Dynamic Alteration Schemes of Real-Time Schedules for I/O Device Energy Efficiency

EUISEONG SEO
Ulsan National Institute of Science and Technology
SANGWON KIM
Agency for Defense Development
SEONYEONG PARK
Korea Advanced Institute of Science and Technology
and
JOONWON LEE
Sungkyunkwan University

Many I/O devices provide multiple power states known as the dynamic power management (DPM) feature. However, activating from sleep state requires significant transition time and this obstructs utilizing DPM in nonpreemptive real-time systems. This article suggests nonpreemptive real-time task scheduling schemes maximizing the effectiveness of the I/O device DPM support. First, we introduce a runtime schedulability check algorithm for nonpreemptive real-time systems that can check whether a modification from a valid schedule is still valid. By using this, we suggest three heuristic algorithms. The first algorithm reorders the execution sequence of tasks according to the similarity of their required device sets. The second one gathers dispersed short idle periods into one long idle period to extend sleeping state of I/O devices and the last one inserts an idle period between two consecutively scheduled tasks to prepare the required devices of a task right before the starting time of the task. The suggested schemes were evaluated for both the real-world task sets and the hypothetical task sets with simulation and the results showed that the suggested algorithms produced better energy efficiency than the existing comparative algorithms.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; C.3 [**Special-Purpose and Application-Based Systems**]: —*Real-time and embedded systems*

This work was supported by the 2009 Research Fund of the UNIST(Ulsan National Institute of Science and Technology) and also supported by the Korea Science and Engineering Foundation KOSEF grant funded by the Korea government (MEST) (No. 2009-0080381).
Author's addresses: E. Seo, School of ECE, UNIST, Ulsan, Republic of Korea; email: euiseong@unist.ac.kr; S. Kim, ADD, Daejon, Rep. of Korea; email: michael@add.re.kr; S. Park, CS Dept. KAIST, Daejeon, Rep. of Korea; email: parksy@calab.kaist.ac.kr; J. Lee, School of ICE, Sungkyunkwan Univ. Suwon, Rep. of Korea; email: joonwon@skku.edu.

DOI 10.1145/1880050.1880059 http://doi.acm.org/10.1145/1880050.1880059

General Terms: Algorithms, Theory, Performance, Reliability

Additional Key Words and Phrases: Dynamic power management, DPM, power-aware computing, real-time scheduling, real-time systems

ACM Reference Format:

Seo, E., Kim, S., Park, S., and Lee, J. 2010. Dynamic alteration schemes of real-time schedules for I/O device energy efficiency. ACM Trans. Embedd. Comput. Syst. 10, 2, Article 23 (December 2010), 32 pages.
DOI = 10.1145/1880050.1880059 http://doi.acm.org/10.1145/1880050.1880059

1. INTRODUCTION

Periodic real-time systems that guarantee the deadlines of periodically released tasks are widely used in embedded controllers of military machineries, industrial robots, vehicles, and sensing devices, as well as application software with QoS requirements, such as multimedia players and games.

In general, real-time systems are designed to follow stimulus-reaction models. Thus, real-time systems usually have I/O devices to accept stimulus and produce reaction for the stimulus. The I/O devices used in real-time systems can be familiar peripheral devices such as keyboards, mouses, network interface cards, displays, sound devices, and so on. However, depending on the purpose of the system, sensors, many special purpose electronic devices and actuators can be the candidates for the I/O devices equipped in real-time systems.

Energy efficiency in real-time systems has drawn much interest. Improved energy efficiency means not only longer operating time, but also greater stability and enhanced environmental friendliness. Improving the energy efficiency of the CPU, the biggest power-consuming component in computing systems, has been thoroughly researched. However, despite the inherent importance in real-time systems, the energy efficiency of I/O devices has been researched much less than that of CPUs.

As the importance of energy efficiency grows and the increased performance of electronic components entail increased power consumption, many I/O devices now support multiple power states. This is called the *dynamic power management* (DPM) feature. With DPM support, an idle device can be put into the sleep state in which it barely consumes power. To process requests, the device should be woken up and brought back to the active state. Unfortunately, the power-state transitions require certain lengths of time and energy [Lu and Micheli 2001], and the tasks that want to use the devices in the sleep state should wait until all the required devices wake up. Therefore, the blind use of the DPM feature may induce the extension of task execution.

Although it might cause undue delays, the extended execution time is not critical in non–real-time systems. However, the unexpected delay from the slept devices can not be tolerated in real-time systems in which breaking a deadline means critical failure. In real-time systems, the guarantee of deadlines is based on the assumption that the execution of the task will be finished within the predefined deadline.

There have been few efforts to apply the DPM into I/O devices in real-time systems. Since the energy-optimal scheduling for I/O devices is an NP-hard

problem, existing solutions are either high-complexity offline algorithms or conservative heuristic algorithms that are open to further improvements.

In this article, we suggest three heuristic algorithms that are based on the observation that we can readily transform the established valid schedules into other schedules with improved energy efficiency. This strategy requires the method to quickly determine whether the transformed schedule is still valid or not. Runtime feasibility check algorithm suggested in our previous work [Kim et al. 2006] is used.

The rest of this article is organized as follows. In Section 2, the target environment and the observation on the I/O device-centric energy-efficient real-time scheduling problem are described. A few related works are introduced in Section 3. The runtime schedulability check algorithm illustrated in Section 4 is the basis of our heuristic solutions for the I/O device-centric energy-efficient scheduling suggested in Section 5. Our algorithms are evaluated in Section 6, and we conclude our work in Section 7.

2. BACKGROUND

2.1 System Model

The target system that will be used in this article is a nonpreemptive periodic real-time system, which is a generalized model of many real-world systems in operation and has been widely used as a research model [Howell and Venkatrao 1995; Jeffay et al. 1991; Jejurikar and Gupta 2005b; Swaminathan and Chakrabarty 2003, 2005]. Generally, a periodic real-time system has a predefined task set. The properties of the task set and the tasks belonging to it are assumed to be known in a priori.

The system consists of n periodic real-time tasks. A task T_i is a tuple of (c_i, p_i, ψ_i) and each element in the tuple is defined as follows.

—c_i: The processor cycles required to execute T_i in the worst case execution path;
—p_i: the period between releases of T_i;
—$\psi_i = \{k_a, k_b, \ldots, k_p\}$: the set of devices needed for execution of T_i.

A task set τ is a set of the tasks $\tau = \{T_1, T_2, \ldots, T_n\}$. We assume that all the tasks are released simultaneously at the system starting time, and the deadline of each task is equal to the period of the task. D_i is the deadline for the last release of T_i which is the nearest deadline from the current time. When T_i is released at time t_i, D_i will be updated to $t_i + p_i$. $s_i(t_i)$ is the status of the last release of T_i at time t_i. $s_i(t)$ can be set to one of three states: released, running, and finished. Released means that the task was released and the execution has not started yet. A scheduler puts a task into Running state and starts the execution of the task. After finishing the execution, the status of the task will be marked as finished.

The target system is nonpreemptible in which execution of a task is unable to be interrupted until its end. A schedule is said to be valid if and only if no task instance misses its deadline under the schedule. And a task set is said

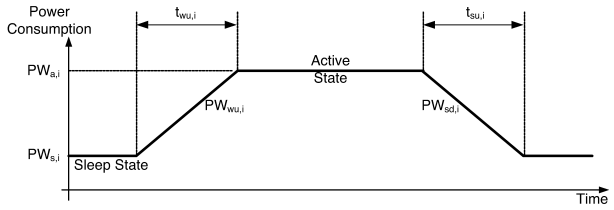


Fig. 1. Power State Transition Model for I/O devices.

to be feasible with respect to a given class of schedulers if and only if there is at least one valid schedule that can be obtained by a scheduler of this class. After this, task sets mentioned in this article are all feasible task sets unless otherwise noted. Without considering DPM of I/O devices, the scheduling of the defined task sets can be done with the EDF (earliest-deadline-first) scheduling algorithm which is the most preferred because it is the optimal solution [Jeffay et al. 1991].

The system uses a set $\kappa = \{k_1, k_2, \ldots, k_m\}$ of m I/O devices. Each device k_i has the following parameters.

—two power states: a low-power sleep state $ps_{s,i}$ and a high-power active state $ps_{a,i}$;
—a transition time from $ps_{s,i}$ to $ps_{a,i}$ represented by $t_{wu,i}$;
—a transition time from $ps_{a,i}$ to $ps_{s,i}$ represented by $t_{sd,i}$;
—power consumption during wake-up $PW_{wu,i}$;
—power consumption during shutdown $PW_{sd,i}$;
—power consumption in the active state $PW_{a,i}$;
—power consumption in the sleep state $PW_{s,i}$.

Naturally, the power consumption of a device during its power-state transition is not constant. However, to model the effect on task scheduling by the power-state transition time, we introduce the transition time and the power consumption during the transition separately, and assume that the power consumption during a state transition is the total energy consumption for the transition divided by the transition time, which means the average power.

We assume that the power consumption in the sleep state is less than the power consumed in the active state (i.e., $PW_{s,i} < PW_{a,i}$). The energy consumed by device k_i is defined as in Equation (1). Here, M is the total number of state transitions for k_i. $t_{a,i}$ is the total time spent by k_i in the active state and $t_{s,i}$ is the total time spent in the sleep state.

$$E_i = PW_{a,i} \cdot t_{a,i} + PW_{s,i} \cdot t_{s,i} + M/2 \cdot PW_{wu,i} \cdot t_{wu,i} + M/2 \cdot PW_{sd,i} \cdot t_{sd,i} \qquad (1)$$

We also assume that the power-state transition time of a device is always shorter than the minimum worst-case execution time (WCET) of tasks. This assumption eases the problem and, therefore, allows more effective solutions than the problem without the assumption. For the same reason, existing related works [Swaminathan and Chakrabarty 2003, 2005] have the common assumption.

The activation of a device is assumed to be automatically finished without any processor intervention once it is initiated by the processor. In other words, the processor issue the activation instruction for a device right before executing a task instance, the activation of the device will progress while the task is being executed. Naturally, because the device cannot be used while it is in power-state transition, the device activation instruction should be issued to be finished before it is required.

2.2 Motivation

The aim of this article is to improve the energy efficiency of I/O devices by utilizing the DPM. However, following two properties in the target system make this difficult.

Nonpreemption: Since the system runs tasks in a nonpreemptive manner, power-state changing operations are able to be issued only in task switching time or processor idle time.

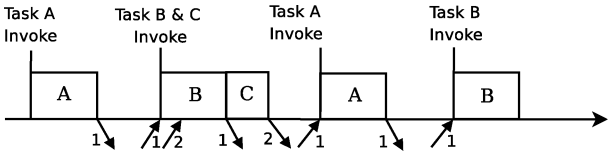
(De)Activation Overhead: Every device has a certain delay for activating or deactivating. A sleeping device can be operational at least after the delay from the start of the activating operation, and every device consumes certain amounts of energy during the activation and the deactivation. If a slept period is shorter than a certain threshold, which is called a *break-even point*, the energy consumed in deactivating and reactivating offsets the energy saving from the reduced power consumption.⌟

Under these conditions, every feasible schedule produced by all the computationally tractable scheduling algorithms has to be nonidling [Howell and Venkatrao 1995].

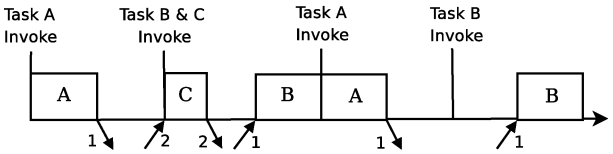
Nonidling: With idling scheduling policies, when a task has been released, it can either be scheduled, or wait a certain time before being scheduled, even if the processor is not busy. With nonidling scheduling policies, when a task has been released, it cannot wait before being scheduled if the processor is not busy. Although the idling scheduling is more flexible and intuitive, finding feasible schedules for a task set in nonpreemptive real-time systems with idling policies is known to be an NP-complete problem [Howell and Venkatrao 1995]. Therefore, well-known and widely used scheduling algorithms such as RM (rate-monotonic) and EDF are mostly nonidling schedulers.

Along with the former two conditions, the nonidling property significantly limits the utilization of the DPM feature. We now introduce two representative examples of the limitation in using DPM due to the combination of those conditions.

Figure 2(a) shows the original EDF schedule of an example task set that consists of three tasks A, B, and C with the corresponding I/O device DPM schedule. Tasks A and B use device 1, and Task C uses device 2. Tasks B and C are released at the same time, and Task B has the shorter deadline. Therefore, Task B is scheduled sooner than Task C. Task C will be started right after the finish of Task B. Since EDF is a nonidling algorithm, there is no time for device 2 to prepare between the execution of Tasks B and C. Accordingly, the activation of device 2 should be initiated right before the start time of Task



(a) Original EDF schedule.



(b) Modified schedule.

Fig. 2. Improving energy efficiency of I/O devices by modifying EDF schedule.

B and stay idle in active state during the execution of Task B. However, if we can modify the original schedule into Figure 2(b), this will be improved. As previously mentioned, inserting an idle period before the already released task without breaking deadlines is an intractable problem.

Another problem is unnecessarily frequent power-state transitions from dispersed short idle periods. In Figure 2(a), eight state transitions occur. However, by reordering Tasks B and C with postponing Task B, two activations can be concealed into one and the idle periods of device 1 in the slept state are extended.

As shown in the examples, schedules from EDF algorithms are not the best for the energy efficiency of I/O devices, and reordering of tasks and inserting idle periods may reduce the state transitions and extend sleep states. Builiding a schedule, having the minimum state transitions as well as the longest sleep periods, from the beginning is a difficult task [Swaminathan and Chakrabarty 2005]. However, if we restrict the approach to the short-term modification of the original schedule generated by the EDF algorithm, the possibility of finding good heuristic algorithms will be greatly increased.

3. RELATED WORK

Most real-time systems are built on stimuli-reactions models. Therefore, real-time systems are inherently associated with many I/O devices and a significant portion of total energy is consumed by these I/O devices. However, almost all prior works on DPM of I/O devices have focused primarily on non–real-time environments.

DPM techniques of I/O devices in non–real-time systems broadly fall into three categories: timeout-based, predictive, and stochastic. Timeout-based DPM schemes shut down I/O devices when they have been idle for a specified threshold interval [Golding et al. 1995]. The first request generated by a task for a device that is in sleep state wakes it up. The device then proceeds to service the request. Predictive schemes are more readily adaptable to workload changes than timeout-based schemes. They predict the idle period of a device based on variety observations such as the usage history [Hwang and Wu 2000], utilization of devices [Lu et al. 2000b], adaptive learning trees [Chung

et al. 1999], and so on. They control the power states with those predictions. Stochastic methods usually involve modeling device requests through different probabilistic distributions and solving stochastic models [Benini et al. 1999; Simunic et al. 1999; Irani et al. 2002]. Based on the constructed model, they control the performance states.

Each introduced approach has its own pros and cons in non–real-time systems. However, they are built on the probabilistic nature of non–real-time sporadic systems, those approaches are not suitable for periodic real-time systems. They also differ from their real-time counterparts in that they attempt to minimize task response times, rather than attempting to meet task deadlines. In real-time systems, minimizing the mean response time of a task does not guarantee that its deadline will be met.

In preemptive real-time system models, a task can be switched out by other tasks or an operating system during its execution. A device activation may occur at any necessary time point in these systems. Consequently, getting a time point for activating an I/O device can be solved by a straightforward algorithm, which just activates the device as early as its activation delay from the moment that it is required.

Energy-efficient device scheduling (EEDS) [Cheng and Goddard 2006b] was introduced for improving energy efficiency by scheduling tasks in aware of I/O device power states. The same authors also suggested EEDS NR (EEDS with nonpreemptive resources) [Cheng and Goddard 2006a] for systems having I/O devices that could not be shared by other tasks until the predominating tasks of those devices are finished. Both of them are online algorithms, which utilize dynamic slack to get better energy efficiency than offline algorithms. They proved that DPM-aware scheduling in preemptive real-time systems could improve energy efficiency significantly.

In some real-time systems, it may be impractical to switch a task while it is being executed because of context switching overhead, and limitations from embedded hardware and software. We verified that the excessive context switching occurred by the priority-based real-time scheduling algorithm drops the overall throughput down to 17% [Seo et al. 2009] of the non–priority-based scheduling. Nonpreemptive scheduling can simplify hardware and remove some complex structures in operating systems such as mutual exclusion [Dolev and Keizelman 1999; Baruah 2005]. Therefore, nonpreemptive real-time systems are frequently used in real-world embedded systems [Dolev and Keizelman 1999; Piaggio et al. 2000; Baruah 2005] and many research activities [Howell and Venkatrao 1995; Jeffay et al. 1991; Dolev and Keizelman 1999; Piaggio et al. 2000; Jejurikar and Gupta 2005b; Baruah 2005; Baruah and Chakrabarty 2006; Swaminathan and Chakrabarty 2003, 2005] have been done on the nonpreemptive real-time model.

Because the I/O power-state transition cannot be initiated during a processor executes tasks in a nonpreemptive real-time system, scheduling algorithms for nonpreemptive real-time systems change it only in idle period or between task switching. Therefore, scheduling algorithms that consider this restriction is necessary not only for improving energy efficiency, but also for guranteeing deadlines.

To the best of our knowledge, the first I/O-based technique for nonpreemptive real-time systems is the Low-Energy Device Scheduler (LEDES) [Swaminathan and Chakrabarty 2003]. LEDES takes inputs of a valid task schedule and a device-usage list for each task to generate a sequence of active or sleep states for each device. LEDES determines this sequence such that the energy consumed by the devices is minimized while guaranteeing that no task misses its deadline.

The pruning-based scheduling algorithm, Energy-Optimal Device Scheduler (EDS) [Swaminathan and Chakrabarty 2005], rearranges jobs to find the minimum energy task schedule. EDS generates a schedule tree by selectively pruning the branches of the tree at offline. Pruning is done based on both temporal and energy constraints. However, finding the optimal schedule is an NP-hard problem. Also, the alternative heuristic algorithm, which was suggested in the article, has high temporal overhead. Therefore, applying it to a task set with a long hyperperiod, the least common multiple of all the periods of the tasks in the task set, requires too much time.

All of the introduced nonpreemptive real-time scheduling algorithms are static, which means they assumed that all the tasks always consume as many CPU cycles as their WCETs. However, in real-world systems, it is a common phenomenon that the actual execution time of a task is shorter than its WCET. Static algorithms cannot utilize the dynamic slacks, which are idle periods from the early ends of the tasks.

4. RUNTIME SCHEDULABILITY CHECKING

Feasibility conditions for nonpreemptive tasks are well studied [Jeffay et al. 1991]. The optimal feasibility conditions for nonpreemptive tasks without inserted idle intervals are as given by Theorem 4.1.

THEOREM 4.1. *[Jeffay et al. 1991] A periodic task set sorted in nondecreasing order of the task period can be feasibly scheduled under a nonpreemptive EDF scheduling policy if and only if*

$$\sum_{i=0}^{n} \frac{c_i}{p_i} \leq 1 \tag{2}$$

$$\forall i, 1 \leq i \leq n; \forall t, p_1 \leq t \leq p_i : \quad c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t}{p_j} \right\rfloor \cdot c_j \leq t \tag{3}$$

In our previous work [Kim et al. 2006], we suggested a dynamic feasibility check algorithm based on Theorem 4.1. This algorithm can be used to check the validity of the modified schedules from a valid schedule on the fly. In this section, we will introduce the algorithm briefly to be used in the description of our approach.

For a feasible task set, the schedulability of the task set after scheduling T_i in a non-EDF manner can be guaranteed by checking a deadline miss within D_i, as stated in the following Theorem 4.2.

Algorithm 1. Runtime Laxity Adjustment

```
1  After release of T_i:
2      L_i ⟵ Laxity_i − (t_current − t_i_release)
3      for j = 1 to n do
4          if (s_j(t_i_release) = Released) ∧ (D_j ≤ D_i) do
5              L_i ⟵ L_i − c_j
6          end if
7      end for
8  On finish of T_i:
9      for j = 1 to n do
10         if (s_j(t_current − a_i) = Released) ∧ (D_j < D_i) then
11             L_j ⟵ L_j − a_i
12         end if
13         if (s_j(t_i_release) = Finished) ∧ (D_j > D_i) then
14             L_j ⟵ L_j + (c_i − a_i)
15         end if
16     end for
```

THEOREM 4.2. *For a feasible periodic task set $\tau = \{T_1, T_2, \ldots, T_n\}$, scheduling a task T_i in a non-EDF manner at some scheduling point in time can make valid schedule of τ if and only if it does not make any deadline miss before D_i.*

Definition 4.3. For a periodic task set $\tau = \{T_1, T_2, \ldots, T_n\}$, where $T_i = (c_i, p_i)$, sorted in nondecreasing order of the task period, $Laxity_i$ of T_i denotes the maximum value of $\boldsymbol{L}$ satisfying Equation (4).

$$\forall t,\ p_1 \le t \le p_i : \quad t \ge c_i + \sum_{j=1, j\neq i}^{n} \left\lfloor \frac{t}{p_j} \right\rfloor \cdot c_j + \boldsymbol{L} \tag{4}$$

Considering only tasks that will be released and finished during p_i by EDF scheduling, $Laxity_i$ is surplus time that can be used by any other tasks without breaking the deadlines of tasks that have releases and deadlines during p_i.

However, there can be tasks that lead $Laxity_i$ to be adjusted at runtime. Tasks can be classified into three different classes with respect to T_i: (i) tasks released prior to the release of T_i and have deadline before D_i, (ii) tasks finished after the release of T_i and have deadline after D_i, and (iii) other remaining tasks. The adjustments of $Laxity_i$ are required regarding the execution of the first two classes as shown in Algorithm 1.

Lines 1 through 7 of Algorithm 1 is called at the first scheduling of any task after release of T_i. $t_{current}$ and $t_{i_{release}}$ denote the points in time at which Algorithm 1 is executed and T_i is released, respectively. Lines 8 through 16 are called after finishing a task. All the tasks that yielded their execution chance to the finished one must have their runtime laxity value readjusted. Here, a_i is actual execution time of T_i. The value of a_i varies at every period. Before the finish of T_i, we should use c_i as the expected execution time of T_i to guarantee the schedulability because we should anticipate the worst case. After finishing T_i, we can substitute a_i for c_i. The difference between a_i and c_i is the runtime

Algorithm 2. Runtime Schedulability Check

```
1  On checking schedulability(to check validity of scheduling T_i):
2      if L_i < 0 then
3          return fail /* Cannot guarantee schedulability */
4      end if
5      for j = 1 to n do
6          if (s_j(t_current) = Released) ∧ (D_j < D_i) ∧ (i ≠ j) ∧ (L_j < c_i) then
7              return fail /* Cannot guarantee schedulability */
8          end if
9      end for
10     return success /* Can guarantee schedulability */
```

dynamic slack of T_i. The dynamic slack increases the chance for the tasks to be scheduled in an non-EDF manner.

Now, before scheduling T_i in a non-EDF manner, the schedulability of the task set after scheduling T_i must be checked using Algorithm 2.

THEOREM 4.4. *For a feasible task set $\tau = \{T_1, T_2, \ldots, T_n\}$, where $T_i = (c_i, p_i)$, the task set τ after scheduling a task in a non-EDF manner using Algorithm 1 and 2 is also feasible.*

5. SCHEDULING HEURISTICS

In this section, we will suggest three heuristic algorithms that transform valid schedules into the schedules with improved energy efficiency of I/O devices. Before introducing those algorithms, we suggest the power-state controlling policy for I/O devices, which decides the power state of each device based on the task schedules to guarantee that the device is ready in active state before it is required by a task.

5.1 Power-State Control Policy

The power states of the devices in the target system should be decided considering the task schedule. In our work, the decisions are made with a simple algorithm which is described as follows:

Since the target system is nonpreemptible, the power-state changing operations can be issued only in idle periods or task switching time. Also, since the transition operation can not be finished instantly, the activation operation of a device should be issued before the scheduling of the task requiring the device.

There are two conditions regarding the device activation: (i) If there is an idle period right before a task execution starts and the idle period is longer than the longest activation time of the required devices of that task, the problem is easy; activating the required devices just before the start of the task execution by their activation delays makes all the devices ready when the task starts to be executed. However, (ii) if a task switching occurs without idle time, or with idle time, which is too short to activate all the required devices for the next

scheduled task, the activation of the devices for the next task should be issued before the beginning of the former task.

The deactivation of a device can only be done when the device is not currently used and does not need to be activated for at least the time to deactivate. Whenever a task is finished, all the activated devices are checked as to whether they can be safely deactivated or not, and only the devices that are verified safe to sleep will be deactivated to the sleep state.

5.2 Minimizing RDS Differences by Device Conscious Scheduling

The power-state changes of I/O devices are affected by the RDS (required devices set) of the next task and also the next next task. The first heuristic algorithm, device-conscious scheduling, schedules the task having the most common devices in the RDS with the last executed task. This approach was applied to non–real-time systems in a previous work and proved effective [Lu et al. 2000a]. However, to adopt the methodology into real-time systems, the deadlines should be guaranteed by formal analysis.

The algorithm works as follows.

First, if there is no released task at the time point when $T_{current}$ finishes, the first released task after idling will be T_{next}. If there are multiple released and feasible task at $T_{current}$ finishes, the algorithm chooses T_{next} among the candidates that minimizes the energy loss from the difference in the RDSs between $T_{current}$ and T_{next}.

$LossSum_i$ is the energy loss from the difference between the RDSs of $T_{current}$ and T_i when T_i is chosen as T_{next}. There are two groups of devices that affect $LossSum_i$ values.

The first group consists of the devices that are used by $T_{current}$, but not by T_i. If these devices were used by T_i, there would be no power-state transitions for these devices. Therefore, the algorithm adds the shutting-down energy values of these devices to the $LossSum_i$.

The other group is made of the devices that are used by T_i, but not by $T_{current}$. These devices should be in active state while $T_{current}$ is being executed to anticipate the consecutive execution of T_i. The differences between the active power and the sleeping power of these devices during the execution time of $T_{current}$ are also the energy loss, so these will be also added to $LossSum_i$. Finally, because these devices would not be activated before $T_{current}$, if T_i were not chosen as T_{next}, the activation energy for these devices are added to $LossSum_i$.

After calculating $LossSum$ of all the released and feasible tasks, the algorithm chooses the task with the least $LossSum$ value for T_{next}.

The task selection algorithm can be presented as Algorithm 3. T_{curr} is the last executed task for which RDS is active at the current time, and we should select the task T_{next} to be executed after T_{curr}.

As described in line 2, the algorithm chooses the task with the earliest deadline after the idle time as the next task when currently there is no ready task. If there are multiple tasks that are released at the same time after the idle time, the task with the earliest deadline among the simultaneously released tasks will be the next task following the principle behind the EDF algorithm.

Algorithm 3. Device-Conscious Scheduling

```
1  On scheduling instant:
2      Select Next Task T_next with EDF scheduling
3      if s_next(t_current) = Released then
4          LossSum_next ⟵ any arbitrarily large number
5          for All released and feasible task T_i do
6              LossSum_i ⟵ 0
7              for All k_j ∈ (ψ_curr − ψ_i) do
8                  LossSum_i ⟵ LossSum_i + t_sd,j PW_sd,j
9              end for
10             for All k_j ∈ (ψ_i − ψ_curr) do
11                 LossSum_i ⟵ LossSum_i + c_i(PW_a,j − PW_s,j)
12                 LossSum_i ⟵ LossSum_i + t_wu,j PW_wu,j
13             end for
14             if LossSum_i < LossSum_next then
15                 T_next ⟵ T_i
16                 LossSum_next ⟵ LossSum_i
17             end if
18         end for
19     end if
```

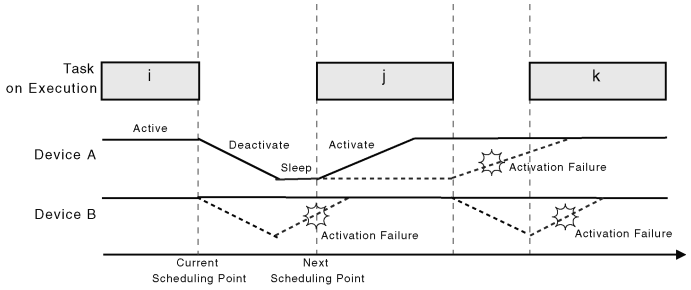
The feasible tasks in line 5 of Algorithm 3 mean the tasks that make the resulting schedule still valid based on Algorithm 2 after choosing it for T_{next}. Algorithm 3 compares the RDS of every feasible released task with the current active device set, and chooses the task that minimizes the energy loss from the difference of the RDS.

The expected energy loss of a next task candidate T_i, $LossSum_i$, can be calculated easily as lines 11 through 12 in Algorithm 3. The resulting schedules from Algorithm 3 are expected to have less power-state transitions and longer sleep periods of I/O devices than the original schedule.

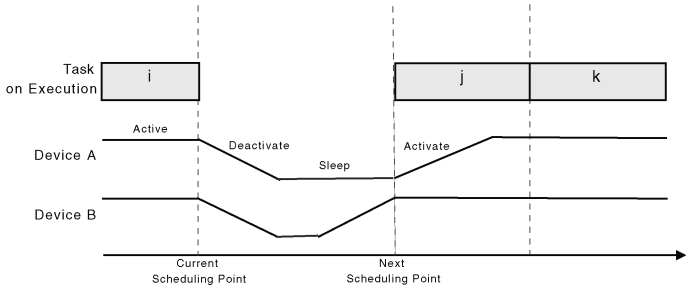
The suggested algorithm does not anticipate the case that a task instance finished earlier than its WCET. However, in actual execution of tasks, tasks complete their execution before WCET and make the dynamic slack. Most of existing dynamic task scheduling algorithms utilize this dynamic slack by expediting the start of the next task. However, the required devices may not be ready at the moment of the expedited starting time with the suggested Power-State Control Algorithm. Therefore, to apply Algorithm 3 combined with dynamic slack utilizing approaches such as Pillai and Shin [2001], Jejurikar and Gupta [2005a], and Kim et al. [2002], the task execution starting time should be expedited only to the point that all the required devices for T_{next} are ready.

5.3 Extending Idle Periods by Delaying Start of Execution

The scheduling previosuly discussed is nonidling, where the system cannot be in idle as long as there is a task in the released state. However, sometimes energy efficiency can be improved by postponing the execution of a task. Figure 3 illustrates a representative example.



(a) Original EDF schedule.



(b) Schedule with delayed starting T_j.

Fig. 3. Extended sleep period from delayed starting.

An idle period between T_i and T_j is too short to deactivate and activate device B again, which is used by both T_i, T_j, and T_k, as shown in Figure 3(a). Also, the sleeping time for device A, which is required by T_i and T_k, should be short, since the idle period between T_j and T_k is shorter than the activation delay of device A. Device A is wakened at the starting point of T_j and has to stay in active state until the starting point of T_k, even though T_j does not use device A at all.

Delaying the starting points of task execution and merging short idle periods can make a long idle period that hopefully can be used for extending sleeping periods and reducing power-state transitions, as shown in Figure 3(b).

It is clear that the energy consumption in I/O devices will be more or less reduced with extended idle periods. However, blindly extending the idle period may break the deadlines. Therefore, to determine the idle length that can be safely extended, we suggest Algorithm 4.

Suppose a situation such as Figure 4. The scheduler added an idle period as long as $Delay_{total}$ after the original idle period. Then, each task T_i is delayed for $Delay_i$ time after its release. To guarantee the schedulability of tasks with inserted idle time, we introduce a virtual task T_∞, denoted as Equation (5).

$$T_\infty = \{c_\infty, p_\infty\}, \quad where \quad c_\infty = 0, \quad p_\infty = \infty \tag{5}$$

The task T_∞ does not affect the feasibility of the original task set because its computation time is 0. We can assume that T_∞ had been released at any time we want. As discussed in Section 4, if the $Laxity_i$ of a task T_i is not less than 0,

Algorithm 4. Calculating t_{break}

```
1  T_now ⟵ current time
2  n ⟵ number of tasks
3  for ALL T_i do
4        Point[(i-1)*3].time ⟵ D_i + p_i
5        Point[(i-1)*3].type ⟵ Type1
6        Point[(i-1)*3 + 1].time ⟵ D_i + p_i + c_i
7        Point[(i-1)*3 + 1].type ⟵ Type2
8        Point[(i-1)*3 + 2].time ⟵ D_i + p_i + (p_i − Laxity_∞)
9        Point[(i-1)*3 + 2].type ⟵ Type3
10 end for
11 Sort all 3n Point[] in ascending order of time
12 Time_prev ⟵ Point[1].time
13 I ⟵ 0; OD_temp ⟵ 0; OD_total ⟵ 0
14 for i = 1 to 3n do
15       Time_cur ⟵ Point[i].time
16       OD_temp ⟵ OD_temp+I(Time_cur − Time_prev)
17       OD_total ⟵ Min(OD_total + (Time_cur − Time_prev), OD_temp)
18       Time_prev ⟵ Time_cur
19       if (Point[i].type = Type1) or (Point[i].type = Type3) then
20             I ⟵ I+1
21       else
22             I ⟵ I−1
23       end if
24       if OD_total ≥ Laxity_∞ then
25             return (Time_cur − (OD_total − Laxity_∞) − T_now)
26       end if
27 end for
```

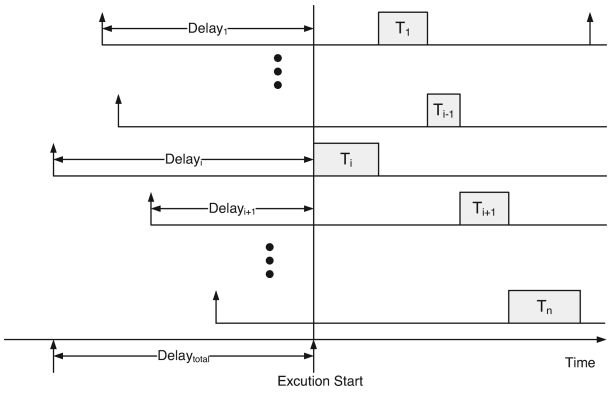


Fig. 4. Delayed scheduling with $Delay_{total}$ and $Delay_i$.

the overall processor demand is not larger than the processing time. Then, we can guarantee the schedulability until the relative deadline of T_i for the tasks that can be released during p_i.

Let us make an assumption that T_∞ was released at the point of time when $Delay_{total}$ ended. We can guarantee the schedulability of tasks to the infinity of time if the $Laxity_\infty$ value at the end of the inserted idle time is not less than

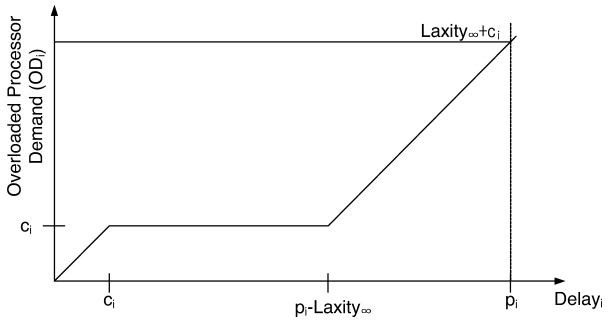


Fig. 5. Overloaded processor demand from delaying T_i.

0. Calculating the value of $Laxity_\infty$ should be done at the end of $Delay_{total}$. The delayed task execution can decrease the value of $Laxity_\infty$.

Now we will take a close look at the relationship between $Laxity_\infty$ and $Delay_{total}$. First, we define the OD_{total} as the total overloaded processor demand caused by delaying the task executions for $Delay_{total}$ that decreases the $Laxity_\infty$ at the start of T_∞. Then, the schedulability condition with overall overloaded processing demand, OD_{total}, can be presented as Equation (6).

$$Laxity_\infty \geq OD_{total} \tag{6}$$

For calculating OD_{total}, we have to obtain OD_i which is the overloaded processor demand for T_i. As shown in Figure 4, a task T_i is delayed for $Delay_i$ before the start of scheduling. OD_i is the overloaded processor demand that decreases $Laxity_\infty$. OD_i is caused by delaying task scheduling for $Delay_i$. Delaying task scheduling for $Delay_i$ causes the same effect as setting the deadline ahead by $Delay_i$. Setting a deadline of T_i ahead by $Delay_i$ decreases the $Laxity_\infty$.

If $0 \leq Delay_i \leq c_i$, the $Laxity_\infty$ would not decrease more than c_i. In this region, putting the deadline ahead cannot make $Laxity_\infty$ decrease more than c_i, but causes alternating of the sequence of workload to be done. If $c_i \leq Delay_i \leq p_i - Laxity_\infty$, putting the deadline of T_i ahead for $Delay_i$ only changes the deadline of workload, not the value of $Laxity_\infty$.

When $Delay_i$ is $p_i - Laxity_\infty$, the workload of T_i comes first at the timeline. Then, further delaying T_i would decrease the $Laxity_\infty$ linearly. Therefore, if $p_i - Laxity_\infty \leq Delay_i$, the OD_i would increase linearly as $Delay_i$ increases.

The value of OD_i, the decrease of $Laxity_\infty$, for a given $Delay_i$ can be presented as in Equation (7):

$$OD_i = Max(Min(c_i, Delay_i), Delay_i - (p_i - c_i - Laxity_\infty)) \tag{7}$$

Figure 5 depicts OD_i for a given $Delay_i$.

The total overloaded processor demand, OD_{total}, caused by delaying the execution of all released tasks is Equation (8).

$$OD_{total} \leq \sum_{i=1}^{n} OD_i \tag{8}$$

And, overloaded processor demand cannot increase faster than the increase of idle time. Therefore, Equation (9) holds.

$$\frac{dOD_{total}}{dt} \leq 1 \tag{9}$$

In Figure 5, $Delay_i$ will be 0 at the release time of T_i. The increase pattern of OD_i has three meaningful time points: D_i (the next deadline of T_i), $D_i + c_i$, and $D_i + (p_i - Laxity_\infty)$. Let us define them as *Type1*, *Type2*, and *Type3* time points, respectively. The overall procedure to calculate $Delay_{total}$ is presented as Algorithm 4.

Delaying Execution is applied to the original idle period, which has no ready task. Consequently, the returned value, t_{break}, of Algorithm 4 is the time point in future when a task should start its execution and breaks the idle period, which is the sum of $Delay_{total}$ and the current time.

We can get the value of $Delay_{total}$ with $O(nlogn)$ computational complexity. However, when that computational complexity is not allowed, we can just set the value $Delay_{total}$ as $Laxity_\infty$ with the assumption that the first release time of T_∞ is the same to that of the task that will be released first, breaking the idle period. Although this method would lose some chances to get a longer idle period, the computational complexity would be constant.

To get $Delay_{total}$, $Laxity_\infty$ needs to be calculated first. By the definition in Equation (4), $Laxity_\infty$ is decided as the maximum L value satisfying Equation (10). However, it is impossible to check the condition to infinity; we are suggesting a method for practically getting the L value.

$$\forall t,\, p_1 \leq t \leq \infty : \quad t \geq \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor \cdot c_i + \boldsymbol{L} \tag{10}$$

By definition, L cannot exceed $p_1 - c_1$ because T_1 cannot have processor cycles as many as c_1 with L that is greater than $p_1 - c_1$. Therefore, although there may exist the lower upper bound than $p_1 - c_1$, we temporarily set the upper bound of L as $p_1 - c_1$ for ease of calculation.

$$\forall t,\, p_1 \leq t \leq \infty : \quad \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor \cdot c_i \leq \sum_{i=1}^{n} \frac{t}{p_i} \cdot c_i \tag{11}$$

Equation (11) is obvious. By Equation (11), L that satisfies Equation (10) also satisfies Equation (12) always.

$$\forall t,\, \frac{p_1 - c_1}{1 - \sum_{i=1}^{n} \frac{c_i}{p_i}} \leq t \leq \infty : \quad t \geq \sum_{i=1}^{n} \frac{t}{p_i} \cdot c_i + \boldsymbol{L} \geq \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor \cdot c_i + \boldsymbol{L} \tag{12}$$

Algorithm 5. Delaying Execution

```
1  On Selecting Tasks:
2      if No Ready Task then
3          Calc t_break
4          for All T_i that are released at t_break do
5              Reduce L_i by OD_i in Equation (7) for Delay_i
6              Set t_i,EST to t_break
7          end for
8          Select a task T_i from the tasks if s_i(t_break) = Released
9      else
10         Select a task
11     end if
```

The maximum L value that satisfies Equation (12) cannot be less than $p_1 - c_1$. However, because we already showed that the maximum L for Equation (10) cannot be greater than $p_1 - c_1$, we can tell that the maximum L value for Equation (10) is bounded by the range of t, as defined in Equation (13). As a result, $Laxity_\infty$ can be decided as the maximum L value that satisfies Equation (12), and can be calculated at a constant time.

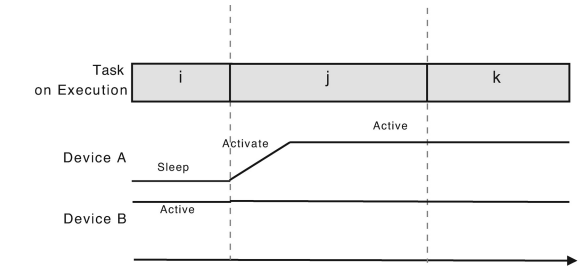
$$\forall t,\ p_1 \le t \le \frac{p_1 - c_1}{1 - \sum_{i=1}^{n} \frac{c_i}{p_i}} : \quad t \ge \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor \cdot c_i + \boldsymbol{L} \tag{13}$$

Algorithm 5 is a brief pseudocode for delaying execution. Here, $t_{i,EST}$ represents the earliest schedulable time of T_i. T_i cannot be executed before $t_{i,EST}$ though it is ready and processor is idle. $t_{i,EST}$ is needed to insert idle time and schedule device power state. In most cases, $t_{i,EST}$ is initialized to the release time of the task. $t_{i,EST}$ needs to be readjusted at runtime for runtime device scheduling because a task cannot be scheduled if devices needed for its execution are not ready. OD_i has to be updated after every delaying of T_i because L_i has to be reused for getting the next t_{break}.

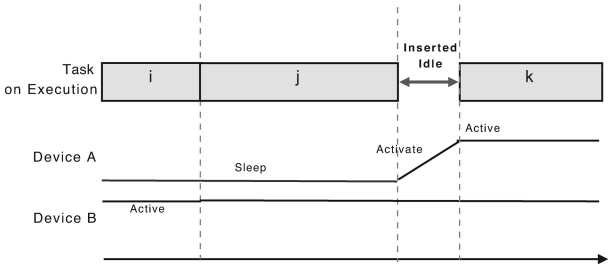
5.4 Inserting Idle Periods for Postponing Device Activation

We assumed that the scheduling algorithms in our target system are non-idling. However, in cases where the actual execution times (ACETs) of tasks may be shorter than their WCETs, idle periods occur that were unexpected in the scheduling stage. As long as the required devices of the next task are ready in active state and the next task is released, the idle periods can be utilized by expediting the starting time of the next task in a cycle-conserving manner to improve the energy efficiency of the processor [Pillai and Shin 2001]. However, considering the energy efficiency of I/O devices, we can put an intentional idle period for preparing the devices in the RDS of the next task, and that will reduce the time for the devices waiting in the active state.

Figure 6 shows a representative example. Assume that both T_i and T_k require device B, and only T_k uses device A. Also, neither device is used by T_j. As in Figure 6(a), device A should be activated before T_j, since there is no time to prepare between T_j and T_k. But if we can insert an idle period after T_j



(a) Original nonidling schedule.



(b) Schedule after inserting idle period.

Fig. 6. Extended sleep period from inserted idle period.

that is longer than the activating time of device A, activating device A can be postponed until the end of T_j, as shown in Figure 6(b).

However, postponing the device activation based on the blindly optimistic prediction that the next task will be finished earlier than its WCET may bring deadline misses. If we can guarantee that inserting an idle period longer than the longest activation time of the required devices of the next task does not break deadlines in all cases, it will be safe to assume that an idle period of that length will go between the former task and the latter task. In the case where the former task is finished earlier than its WCET as expected, we can save energy by utilizing the remaining time for preparing the devices required by the latter task. In constrast, if the former task is on its worst-case execution path, the devices for the next task will be activated in the inserted idle time and no deadlines will be broken.

Delaying the start of a task execution by inserting an idle period before the task is scheduled may result in an increase of energy consumption for some devices. For example, the time for device B in active state may be longer for the inserted idle period in Figure 6(b) than that in Figure 6(a). Therefore, extended active time of device B, in this case, becomes the cost for extending sleep time of device A. Depending on the properties and numbers for different power states of those devices, the schedule in Figure 6(b) may consume more energy than the schedule in Figure 6(a).

The change of energy consumption in I/O devices from inserting the idle period largely depends on the execution time of the task preceding the inserted idle period. For example, if T_j is finished earlier than its WCET by the length of inserted idle period in Figure 6(b), postponing the activation of A will not

cost any more energy consumption from B. Therefore, to analyze the benefit from inserting an idle period, we introduce a function $EET(T_i)$ that returns the expected execution time of T_i. Naturally, $EET(T_i)$ depends on the random distribution, which the actual execution time of T_i follows. Therefore, we cannot define the function EET formally in this stage. For example, in the evaluation of our research, because we assume that ACET of a task set follows a uniform distribution. the mean value between the predefined minimal execution time of T_i and c_i is used for $EET(T_i)$ in our evaluation.

If the difference between the ACET and the WCET of the task before the inserted idle period is less than the length of the inserted idle period, the performance state-changing schedules of all the devices after the task following the inserted idle period can be affected. However, the effects are unpredictable as long as we do not know the ACET in a priori. Therefore, we simplify the analysis of the benefit by considering only the changes before the end of the inserted idle period. This is a greedy approach, based on the optimistic expectation that the task before the inserted idle period will be finished earlier than its WCET at least by the length of the inserted idle period.

Assume that we put an idle period between T_j and T_k, which are consecutively scheduled tasks, and that T_j precedes T_k. In this case, all the devices that have changes in their power state due to the inserted idle period belong to ψ_k. Those devices can also be categorized into two groups: $\{\kappa_m | \kappa_m \in \psi_j \cap \psi_k\}$ and $\{\kappa_m | \kappa_m \notin \psi_j \wedge \kappa_m \in \psi_k\}$.

The devices belonging to the former group could not sleep in the original schedule. However, inserting an idle period that is longer than the sum of both deactivating and activating time of a device in the former group enables the device to sleep within the idle period. Naturally, the shorter idle period in which a device cannot sleep and awaken does not produce any additional power-state transition for the device. This is represented in Equation (14).

$$E_{on-on}(\kappa_m, t_{idle}) = \begin{cases} t_{idle} PW_{a,m}, & \text{for } t_{idle} < t_{wu,m} + t_{sd,m} \\ Min(t_{idle} PW_{a,m}, (t_{idle} - t_{sd,m} - t_{wu,m}) PW_{s,m} & \\ + t_{sd,m} PW_{sd,m} + t_{wu,m} PW_{wu,m}), & \text{for } t_{idle} \geq t_{wu,m} + t_{sd,m} \end{cases} \tag{14}$$

For the latter group belonging to only ψ_k, if the inserted idle period is longer than the activation time of a device in it, the activation of the device can be postponed. However, if the inserted idle period is shorter than that of the device, it should be activated before T_j. In other words, inserting an idle period may extend the time that the device is in active state in that case. Equation (15) expresses the energy consumption of the devices in the latter group.

$$E_{off-on}(\kappa_m, t_{idle}, T_i) = \begin{cases} t_{wu,m} PW_{wu,m} + (EET(T_i) + t_{idle} - t_{wu,m}) PW_{a,m}, & \\ \qquad \text{for } t_{idle} < t_{wu,m} & \\ t_{wu,m} PW_{wu,m} + (EET(T_i) + t_{idle} - t_{wu,m}) PW_{s,m}, & \\ \qquad \text{for } t_{idle} \geq t_{wu,m} & \end{cases} \tag{15}$$

Equation (16) is the aggregation of Equations (14) and (15). It means the total energy consumption of the devices have changes in their power-state schedule

Algorithm 6. Delaying Transition

```
1  On Selecting Tasks:
2  Select T_next based on Algorithm 3
3  t_idle ⟵ FindOptIdle(T_last, T_next)
4  if t_idle > 0 then
5      if isFeasible(t_idle) = Max(∀κ_m∈ψ_next t_wu,m) then
6          T_next ⟵ T_idle
7          c_idle ⟵ t_idle
8      end if
9  end if
```

before the starting point of T_k due to the inserted idle period. Therefore, a desirable idle period length is the one that minimizes Equation (16).

$$\sum_{\kappa_m \notin \psi_j, \kappa_m \in \psi_k} E_{off-on}(\kappa_m, t_{idle}, T_j) + \sum_{\kappa_m \in \psi_j, \kappa_m \in \psi_k} E_{on-on}(\kappa_m, t_{idle}) \tag{16}$$

Here, $Laxity_\infty$ is used again for calculating the safe bound of idle period length to be inserted. In scheduling tasks, we can insert idle time t_{idle} between executions of tasks if the laxity value of T_∞ and all other ready tasks are greater than t_{idle} at the time point of idle time insertion.

This can be simply proved. We can divide T_∞ into several subtasks, $T_{\infty-1}$, $T_{\infty-2}, \ldots, T_{\infty-m}$, which have deadline $\infty + 1, \infty + 2, \ldots, \infty + m$, respectively. The execution time of these tasks are assumed to be arbitrarily assigned on the condition that the sum of execution time of these tasks is the value of laxity T_∞. This meets the feasibility condition of nonpreemptive EDF scheduling regardless of the execution time assignment. Therefore, at any time, at any time, idle time t_{idle} if no ready tasks has laxity value smaller than t_{idle} including T_∞.

In implementing the delayed transition and a lookahead scheduling of T_k, we need to decide the length of an idle period to be inserted, not the fixed time point to start T_k, because the execution time of T_j varies at runtime. To ensure a fixed period of time between T_j and T_k, we insert a virtual idle task T_{idle} having a fixed execution time of t_{idle}. T_{idle} uses no device and its priority is always the lowest one. That means that the execution of T_{idle} consumes the laxity values of all other tasks including T_∞. The rough algorithm of delayed transition is presented in Algorithm 6.

In Algorithm 6, *FindOptIdle* is a function that returns the optimal length of idle period to be inserted, which minimizes Equation (16). To simplify the calculation for getting t_{idle} that minimizes Equation (16), we consider only the cases with meaningful candidates of t_{idle}. The meaningful candidates are idle periods that change either Equation (14) or Equation (15). Therefore, we can define the candidates of t_{idle} in Equation (16) to $\{t_{wu,m} | \kappa_m \notin \psi_j \wedge \kappa_m \in \psi_k\} \cup \{t_{sd,m} + t_{wu,m} | \kappa_m \in \psi_j \wedge \kappa_m \in \psi_k\}$. With these restricted candidates, the complexity to produce the idle length of the minimum value of Equation (16) becomes $O(m^2)$ when there are m devices in the system.

Because the scheduler looks only one scheduling step ahead, T_{next} cannot be ensured to be scheduled after T_{idle}. Therefore, in scheduling the task after T_{idle}, the scheduler makes an assumption that T_{idle} uses the same RDS as T_{last} and

makes T_{last} likely to be scheduled after T_{idle}. For the case when T_{next} cannot be scheduled after T_{idle}, the inserted idle time between T_{last} and T_{next} must be able to be extended to the maximum activation time of the device set. The scheduler should check this as in line 5 of Algorithm 6.

6. EVALUATION

6.1 Evaluation Environment

In saving energy, the performance of a real-time scheduling algorithm can be affected by some parameters such as processor utilization, task set size, ACET/WCET ratio, and device characteristics. We conducted some simulations to analyze the effects of these parameters on the performance of the suggested scheduling algorithms. For comparison, we measured energy efficiency using the algorithms described as follows:

—*LEDES:* A device-scheduling algorithm introduced in Swaminathan and Chakrabarty [2003]. The input task schedules were assumed to be generated by EDF algorithm and not changed even by the dynamic slack from early finishes of tasks.
—*OLEDES:* An enhanced variation of LEDES that starts tasks as soon as possible to utilize dynamic slack.
—*DCS: Device-Conscious Scheduling*
—*DCSDE: Device-Conscious Scheduling* with *Delaying Execution*
—*DCSDT: Device-Conscious Scheduling* with *Delaying Transition*
—*DCSDEDT: Device-Conscious Scheduling* with *Delaying Execution* and *Delaying Transition*
—*OFFOPT:* Offline optimal power-state management on the assumption that all the power-state transitions are done instantly without any power consumption
—*ONOPT:* Online optimal power-state management on the assumption that all the power state transitions are done instantly without any power consumption and a device is in active state only when it is needed

All the results introduced in this section are normalized values to that of OFFOPT unless mentioned otherwise.

OLEDES is an enhanced version of LEDES to cope with utilizing runtime dynamic slack. In LEDES, the task schedule is stored as a table and every task starts at the time point, which is fixed at offline. Therefore, it cannot run a task before its offline schedule, even though it is ready and the processor is idle. However, in OLEDES, task schedules are changed at runtime and every task can start when the processor is idle and the task having the highest priority is ready. Therefore, OLEDES starts a task as soon as possible in a case when a previous task completes its execution before its WCET. We evaluated these algorithms with the same workload where each task has a variable execution time and tasks are scheduled by EDF.

Table I. Device Parameters Used for Evaluation

Device Type	$PW_a(W)$	$PW_{sd} = PW_{wu}(W)$	$PW_s(W)$	$T_{sd} = T_{wu}(s)$
HDD	2.3	1.5	1.0	0.6
NIC	0.3	0.2	0.1	0.5
DSP	0.63	0.4	0.25	0.5

EDS [Swaminathan and Chakrabarty 2005], which generates the energy-optimal task schedule for I/O devices in offline, is impractical when scheduling task sets having a hyperperiod longer than 100 units of time due to its complexity. Therefore, we substitute an ideal lower bound of energy consumption that can be obtained from offline-based DPM techniques for OFFOPT. The ideal lower bound of energy is the energy consumption that occurs when devices have no transition time. This means that a device can be switched off when it is not used and switched on when it is to be used. The ideal energy consumption of OFFOPT is computed using Equation (17).

$$Energy_{OFFOPT} = \sum_{\forall k_i \in \kappa} \left(\sum_{\forall T_j : k_i \in \psi_j} \frac{c_j}{p_j} \cdot PW_{a,i} + \left(1 - \sum_{\forall T_j : k_i \in \psi_j} \frac{c_j}{p_j} \right) \cdot PW_{s,i} \right) \cdot (\text{Exec. time}) \quad (17)$$

Energy consumption of EDS cannot be less than $Energy_{OFFOPT}$, defined in Equation (17), because it uses a time table saved in memory to control the power state of devices, considering device deactivation and activation latencies. Therefore, the original EDS algorithm is inferior in using dynamic slack compared to OFFOPT.

ONOPT is the result for the online version of OFFOPT. All the devices are assumed to become inactive instantly when it is not necessary. ONOPT gains more energy saving than OFFOPT by putting the devices, which were used by the early finished tasks, into the inactive state in the slack time when ACETs are different from WCETs.

Each task in the evaluated task sets uses one or more out of the three I/O devices. The devices and their parameters used in the evaluation are listed in Table I. These I/O devices are a representative set of devices commonly used in embedded applications and are also used for evaluation in Swaminathan and Chakrabarty [2003, 2005].

The device-state transition delays are randomly chosen. But, they could be set only up to 70% of minimum WCET of the running task set unless otherwise stated. The required devices for a task were randomly decided at the deploying stage. The following evaluation are averaged results from 100 experiments.

6.2 Results for Hypothetical Task Sets

We first evaluated with several randomly generated task sets. The WCETs of tasks in those task sets were generated with uniform distribution between 10 and 100 unit processor cycles. Their periods are between 500 and 5,000 unit processor cycles. A task has processor utilization between 0.1% and 10%, which is its WCET divided by its period. A task set has 20 tasks unless otherwise mentioned.

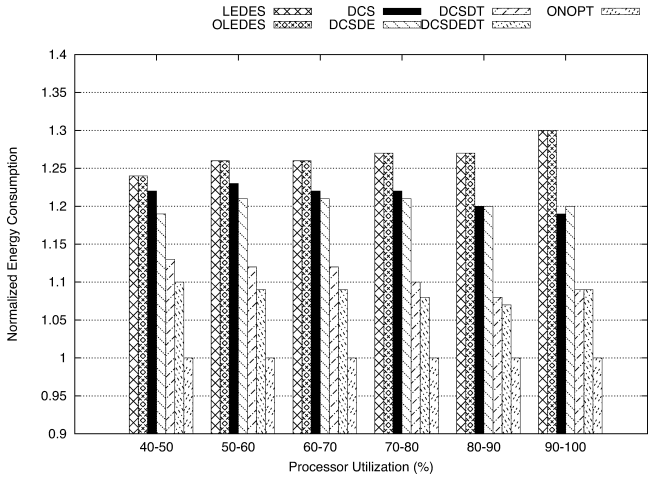


Fig. 7. Normalized energy consumption of random task sets with varying processor utilization (ACET/WCET ratio = 1).

Among the many parameters that affect the evaluation results, processor utilization, number of tasks, and ACET/WCET ratio have major roles. Therefore, we conduct experiments with varying values of these factors.

First, we evaluate with varying processor utilization. Figure 7 shows the result. Since the actual execution time is always the same as the WCET, OLEDES performs exactly the same as LEDES in this evaluation.

Tasks in a task set with low processor utilization tend to have a low WCET to period ratios and, therefore, idle periods usually separate execution of tasks. In this case, merging idle periods by delaying execution may improve the energy efficiency. Therefore, DCSDE and DCSDEDT show better energy efficiency than the others. Also, since the idle periods are likely to be long, LEDES can provide more chances to save energy. Therefore, LEDES and naturally OLEDES show good energy efficiency for the task sets having low processor utilization.

We could verify that as the processor utilization of task sets grows, the effectiveness of Device-Conscious Scheduling improves. The higher the processor utilization is, the more tasks wait to be scheduled at a certain scheduling time. This means increased chances for reordering tasks, resulting in enhcnaed performance of Device-Conscious Scheduling.

However, DCSDE consumed more energy than DCS under high processor utilization, since high processor utilization original schedules barely have idle periods. Moreover, delaying execution of a task reduces laxity values and, therefore, becomes an obstacle for remaining tasks to be reordered later.

The effectiveness of Delaying Transition improved as the processor utilization increased. Since the number of tasks in a task set is fixed, the task sets with high processor utilization tend to have tasks with a long WCET. Thus, inserting an idle period of adequate length will postpone the activation of devices longer in high load than low load. Also, the possibility that an idle period for activating the devices before a task execution exists is higher

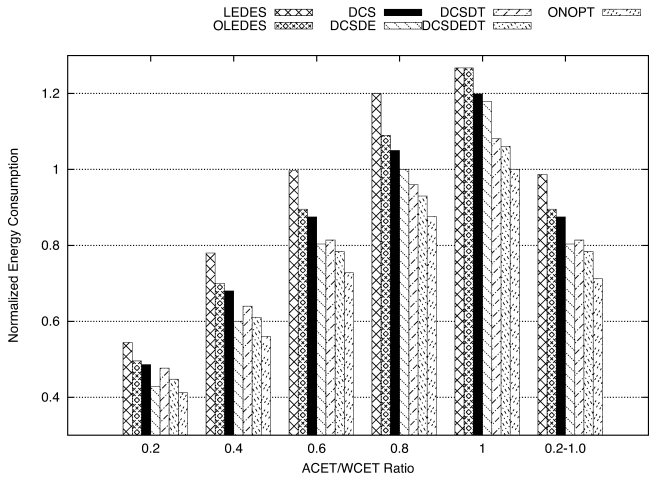


Fig. 8. Normalized energy consumption of random task sets with varying ACET/WCET ratio.

under low processor utilization than high processor utilization, and this will make Delaying Transition less effective for low processor utilization task sets.

Next, we evaluated using varying ACET/WCET ratios. The ACET/WCET ratio affects the amount of dynamic slack as well as the laxity value. The target task set used in this evaluation have processor utilization between 0.78 and 0.82. The ACETs follow uniform distribution with different average values for each experiment. The expected execution time for Delaying Transition of a task is set to 0.6 of its WCET.

Figure 8 shows the evaluation results. All heuristic algorithms that utilize the dynamic slack save more energy as the ACET/WCET ratio decreases and the difference between ONOPT and the others slightly increases as the ratio increases because there may be more chance for the heuristics to deactivate unused devices when the ratio becomes lower, which means the idle length is longer.

Low ACET/WCET means that a large dynamic slack was produced and, therefore, additional idle periods occur, which could not be expected at the scheduling stage. By merging those idle periods, DCSDE and DCSDEDT show better energy efficiency than the others.

As expected, Delaying Transition works well when execution times of tasks are much longer than the device activations times. If a task was finished earlier than the expectation and the actual execution time was near the device activation time for the next task, the activation time would overlap with the finished task, and the next task can be started right after the previous task. However, if the activation was postponed with Delaying Transition, the next task should be delayed until the activation is finished, and reduce the laxity value. Therefore, the chance for reordering scheduling sequences for improved energy efficiency will be decreased. As a result, under low ACET/WCET DCSDT and DCSDEDT consume more energy than DCS and DCSDE.

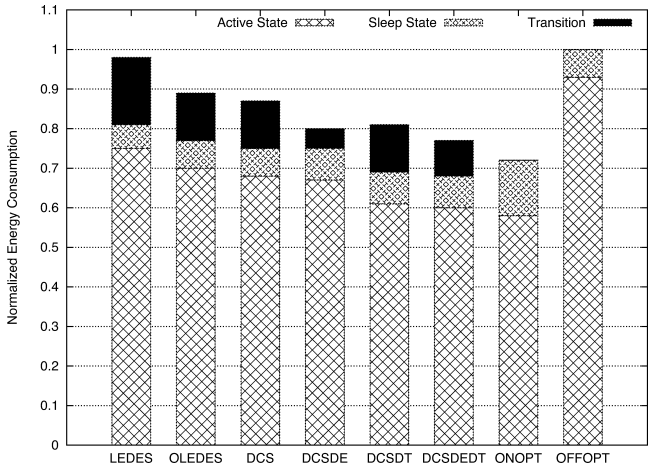


Fig. 9. Constitution of energy consumption for random task sets with random ACET/WCET from 0.2 to 1.0 normalized to OFFOPT.

In contrast, with high ACET/WCET, each task takes a longer execution time relative to device transition time. Thus, DCSDT and DCSDEDT show better energy efficiency than the other heuristics.

The efficiency of delayed execution decreases as dynamic slack decreases. DSCDEDT shows the best energy efficiency among the heuristic algorithms because it holds the benefits from both Delaying Execution and Delaying Transition.

Figure 9 shows the constitution of total energy consumption when the ACET/WCET ratio varies randomly between 0.2 and 1.

We did not count transition energy for OFFOPT since the transition is assumed to be done instantly in OFFOPT, as previously mentioned.

OLEDES consumes less energy in the active state than LEDES because it can run tasks as soon as possible when a previous task completes earlier than its WCET. Therefore, OLEDES can also reduce the number of state transitions.

Energy consumption in the active state with DCS shows that reordering task sequences reduced the time that the devices are in active state by making continuous use of the devices.

DCSDE and DCSDT consume almost the same energy in total, but the constitution was different. The energy consumed in the transition state under DCSDE was the least among that of the others, as expected, by removing the redundant state transitioning for many short idle periods. Also as expected, DCSDT saved energy in the active state mostly by postponing the activation of devices.

The energy for sleep and active states under DCSDEDT was almost the same as for that under DCSDT. But the energy for transition was reduced by Delaying Execution. However, the amount of reduction in transition energy was not as much as DCSDE, since Delaying Transition also consumes the laxity, which was required to delay the execution of tasks.

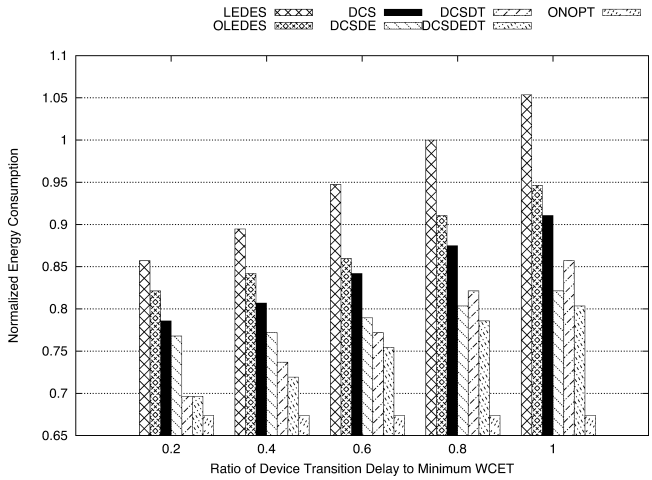


Fig. 10. Normalized energy consumption of random task sets with varying device transition time.

We also evaluate with varying the power-state transition time of devices. The processor utilization of the task sets used in the evaluation were between 0.78 and 0.82. The ratio of the maximum power-state transition time to the minimum WCET of the task set in execution varied from 0.2 to 1.0. The power-state transition time of each device are decided at the deploying stage and maintained during the run.

Figure 10 shows the evaluation results. All heuristic algorithms worked poorly with long power-state transition time. This tendency is more clearly shown with LEDES. The idle periods tend to be short compared to the others, since there is no schedule reordering and merging idle periods under LEDES. Thus with long transition time, a device hardly sleeps.

The execution time of a task becomes relatively shorter when the state transition time of devices gets longer. As shown in the previous results, Delaying Transition is not effective for the task sets having short-term tasks. Naturally, the effectiveness of DCSDT and DCSDEDT become worse as the power-state transition delay increases.

In contrast to Delaying Transition, Delaying Execution performs well with long power-state transition time delays. In the case with long transition time, many idle periods that are too short for changing the power states can be merged into one long idle period, in which the devices can change their power states.

Finally, we check the effectiveness of the suggested algorithms varying the number of tasks in a task set to the energy efficiency. The ACET/WCET ratio of a task followed uniform distribution from 0.2 to 1 and it was decided at every period. The number of tasks in a task set was from 10 to 50, and the processor utilization of a task set was between 0.78 and 0.82.

Figure 11 shows the results. If there are a small number of tasks in a task set, the average WCET of tasks will be long compared to task sets with a large number of tasks. Therefore, there are few tasks switching and the chances for

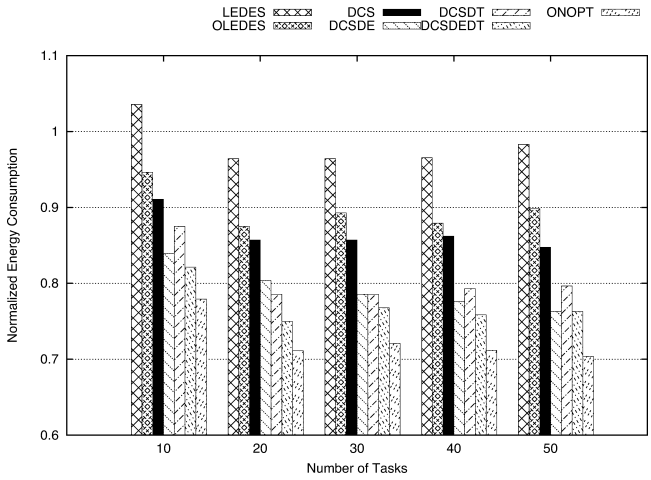


Fig. 11. Normalized energy consumption of random task sets with varying task set size.

utilizing sleep state will be low. With the number of tasks varying from 10 to 20, the energy consumptions under all the algorithms decreased.

However, if the number of tasks in a task set grows larger than a certain threshold, the effectiveness turns to diminishing for all algorithms. Although the sum of all of the idle periods remains the same, the number of idle periods will be increased as the number of tasks grows. Delaying Execution and Delaying Transition change these needless short idle periods into a small number of useful idle periods. Therefore, with DCSDE, DCSDT, and DCSDEDT, the increase of energy consumption by increasing the number of tasks was slower than the others.

6.3 Results for Benchmark Task Sets

For identifying the effectiveness of the suggested heuristic algorithms in real-world applications, we evaluate similar experiments with the generic aviation platform (GAP) [Locke et al. 1991] and computer numerical control (CNC) [Kim et al. 1996] benchmark task sets, both of which represent examples of real-world real-time systems. The deadlines of the tasks are assumed to be identical to their periods.

The GAP task set has a fixed number of tasks and processor utilizations. Therefore, we measured the effectiveness of the algorithms only, with varying ACET/WCET values between 0.2 and 1.

Figure 12 shows performance evaluation results using the GAP task set when all the tasks are released at the same time. The overall properties are similar to Figure 8. Also the constitution of the total energy consumption for the GAP task set is shown in Figure 13.

With the CNC task set, we perform similar experiments that measure the energy consumption of devices, varying the ACET/WCET ratio. As shown in Table II, the tasks in the CNC task set are related to each other, and their periods are multiple times of a base period. Therefore, many tasks are

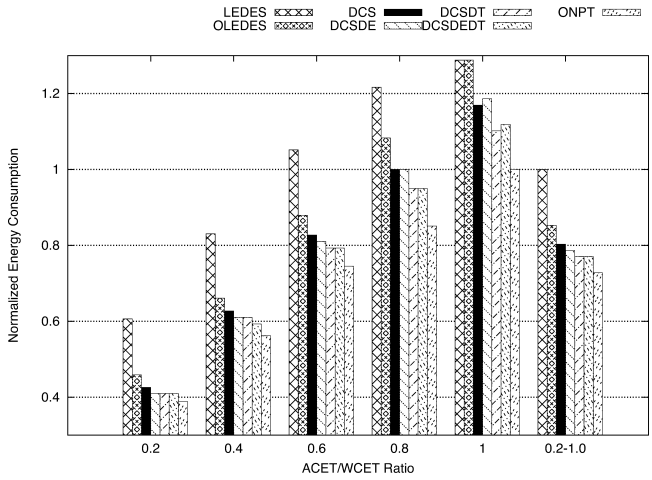


Fig. 12. Normalized energy consumption of GAP task set with varying ACET/WCET ratio.

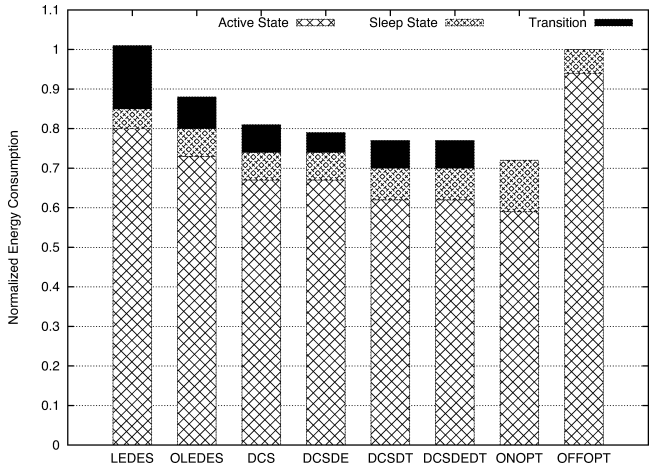


Fig. 13. Normalized energy consumption of GAP task set with random ACET/WCET ratio from 0.2 to 1.

released at the same time. Under the low ACET/WCET condition with scheduling algorithms utilizing dynamic slack, short burst periods and long idle periods occur repeatedly. Thus, the devices can stay in the sleep state for a long time. As a result, including OLEDES, the algorithms using dynamic slack showed good results for the low ACET/WCET values. In addition, the difference between OLEDES and the suggested algorithms is small under the low ACET/WCET values. However, the difference grows as the ACET/WCET increases.

The characteristics of repeating burst and idle periods also affect the constitution of the energy consumption so that the energy consumed in transition was reduced compared to Figure 9.

Table II. Tasks in CNC Task Set

Task	Execution Time	Period
T_{smpl}	35	2400
T_{calv}	40	2400
T_{dist}	180	4800
T_{stts}	720	4800
T_{xref}	165	2400
T_{yref}	165	2400
T_{xctrl}	570	9600
T_{yctrl}	570	7800

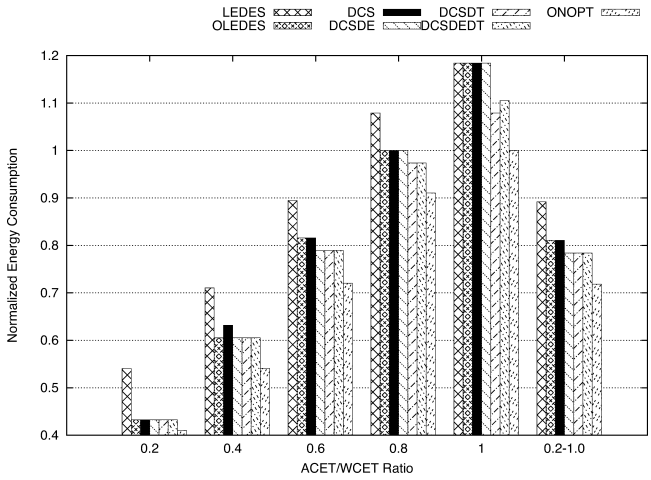


Fig. 14. Normalized energy consumption of CNC task set with varying ACET/WCET ratio.

7. CONCLUSION

The wide use of real-time scheduling in mobile embedded systems makes the energy efficient management of I/O devices a significant issue. Unfortunately task scheduling with the optimal use of I/O devices' DPM feature proved to be infeasible. Therefore, we suggested heuristic algorithms that could transform original EDF-based schedules to schedules with improved energy efficiency, by reordering the task sequences within the short future and delaying the starting of task execution. To determine the validity of the reordered schedule and how long the starting points of tasks could be delayed, our heuristic algorithms use the runtime feasibility check algorithm, which was suggested in our previous work.

As a result, three heuristics were suggested. The first one is to reorder the task schedule to minimize the differences of the required devices sets between two consecutively scheduled tasks. The second one is to postpone the start of the task execution. With this approach, short dispersed idle periods are expected to be merged into one long idle period, which means longer sleep with less power-state transitions. The last one is inserting a certain length of idle period between two consecutive tasks. The devices required for the latter task

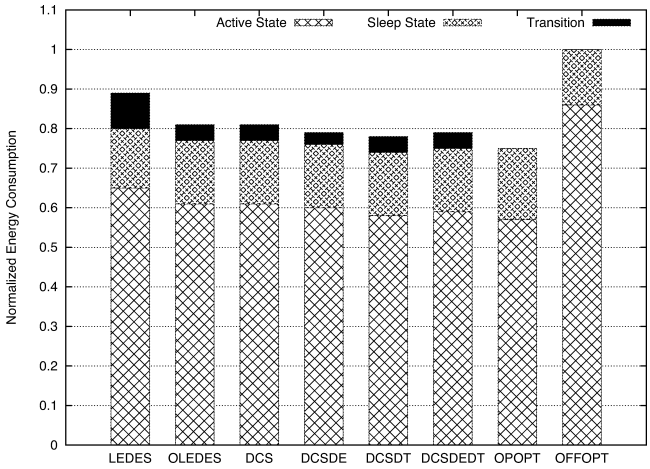


Fig. 15. Constitution of energy consumption of CNC task set with random ACET/WCET ratio from 0.2 to 1 normalized to OFFOPT.

can be activated in the inserted idle period, not in the scheduling time before the start of the former task.

All the suggested heuristics are able to be easily combined with each other or with one of the existing DVS algorithms. We evaluated some combinations of the suggested heuristics with careful simulation with the hypothetical task sets, as well as with the real-world task sets, and showed that our approaches perform better than the existing comparable algorithms.

We expect that the suggested heuristics, combined with the existing CPU-centric energy efficient scheduling heuristics, will produce a synergistic effect, and dramatically improve the energy efficiency of real-time systems.

Our aim was saving the energy consumption of peripheral devices. As our future work, we will explore the nonpreemptive real-time scheduling heuristics for reducing the power consumption of the overall system by considering the power consumption of peripheral devices along with that of processors. In the near future, multicore processors will be widely used in the real-time systems. Although using multicore processors makes the problem even more complex, the power-aware scheduling algorithm with consideration of the power consumption of multicore processors together with peripheral devices will be absolutely necessary for the future real-time systems.

REFERENCES

BARUAH, S. 2005. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. IEEE, Los Alamitos, CA.

BARUAH, S. K. AND CHAKRABARTY, S. 2006. Schedulability analysis of non-preemptive recurring real-time tasks. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA.

BENINI, L., BOGLIOLO, A., PALEOLOGO, G. A., AND MICHELI, G. D. 1999. Policy optimization for dynamic power management. *ACM Trans. Des. Autom. Electron. Syst. 18*, 6, 813–833.

CHENG, H. AND GODDARD, S. 2006a. EEDS NR: An online energy-efficient I/O device scheduling algorithm for hard real-time systems with non-preemptible resources. In *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE, Los Alamitos, CA, 251–260.

CHENG, H. AND GODDARD, S. 2006b. Online energy-aware I/O device scheduling for hard realtime systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM, New York, 1055–1060.

CHUNG, E.-Y., BENINI, L., AND MICHELI, G. D. 1999. Dynamic power management using adaptive learning tree. In *Proceedings of the 1999 International Conference on Computer-Aided Design*. IEEE, Los Alamitos, CA, 274–279.

DOLEV, S. AND KEIZELMAN, E. 1999. Non-preemptive real-time scheduling of multimedia tasks. *J. Real Time Syst. 17*, 23–39.

GOLDING, R. A., II, P. B., STAELIN, C., SULLIVAN, T., AND WILKES, J. 1995. Idleness is not sloth. In *Proceedings of the Winter Conference*. USENIX, Berkley, CA, 201–212.

HOWELL, R. R. AND VENKATRAO, M. K. 1995. On non-preemptive scheduling of recurring tasks using inserted idle times. *Inf. Comput. 117*, 1, 50–62.

HWANG, C.-H. AND WU, A. C.-H. 2000. A predictive system shutdown method for energy saving of event-driven computation. *ACM Trans. Des. Autom. Electron. Syst. 5*, 2, 226–241.

IRANI, S., GUPTA, R., AND SHUKLA, S. 2002. Competitive analysis of dynamic power management strategies for systems with multiple power savings states. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, Los Alamitos, CA, 117.

JEFFAY, K., STANAT, D. F., AND MARTEL, C. U. 1991. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 129–139.

JEJURIKAR, R. AND GUPTA, R. 2005a. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the 42nd Annual Conference on Design Automation*. IEEE, Los Alamitos, 111–116.

JEJURIKAR, R. AND GUPTA, R. K. 2005b. Energy aware non-preemptive scheduling for hard real-time systems. In *Proceedings of 17th of Euromicro Conference on Real-Time Systems*. IEEE, Los Alamitos, CA, 21–30.

KIM, N., M.RYU, HONG, S., SAKSENA, M., CHOI, C., AND SHIN, H. 1996. Visual assessment of a real-time system design: Case study on a cnc controller. In *Proceedings of the Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 300–310.

KIM, S., LEE, J., AND KIM, J. 2006. Runtime feasibility check for non-preemptive real-time periodic tasks. *Inf. Process. Lett. 97*, 3, 83–87.

KIM, W., KIM, J., AND MIN, S. 2002. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE, Los Alamitos, CA, 788.

LOCKE, D., VOGEL, D., AND MESLER, T. J. 1991. Building a predictable avionics platform in ada. In *Proceedings of the Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 181–189.

LU, Y.-H., BENINI, L., AND MICHELI, G. D. 2000a. Low-power task scheduling for multiple devices. In *Proceedings of International Workshop on Hardware/Software Codesign.* ACM, New York, 39–43.

LU, Y.-H., BENINI, L., AND MICHELI, G. D. 2000b. Operating-system directed power reduction. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 37–42.

LU, Y.-H. AND MICHELI, G. D. 2001. Comparing system-level power management policies. *IEEE Des. Test Comput. 18*, 2, 10–19.

PIAGGIO, M., SGORBISSA, A., AND ZACCARIA, R. 2000. Pre-emptive versus non-preemptive real-time scheduling in intelligent mobile robotics. *J. Exp. Theor. Artif. Intell. 12*, 2, 235–245.

PILLAI, P. AND SHIN, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles.* ACM, New York, 89–102.

SEO, E., JEONG, J., PARK, S., KIM, J., AND LEE, J. 2009. Catching two rabbits: Adaptive real-time support for embedded Linux. *Software: Practice and Experience*.

SIMUNIC, T., MICHELI, G. D., AND BENINI, L. 1999. Event-driven power management of portable systems. In *Proceedings of the International Symposium on System Synthesis*. IEEE, Los Alamitos, CA, 18–23.

SWAMINATHAN, V. AND CHAKRABARTY, K. 2003. Energy-conscious, deterministic i/o device scheduling in hard real-time systems. *IEEE Trans. Comput. Aided Des. Integr. Circuit Syst. 22*, 7, 847–858.

SWAMINATHAN, V. AND CHAKRABARTY, K. 2005. Pruning-based, energy-optimal, deterministic i/o device scheduling for hard real-time systems. *ACM Trans. Embedded Comput. Syst. 4*, 1, 141–167.

Received October 2008; revised February 2009; accepted June 2009