + 1]$. We discuss it in these two cases.

Case 1 $J_{i,j}$ is in a fully-preemptible section. According to Equation (10), the job slack of $J_{i,j}$ at time t_1 is the larger of $R^r(J_{i,j}, t_1) - R^e(J_{i,j}, t_1)$ and $LT(J_{i,j}) - t$. It is easy to see that both items can be reduced at most by 1 per system tick. Therefore, in this case we have $JS(J_{i,j}, t_1 + 1) \geq JS(J_{i,j}, t_1) - 1$.

Case 2 $J_{i,j}$ is in a resource accessing section. In this case, the job slack of job $J_{i,j}$ is the minimal job slack of all jobs that can be blocked by $J_{i,j}$. Suppose the set of jobs that can be blocked by $J_{i,j}$ is α. For any job $J_{x,y} \in \alpha$, we have $PL(J_{i,j}) \leq PL(J_{x,y}) \leq PL_{high}(J_{i,j}, t_1)$, as presented in Equation (7). Furthermore, for any job $J_{x,y} \in \alpha$, $J_{x,y}$ must be in a fully-preemptible section during $[t_1, t_1 + 1]$. Otherwise, either (1) $J_{x,y}$ is allocated a resource when $J_{i,j}$ is holding a resource with a preemption ceiling level ($PL_{high}(J_{i,j}, t_1)$) no lower than the preemption level of $J_{x,y}$ ($PL(J_{x,y})$); or (2) $J_{i,j}$ is allocated a resource when $J_{x,y}$ is holding a resource with a preemption ceiling level (higher than or equal to $PL(J_{x,y})$) no lower than the preemption level of $J_{i,j}$ ($PL(J_{i,j})$). Both cases violate the resource allocation rule of EEDS_NR.

Since $J_{x,y}$ is in a fully-preemptible section during $[t_1, t_1 + 1]$, the job slack of $J_{x,y}$ can be decreased by at most 1 during $[t_1, t_1 + 1]$ as discussed in the first case. It follows that the job slack of $J_{i,j}$ can be decreased by at most 1 during $[t_1, t_1 + 1]$.

Thus each case leads to a contradiction. This completes our proof of Lemma 5.2.

□

Lemma 5.3. *Let $rt(J_{i,j})$ be the run-time associated with $J_{i,j}$. With the EEDS_NR algorithm, $rt(J_{i,j})$ is no less than the worst case residual execution time of job $J_{i,j}$ at any time t. That is, $rt(J_{i,j}) \geq R^e(J_{i,j}, t)$.*

Proof: Suppose that the lemma is false. Let t be the first time that the run-time associated with a job is less than the worst case residual execution time of the job. Let $J_{i,j}$ be the job with the highest priority of such jobs.

It is obvious that the lemma is true for an unreleased job $J_{i,j}$ since $init\_rt(J_{i,j}) \geq WCET(J_{i,j})$. Thus here we let $J_{i,j}$ be a released job. It is known that $rt(J_{i,j}) = init\_rt(J_{i,j}) \geq WCET(J_{i,j})$ at time $R(J_{i,j})$. It follows that $t > R(J_{i,j})$.

By the choice of t, the run-time associated with $J_{i,j}$ is equal to the worst case residual execution time of $J_{i,j}$ at time $t-1$. That is, $rt(J_{i,j}) = R^e(J_{i,j}, t-1)$ at time $t-1$.

$rt(J_{i,j})$ can be reduced during $[t-1, t]$ in two cases: (1) $rt(J_{i,j})$ is at the head of RT-list at time $t-1$ and is consumed by job execution; (2) $J_{i,j}$ blocks a higher priority job and there is no run-time in the run-time list with a priority higher than $Pr_{dyn}(J_{i,j}, t-1)$. Next we proceed with our proof in these two cases.

Case 1: $rt(J_{i,j})$ is at the head of RT-list at time $t-1$ and is consumed by job execution;. Let J_{exec} be the job that executes during $[t-1, t]$, then J_{exec} can only be one of the following three cases:

Case i: 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.

Case ii: 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 $t-1$. J_{exec} can execute during $[t-1, t]$ only if $J_{i,j}$ is blocked in $[t-1, t]$ by a lower priority job. We discuss it in two cases: (a) J_{exec} is the blocking job; and (b) J_{exec} is not the blocking job.

Case a: J_{exec} blocks $J_{i,j}$. in this case, J_{exec} inherit the priority of $J_{i,j}$ and there is no run-time with priority higher than $Pr_{dyn}(J_{exec}, t-1)$. Thus the execution of J_{exec} consumes its own run-time, while $rt(J_{i,j})$ remains unchanged. This contradicts our definition of t.

Case b: J_{exec} is not the blocking job. Let J_{low} be the job that blocks $J_{i,j}$ at time $t-1$. Since the job slack of $J_{i,j}$ is 0 and $J_{i,j}$ is blocked by J_{low}, the job slack of J_{low} should also be 0 at time $t-1$ according to the job slack computation discussed in Section 5.1. It follows that the device slack of all devices used by J_{low} are 0. Thus all devices used by J_{low} are active according to Lemma 5.2. In this case, J_{exec} cannot execute since J_{low} inherits the priority of $J_{i,j}$, which is the highest priority of all jobs in the system.

Case iii: J_{exec} is a higher priority job. In this case, the execution of J_{exec} consumes the run-time associated with $J_{i,j}$. It follows that there is no available run-time for J_{exec} at time $t-1$. This contradicts the assumption that $J_{i,j}$ is the first job with no available run-time.

Case 2: $J_{i,j}$ blocks a higher priority job and there is no run-time has a priority higher than $Pr_{dyn}(J_{i,j}, t-1)$. In this case, the execution of J_{exec} consumes its own run-time. It follows that $R^e(J_{i,j}, t-1) = R^e(J_{i,j}, t) - 1$. Therefore, $rt(J_{i,j}) = R^e(J_{i,j}, t)$ at time t. This contradicts our assumption of t.

Thus each case leads to a contradiction. This completes our proof of Lemma 5.3. □

Lemma 5.4. *With the EEDS_NR algorithm, the run-time available to a job $J_{i,j}$ is depleted at or before its deadline if Equation* (12) *holds.*

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_x 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 $Pr(rt_x)$ in the RT-list. Since there is no run-time before the system start time 0, t_0 is well defined.

By the choice of t_0, there is always run-time with priority higher than or equal to $Pr(rt_x)$ during $[t_0, t]$. Let α be the set of these run-times. These run-times are generated by the releasing of jobs that have priorities higher than or equal to $Pr(rt_x)$. Let ρ denote the set of such jobs and let J_{long} be the job with the longest relative deadline in ρ. Assume J_{long} is a job of a task T_k. It is clear that the relative deadlines of all jobs in ρ are less than or equal to $t - t_0$. The sum of run-times with priorities higher than or equal to $Pr(rt_x)$ generated during $[t_0, t]$ is

$$\sum_{rt_i \in \alpha} rt_i = \sum_{i=1}^{k} \lfloor (t - t_0)/T_i) \rfloor \times init\_rt(T_i) \tag{13}$$

It is possible that, at some time during $[t_0, t]$, the execution of a job does not consume the run-time with the priority higher than or equal to $Pr(rt_x)$. This could happen only when a job $J_{blocking}$ ($J_{blocking} \notin \rho$) blocks a job in ρ. By the choice of ρ, the deadline of $J_{blocking}$ is larger than t. According to EEDS_NR, $J_{blocking}$ must be released before t_0 because a low priority job cannot acquire new resources other than resources that it already held when there are pending higher priority run-times. Furthermore, there is at most one such job ($J_{blocking}$) that is not in ρ and can block a job in ρ. This conclusion can be proved as follows:

Suppose there are two blocking jobs, J_a and J_b. Both of them must have been released before t_0 and have deadlines larger than t. They must have been holding resources at t_0, and since they are blocking some job(s) in ρ, these resources must have ceilings higher than or equal to $PL(J_{long})$, which has the lowest preemption level of all jobs in ρ. Without loss of generality, we let J_b be the first job that acquired a resource res_i whose ceiling is higher than or equal to $PL(J_{long})$. Since

Device	$P_a(W)$	$P_s(W)$	$P_{wu}, P_{sd}(W)$	t_{wu}, t_{sd}(ms)[3]
Realtek Ethernet Chip [12]	0.187	0.085	0.125	10
MaxStream wireless module [11]	0.75	0.005	0.1	40
IBM Microdrive [16]	1.3	0.1	0.5	12
SST Flash SST39LF020 [15]	0.125	0.001	0.05	1
SimpleTech Flash Card [14]	0.225	0.02	0.1	2

Table 1. Device Specifications.

$PL(J_a) < PL(J_{long})$, J_a should be blocked by J_b. We get the desired contradiction. Therefore, there can be at most one job that blocks any job in ρ.

The total length of time that $J_{blocking}$ executes in $[t_0, t]$ is bounded by the longest time $J_{blocking}$ uses a resource. This is bounded by $B(J_{i,j})$ for each job $J_{i,j}$ in ρ. In particular, the maximum execution time of $J_{blocking}$ in $[t_0, t]$ is bounded by $B(J_{long})$, where J_{long} is the job with the longest relative deadline in ρ. As before, let J_{long} be a job of the task T_k. At other times during $[t_0, t]$, the execution of jobs always consumes the run-time in α.

Since the run-time rt_x is not depleted at its deadline, the sum of run-times in α must be greater than the run-times in α consumed in $[t_0, t]$. Therefore,

$$\begin{aligned} &\textstyle\sum_{i=1}^{k} \lfloor (t - t_0)/P(T_i) \rfloor \times init\_rt(T_i) > (t - t_0) - B(T_k) \\ &\quad \Longrightarrow \textstyle\sum_{i=1}^{k} init\_rt(T_i)/P(T_i) + B(T_k)/P_k > 1 \end{aligned} \tag{14}$$

This contradicts the assignment of initial run-times. 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 by $D(J_{i,j})$. □

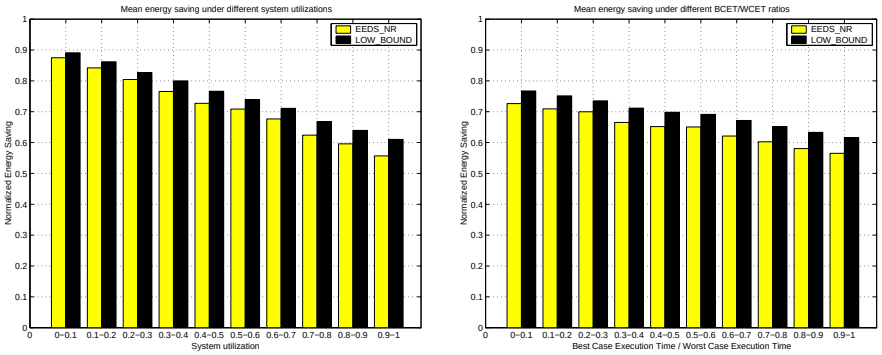
Proof of Theorem 5.1:

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 5.3, the available run-time to $J_{i,j}$ at time $D(J_{i,j})$ must be larger than 0. However, this contradicts Lemma 5.4 because the run-time available to $J_{i,j}$ should be depleted at $D(J_{i,j})$. This completes our proof.

6. Evaluation

We evaluated the EEDS_NR algorithm using an event-driven simulator. This approach is consistent with evaluation approaches adopted by other researchers for energy-aware I/O scheduling [19, 21, 20]. Since there is no previously studied energy-efficient I/O device scheduling algorithm for preemptive tasks with non-preemptible shared resources, we cannot directly compare EEDS_NR with others. However, to better evaluate the EEDS_NR algorithm, we compute the minimal energy requirement, LOW_BOUND, for each simulation. The LOW_BOUND is acquired by assuming that the time and energy overhead of device state transition is 0. A device is shut off whenever it is not required by the currently executing job, and is

[3]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}.



(a) Normalized energy savings with various system utiliztions.

(b) Normalized energy savings for various ratios of the best/worst case execution time.

Figure 7. Normalized energy savings with non-preemptible shared resources.

powered up as soon as a job requiring it begins executing. Therefore, the LOW_BOUND represents an energy consumption level that is not achievable for any scheduling algorithm.

The devices used in the experiments are listed in Table 1. The data were obtained from data sheets provided by the manufacturer. We evaluated the energy savings by the *normalized energy savings*, which is the amount of 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 (15).

$$\textit{Normalized Energy Savings} = 1 - \frac{\textit{Energy with DPM}}{\textit{Energy with No DPM}} \tag{15}$$

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 $[50, 2000]$. Resource accessing sections and WCETs of all jobs were randomly generated such that the feasibility condition shown in Equation (12) is satisfied. We repeated each experiment 500 times and present the mean value.

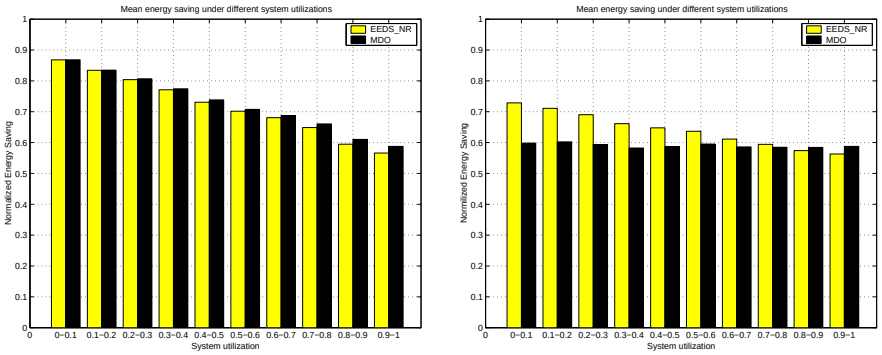
We did not measure scheduling overhead of EEDS_NR in real systems since all algorithms were evaluated with simulations. Instead, we compared the scheduling overhead of EEDS_NR with respect to EDF (BPCP) in our simulations. The *relative scheduling overhead* was used to evaluate the scheduling overhead of EEDS_NR, which is given by

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

The mean value of the relative scheduling overhead of EEDS_NR is 11.7%. Considering that the scheduling overhead of EDF (BPCP) is very low, a relative overhead of 11.7% is very affordable.

6.1. Average energy savings

The first experiment measured the overall performance of EEDS_NR. The best/worst case execution time ratio was set to 1. Figure 7(a) shows the mean normalized energy saving for EEDS_NR and LOW_BOUND under different system utilizations. On average, the EEDS_NR achieves more than 90% energy savings of LOW_BOUND. Note that no power transition overhead



(a) Normalized energy savings with various system utiliztions.

(b) Normalized energy savings for various ratios of the best/worst case execution time.

Figure 8. Comparison of EEDS_NR and MDO without non-preemptible shared resources.

is included in the computation of LOW_BOUND. Therefore LOW_BOUND represents an energy consumption level that is not achievable for any scheduling algorithm.

In practice, job actual execution times can be less than their WCETs. Unused WCETs can be reclaimed to save energy under EEDS_NR. In this experiment, we evaluate the ability of EEDS_NR 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 7(b) shows the normalized energy savings for EEDS_NR and LOW_BOUND 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 7(b), EEDS_NR 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.

6.2. Comparison of EEDS_NR and MDO without shared resources

A fully preemptive task set can be treated as a special cases of task sets with 0 shared resources. EEDS_NR reduces to EEDS when task sets are fully preemptive. In this section, we compare the energy saving of EEDS_NR with MDO [22], which is the only published energy-aware device scheduling algorithm for fully preemptive schedules except EEDS. Note that MDO is an offline method and does not support non-preemptible shared resources. Thus no shared resources were used in this experiment.

As discussed in Section 2, 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 very good 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 [22] 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.

Figure 8(a) shows the mean normalized energy saving for EEDS_NR and MDO under different system utilizations when the best/worst case execution time ratio is set to 1. On average, MDO performs slightly better than EEDS_NR. 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 reduce energy consumption in device power state transitions. Therefore, MDO performs better in this case.

With more flexibility and much less overhead, EEDS_NR performs comparable to MDO. As shown in Figure 8(a), MDO has additional average energy savings of less than 2.1% over EEDS_NR for all system utilizations. And when the system utilization is less than 70%, EEDS_NR performs almost the same as MDO (the additional energy saving is less than 1%).

Figure 8(b) shows the normalized energy savings for EEDS_NR and MDO with increasing best/worst case execution time ratios. As an offline scheduling method, MDO computed all 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 8(b), MDO saves less energy than EEDS_NR when the best/worst case execution time ratio is less than 80%.

7 Conclusion

EEDS_NR is a hard real-time scheduling algorithm for conserving energy in device subsystems. This algorithm supports the preemptive periodic tasks with non-preemptible shared resources. As an online scheduling algorithm, EEDS_NR 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. This work provides the foundation for a family of general, online energy saving algorithms that can be applied to systems with hard temporal constraints.

References

[1] Baker, T.P., "Stack-Based Scheduling of Real-Time Processes," *Real-Time Systems*, 3(1):67-99, March 1991.

[2] Cheng, H., Goddard, S., "Online Energy-Aware I/O Device Scheduling for Hard Real-Time Systems", *DATE*, 2006.

[3] Golding, R.A., Bosch, p., Staelin, C., Sullivan, T., and Wilkes, J., "Idleness if not sloth", *Winter USENIX*, 1996.

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

[5] Jejurikar, R., Gupta, R., "Dynamic Voltage Scaling for Systemwide Energy Minimization in Real-Time Embedded Systems", *ISLPED*, 2004.

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

[7] Liu, J., *Real-time Systems*, page 311, Prentice Hall, 2000.

[8] Lu Y. H., Benini L., “Power-Aware operating systems for interactive systems”, *IEEE Transactions on Very Large Scale Integration Systems*, 10(2):119-134, April 2002.

[9] Lu Y. H., Benini L. and Micheli G., “Operating-System Directed Power Reduction”, *ISLPED*, 2000.

[10] Lu Y. H., Benini L., and Micheli G., “Requester-Aware Power Reduction”, *ISSS*, 2000.

[11] Maxstream 9xstream 900mhz wireless OEM module. http://www.maxstream.net/pdf_xstreammanual.pdf.

[12] Realtek ISA full-duplex ethernet controller RTL8019AS. ftp://152.104.125.40/cn/nic/rtl8019as/spec-8019as.zip.

[13] Sha, L., Rajkumar, R., and Lehoczky, J.P., “Priority inheritance protocols: an approach to real-time synchronization”, *IEEE Transactions on Computers*, page 1175-85, 1990.

[14] Simpletech compact flash card. http://www.simpletech.com/flash/flash_prox.php.

[15] SST multi-purpose flash SST39LF020. http://www.sst.com/downloads/ datasheet/S71150.pdf.

[16] IBM microdrive DSCM-11000. http://www.hgst.com/tech/techlib.nsf/techdocs/ F532791CA062C38F87256AC00060DD49 /file/ibm md datasheet.pdf.

[17] Mochocki, B., Hu, X., and Quan G., “A realistic variable voltage scheduling model for real-time applications”, *ICCAD*, 2002.

[18] Niu., L and Quan., G., “Reducing both dynamic and leakage energy consumption for hard real-time systems”, *CASE*, 2004.

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

[20] Swaminathan, V., Chakrabarty, K., “Pruning-based energy-optimal device scheduling for hard real-time systems”, *CODES*, 2002.

[21] 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.

[22] 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.

[23] Tia, T.S., “Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-time Systems.”, Ph.D. thesis, University of Illinois at Urbana-Champaign, Department of Computing Science, 1995.

[24] Weiser, M., Welch, B., Demers, A.J., and Shenker, S., “Scheduling for Reduced CPU Energy”, *OSDI*, 1994.

[25] Zhang, F., Chanson, S., “Processor Voltage Scheduling for Real-Time Tasks with Non-Preemptible Sections”, *RTSS*, 2002.

[26] Zhuo, J., Chakrabarti, C., “System-Level Energy-Efficient Dynamic Task Scheduling”, *DAC*, 2004.