View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@University of Nebraska

University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

CSE Technical reports

Computer Science and Engineering, Department of

2004

A Dynamic Real-time Scheduling Algorithm for Reduced Energy Consumption

Rohini Krishnapura
University of Nebraska-Lincoln, rohini@cse.unl.edu

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

Ala' Adel Qadi
University of Nebraska-Lincoln, aqadi@cse.unl.edu

Follow this and additional works at: https://digitalcommons.unl.edu/csetechreports

Part of the Computer Sciences Commons

Krishnapura, Rohini; Goddard, Steve; and Qadi, Ala' Adel, "A Dynamic Real-time Scheduling Algorithm for Reduced Energy Consumption" (2004). *CSE Technical reports*. 72.
https://digitalcommons.unl.edu/csetechreports/72

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

A Dynamic Real-time Scheduling Algorithm for Reduced Energy Consumption

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

Technical Report TR-UNL-CSE-2004-0009
May 2004

Abstract

In embedded real-time systems, Dynamic Power Management (DPM) techniques have traditionally focused on reducing the dynamic power dissipation that occurs when a CMOS gate switches in a processor. Less attention has been given to processor leakage power or power consumed by I/O devices and other subsystems. I/O-based DPM techniques, however, have been extensively researched in non-real-time systems. These techniques focus on switching I/O devices to low power states based on various policies and are not applicable to real-time environments because of the non-deterministic nature of the policies. The challenge in conserving energy in embedded real-time systems is thus to reduce power consumption while preserving temporal correctness. To address this problem, we introduce three scheduling algorithms of increasing complexity: Energy-Aware EDF (EA-EDF), Enhanced Energy-Aware EDF (EEA-EDF) and Slack Utilization for Reduced Energy (SURE). The first two algorithms are relatively simple extensions to the Earliest Deadline First (EDF) scheduling algorithm that enable processor, I/O device, and subsystem energy conservation. The SURE algorithm utilizes slack to create a non-work-conserving approach to reducing power consumption. An evaluation of the three approaches shows that all three yield significant energy savings with respect to no DPM technique. The actual savings depends on the task set, shared devices, and the power requirements of the devices. When the cost of switching power states is low, the EA-EDF and EEA-EDF algorithms provide remarkable power savings considering their simplicity. In general, however, the higher the energy cost to switch power states, the more benefit SURE provides.

1. Introduction

Traditionally, power conservation in systems has been implemented via efficient power management using static and dynamic techniques. Static power management techniques are built into the design of a system. Dynamic Power Management (DPM) techniques are applied at run-time, based on workload variation [7]. DPM techniques can be employed through the operating system (OS) to obtain fine-grained control over power management. For example, the Advanced Configuration and Power Interface (ACPI) [1], conceived by Intel, Microsoft and Toshiba, provides a standard interface for the PC platform, through which devices can be controlled for power management by the OS. Earlier, BIOS-based power management techniques were widely used with the OS unaware of any power management policies. The OnNow technology from Microsoft [11] takes advantage of ACPI to provide OS-directed power management techniques through Windows.

OS-directed DPM techniques can be processor-based (CPU-based) or I/O-based. An example of CPU-based DPM is Dynamic Voltage Scaling (DVS), wherein the operating voltage of the CPU is varied to save energy by computing at a lower frequency, thereby trading computational speed for power conservation. I/O-based DPM techniques focus on switching I/O devices into low power states based on predictive, stochastic, or timeout policies [9].

DPM in embedded real-time systems, however, have centered mainly on the CPU with little focus given to I/O devices. CPU-based DPM methods in these systems include real-time scheduling algorithms aimed at reducing the dynamic power dissipation that occurs when a CMOS gate switches, which is the dominant component of power consumed by processors. The other two power components are leakage power and shortcut power. Leakage power is expected to account for an increasing fraction of power consumption in processors of the future [2] and cannot be ignored.

Energy conservation in I/O devices presents an even greater challenge since the approaches developed for processors (e.g., DVS) generally are not applicable. For example, probabilistic power-saving policies for shutting down I/O devices cannot be implemented in hard real-time systems, as jobs are not guaranteed to meet deadlines. The challenge in conserving energy in embedded real-time systems is to reduce power consumption while preserving temporal correctness.

In this paper, we introduce three scheduling algorithms of increasing complexity: Energy-Aware EDF (EA-EDF), Enhanced Energy-Aware EDF (EEA-EDF) and Slack Utilization for Reduced Energy (SURE). These algorithms can be classified as OS-directed DPM techniques applicable to either processor or I/O devices. These techniques do not rely on DVS to save power. Instead, they shutdown the device whenever possible to conserve power. When applied to processors, the goal is to reduce leakage power consumption, though the algorithm also saves the maximum amount of power possible when only frequency can be scaled (and not voltage).

EA-EDF and EEA-EDF are obvious and simple extensions to the Earliest Deadline First (EDF) scheduling algorithm to enable I/O and subsystem energy conservation. We use these algorithms to illustrate the amount of energy savings possible with simple techniques. The main focus of this paper, however, is the SURE algorithm. SURE schedules jobs so that devices can be kept in a given power state for as long as possible thus reducing the device power state transitions and reducing energy consumption, while ensuring that all jobs meet their deadlines.

The general SURE algorithm can be applied to the CPU, co-processors (e.g., digital signal processors), device subsystems, and individual I/O devices. The primary focus of this paper is the fully preemptive version of SURE which applies to processors, of course, but also to preemptive I/O devices such as memory flash cards or Micro-Electro-Mechanical Systems (MEMS) storage devices [3, 19], which are anticipated to be a main focus of secondary storage devices in the near future [6, 10]. For non-preemptive devices, blocking factors for the non-preemptive periods in the task execution due to accessing the device need to be accounted for, which are extensions to the version of SURE presented here. The preemptive SURE algorithm provides the foundation for a family of general, non-work-conserving, online I/O and subsystem energy saving algorithms.

The rest of this paper is organized as follows. Section 2 gives a review of related work and Section 3 describes the energy conservation model assumed. We present the three scheduling algorithms and a necessary and sufficient feasibility condition for the preemptive version of SURE in Section 4. (Non-preemptive and preemptive versions with non-preemptive [blocking] intervals are relatively simple extensions to the version of SURE presented here, but beyond the scope of this paper.) Finally, we present evaluation results in Section 5 and conclude in Section 6.

2. Related Work

This work addresses power consumption in the processor, device subsystems, and I/O devices. Within a processor, there are three primary components to processor power consumption: dynamic power, leakage (static) power, and shortcut power. Dynamic power is currently the largest fraction of three, but it is expected that leakage power will become an increasingly larger fraction [2]. From a processor perspective, this work primarily addresses leakage power reduction, though it also helps to reduce dynamic power consumption in the processor.

Even when CMOS circuits are not switching, they still dissipate leakage power. Leakage power consumption is reduced by disabling all or parts of the processor whenever possible. To the best of our knowledge, Lee et al. [5] present the first earliest-deadline-first (EDF) and rate monotonic (RM) and scheduling algorithms designed exclusively to reduce leakage power in hard real-time systems. These algorithms insert idle intervals in the schedule to delay the transition of the processor from a low power state to the processing power state.

When the CPU is the only device used by the task set, the EA-EDF and EEA-EDF algorithms presented here both reduce to the the EDF-S algorithm of [5]. However, the EA-EDF and EEA-EDF algorithms are more general than the EDF-S algorithm in that they can also save energy in I/O devices. Conceptually, SURE is similar to the leakage-control EDF (LC-EDF) algorithm developed by Lee et al. in [5] in that idle times are inserted into the schedule at key points to keep the CPU idle for as long as possible. However, SURE differs in the job scheduling strategy and the application of the concept to also reduce energy in I/O devices. In addition, we use system slack computation to insert idle time into the schedule, rather than the ad-hoc method used by Lee et al. Since the SURE algorithm uses all available system slack before transitioning the processor from a low power state to a normal power state, it will always save at least as much leakage power as the LC-EDF algorithm.

The approach taken in this work to reduce leakage power consumption is to view the processor as a device with two states: idle and active. (In this work, we do not distinguish between idle and sleep states. The idle state is the lowest power state available that still meets the application requirements.) Thus, from this perspective, reducing leakage power consumption is equivalent to reducing energy consumption in I/O devices—or device subsystems, depending on the platform.

Most DPM techniques for shared 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., [7, 8, 9, 4, 20]). These strategies cannot be directly applied to real-time systems because of their non-deterministic nature. Nonetheless, the non-real-time scheduling algorithm for energy conservation in I/O devices by Lu, Benini, and Micheli presented in [9] is similar to the approach presented here. Their scheduling technique also rearranges jobs so as to minimize the device state switches. Our approach, however, differs in that it is deterministic and ensures temporal correctness of the system.

To the best of our knowledge, the first I/O-based technique for real-time systems is the Low Energy Device Scheduler (LEDES), by Swaminathan and Chakrabarty [14]. LEDES takes as input a pre-determined task schedule and a device-usage list for each task to generate a sequence of sleep/working 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. However, LEDES differs from SURE in that jobs are not rearranged to reduce energy consumption. SURE relies on the observation that under some conditions, jobs can be rearranged to facilitate energy conservation and still meet their deadlines.

The pruning-based scheduling algorithm, Energy-optimal Device Scheduler (EDS), is different from LEDES and similar to SURE in that jobs are rearranged to find the minimum energy task schedule [15]. EDS generates a schedule tree by selectively pruning the branches of the tree. Pruning is done based on both temporal and energy constraints. A drawback of EDS is that it generates only non-preemptive energy optimal schedules. Thus EDS is successful only for task sets which have at least one feasible non-preemptive schedule. SURE removes this drawback by providing the ability to generate preemptive schedules, which in some cases consume less energy than a non-preemptive schedule—provided that the I/O device or subsystem is at least partially preemptive. In addition, the algorithm we present is different from EDS in that EDS is inherently an offline algorithm, with schedules computed statically, whereas SURE can be implemented as an online or offline algorithm. An advantage of generating an online schedule is the flexibility in adapting to changing task parameters. This also has the merit of utilizing dynamic slack which results from actual job execution times being much less than the WCET.

An extension of LEDES to handle I/O devices with multiple power states is presented in [16] by the same authors. Multi-state Constrained Low Energy Scheduler (MUSCLES) takes as input a pre-computed task schedule and a per-task device usage list to generate a sequence of power states switching times for I/O devices while guaranteeing that real-time constraints are not violated. Although MUSCLES has the advantage over SURE to efficiently utilize multiple power states of a device, MUSCLES has the same drawback of LEDES in that jobs are not rearranged to exploit the inherent device dependencies of different tasks.

3. Energy Conservation Model

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

Suppose that the set of devices required by each task during its execution is specified along with the temporal parameters of a periodic task set. More formally, given a periodic task set with deadlines equal to periods, $\tau = \{T_1, T_2, ...T_n\}$, let task T_i be specified by the three tuple (p_i, e_i, Λ_i) where, p_i is the period, e_i is the worst case execution time, and $\Lambda_i = \{\lambda_1, \lambda_2, ...\lambda_m\}$ is the Device Requirement Specification (DRS) for the task T_i. Then, $U_{\lambda_i} = \sum\limits_{\forall T_j, \{\lambda_i\} \subseteq \Lambda_j} (e_j/p_j)$. The generalized problem that we aim to solve in this paper can now be stated as, given this periodic task set $\tau = \{T_1, T_2, ...T_n\}$, $T_i = (p_i, e_i, \Lambda_i)$, is there a schedule which meets all deadlines and also reduces the energy consumed by each device λ_j?

In a hyperperiod (i.e., $\text{lcm}\{p_i\}, 1 \leq i \leq n$), the total time that device λ_i will be used is $U_{\lambda_i} \cdot H$. Consequently, the device is not in use for at least $(1 - U_{\lambda_i}) \cdot H$ time units. For the periodic task model, consider the energy consumed over one hyperperiod. If the device remained *active* over the entire hyperperiod, the total energy consumed would be $E_{orig} = P_{active} \cdot H$, where P_{active} is the rate at which energy is consumed when the device is *active*. Since the device is not in use for at least $(1 - U_{\lambda_i}) \cdot H$, the device does not need to be *active* for the entire hyperperiod. However, significant cost is incurred when an I/O device switches or transitions from one power state to another. This cost is high in terms of both time and energy. Thus by employing a DPM technique, the total energy consumed by a device λ_i in the hyperperiod H, is given by,

$$E_{\lambda_i} = E_{active} + E_{idle} + E_{sw} \tag{1}$$

where, E_{active} is the energy consumed when λ_i is in the *active* state and E_{idle} is the energy consumed by λ_i when it is in the *idle* state and E_{sw} is the energy consumed when λ_i is in transition states. From the discussion above, it should be clear that $E_{active} = P_{active} \cdot U_{\lambda_i} \cdot H$. For simplification, let the time taken to switch from *active* to *idle* and vice-versa be the same. Let us call this switch time t_{sw}. In addition, let the power consumed during both transitions be the same. Let this power be P_{sw}. Then, $E_{sw} = \sigma_i \cdot P_{sw} \cdot t_{sw}$, where σ_i is the total number of device state switches in a hyperperiod. So, the actual time the device is in the *idle* state is $[(1 - U_{\lambda_i}) \cdot H - \sigma_i t_{sw}]$. Thus, $E_{idle} = P_{idle}[(1 - U_{\lambda_i}) \cdot H - \sigma_i \cdot t_{sw}]$, where P_{idle} is the rate at which energy is consumed when the device is *idle*. Substituting for E_{active}, E_{idle} and E_{sw} in Equation (1), the total energy consumed by λ_i in a hyperperiod is,

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

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

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

Thus, to increase energy savings, the time for which the device λ_i is *idle* must be increased whereas the total number of power state transitions (σ_i) must be decreased. However, this is an optimistic equation in that device idle times are assumed to be longer than $2 \cdot P_{sw}$ time units. If a device is idle for less than $2 \cdot P_{sw}$ time units, it will not have sufficient time to switch from the *active* state to the *idle* state and vice-versa. Thus, if a device is idle for less than $2 \cdot P_{sw}$ time units, it should not be switched to the *idle* state. This will ensure that device transitions still preserve temporal constraints.

4. Scheduling Algorithms

This section introduces three OS-directed, real-time, DPM techniques applicable to either processors or I/O devices. The first two methods are relatively simple extensions to EDF scheduling. The third method is more complicated but still based on EDF. Rather than using dynamic voltage scaling (DVS) to save power, the three algorithms shutdown the device whenever possible to conserve energy.

4.1. Simple Extensions to EDF

A simple scheduling algorithm is to extend EDF to conserve energy in devices by switching a device to the low power state whenever the system is idle. This is an obvious extension to EDF and we refer to this facile technique as Energy-Aware EDF (EA-EDF). Observe that with this method there is no need to associate a DRS with each task since devices are put in the low power state only when the CPU is idle, which implies that no device is in use (under the stated assumptions). Energy savings is calculated by considering the amount of time the device is in the *active* state and the *idle* state and the number of device state switches that are incurred with the EA-EDF schedule. When the CPU is the only device used by the task set, the EA-EDF algorithm reduces to the EDF-S algorithm presented by Lee et al. in [5]. The *only* difference between the EA-EDF algorithm and the EDF-S algorithm is that the EA-EDF algorithm switches I/O devices to the low power state in addition to the CPU. This algorithm is presented here to provide a base-line comparison.

Another obvious extension is the Enhanced Energy-Aware EDF (EEA-EDF) algorithm, wherein a device is switched to a low power state whenever it is not in use. This implies that the underlying EDF scheduling algorithm is made aware of the device requirements of the tasks (i.e., a DRS is associated with each task). The EEA-EDF schedules tasks using the EDF algorithm. It improves energy conservation over the EA-EDF algorithm by switching devices to the idle state whenever they are not used by the currently executing task. The device stays in the idle state until a task that uses that device is dispatched. Thus, whenever the CPU is idle the devices used by the task set will be in a low-power state. When the CPU is the only shared device, the EA-EDF and EEA-EDF algorithms generate the exact same schedule and consume the same amount of energy.

EA-EDF and EEA-EDF can be executed as online or offline algorithms. In both cases, it should be clear that the standard utilization test, $U \leq 1$, is a necessary and sufficient condition for the temporal correctness of the preemptive versions of EA-EDF and EEA-EDF, assuming devices are ready whenever the task makes a request.

A more complex algorithm than either of these is the SURE scheduling algorithm, which schedules jobs that require the same device to run in succession. This results in combining small and scattered device idle times to generate device idle times of longer duration. Moreover, slack is used to combine CPU idle times to produce longer intervals of CPU idle time. As with EA-EDF and EEA-EDF, all devices are switched to the *idle* state to save additional power during these CPU idle intervals. The next section describes this more sophisticated approach in detail.

4.2. Slack Utilization for Reduced Energy (SURE)

The SURE algorithm is a non-work-conserving, real-time, scheduling algorithm that is designed to reduce system energy consumption while ensuring the temporal correctness of the application. By non-work-conserving, we mean that the SURE algorithm deliberately inserts idle time in the schedule when there are pending jobs to execute. The SURE algorithm can be executed offline to generate a cyclic schedule for online execution, or it can be executed online for more flexibility—and potentially more energy savings. Of course, the online execution of SURE adds scheduling overhead, and a cost-benefit tradeoff must be made, which will be application specific. Thus, as with most scheduling algorithms, SURE is not a panacea for energy savings. Rather, it should be viewed as another tool available to engineers.

This section provides an introduction to the SURE algorithm. For simplicity, a preemptive version of SURE is presented that provides the foundation for a family of general, non-work-conserving, I/O and subsystem energy saving algorithms. Before presenting the algorithm, however, we first present the concept of slack and its computation, which is used to determine the location and duration of inserted idle intervals in the schedule.

4.2.1. Slack Computation

It is helpful to provide formal definitions of job and system slack since the SURE algorithm utilizes system slack to conserve energy.

Definition 4.1. *Initial Job Slack.* The initial slack of a job J_k, at time $t = 0$, is denoted $\omega_k(0)$ and computed by subtracting the total time required to execute job J_k and other periodic requests with higher priorities than this job from the total time available to execute job J_k. That is, the slack of a job J_k at $t = 0$ is given by

$$\omega_k(0) = D_k - \sum_{D_i \leq D_k} e_i, \text{where } D_k \text{ is the absolute deadline of } J_k.$$

Definition 4.2. *Dynamic Job Slack.* The slack of a job J_k, at t, $\forall t > 0$, is denoted $\omega_k(t)$ and changes dynamically as it gets consumed by CPU idling and by the execution of lower priority jobs. That is, the dynamic slack of a job J_k at time t is given by

$$\begin{aligned}\omega_k(t) &= \omega_k(0) - I(0,t) - \sum_{D_i > D_k, r_i < t} f_i(t) \\ &= D_k - \sum_{D_i \leq D_k} e_i - I(0,t) - \sum_{D_i > D_k, r_i < t} f_i(t)\end{aligned}$$

where $I(0,t)$ is the amount of time the CPU has been idled till t; $\sum_{D_i > D_k, r_i < t} f_i(t)$ is the amount of time jobs with deadlines greater than D_k, have executed till t, which implies that these jobs have to be released before t; and $I(0,t) + \sum_{D_i > D_k, r_i < t} f_i(t)$ is the total amount of slack consumed till t.

Definition 4.3. *System Slack.* The slack of a system at t, $\forall t \geq 0$, is denoted $\Omega(t)$ and is the minimum slack at time t among all the jobs in the hyperperiod. That is,

$$\Omega(t) = \begin{cases} 0 & \exists J_i, D_i > t \text{ and } \omega_i(t) < 0 \\ min(\omega_i(t)) & \forall J_i, D_i > t \text{ and } \omega_i \geq 0 \end{cases}$$

Based on these definitions, the system slack at time t is the maximum amount of time that job execution can be delayed without causing any jobs (both current and future) to miss their deadlines. Henceforth, unless specified explicitly, the term *slack* will be used to refer to the system slack.

If the system slack is computed by looking at the slack of all N jobs in the hyperperiod, the complexity would be O(N). For a more efficient method of slack computation, we use Tia's method of static computation [18]. In Tia's method, the scheduler computes the amount of slack for all periodic requests before runtime and stores them in a table. This pre-computed slack is then adjusted appropriately during runtime.

To compute $\Omega(t)$ efficiently during runtime, periodic requests are grouped into n disjoint sets such that only one request from each set needs to be examined. At any time t, to compute the system slack, jobs that have finished execution need not be considered. Suppose that the current job of each task T_i at the time of slack computation, t, is given by J_{c_i} whose absolute deadline is denoted by D_{c_i}. Without loss of generality, suppose we order the current n jobs, $D_{c_1} < D_{c_2} < \ldots D_{c_n}$. A job in the hyperperiod with deadline after t is grouped into a subset Z_i if its deadline falls in the range $[D_{c_i}, D_{c_{i+1}})$, i.e. between the deadline of the current job of task T_i and of T_{i+1}. The last subset Z_n contains all jobs whose deadlines are equal to or greater than D_{c_n}, i.e. all jobs with deadlines greater than the absolute deadline of the current job of task T_n. This results in grouping all jobs in the hyperperiod among n subsets.

Suppose a low priority job J_{c_j} executes ahead of a higher priority job. Also suppose that J_{c_j} falls in the group Z_i. Then, the portion that is subtracted from the initial slack of a job, is constant for all jobs in the set Z_{i-1}. This is because the absolute deadlines of *all* jobs in Z_{i-1} is less than the absolute deadline of the job J_{c_j} in Z_i. Hence, the same amount has to be subtracted for all these jobs. Thus the job in Z_i with the minimum slack at time t will also be the job with the minimum *initial* slack at $t = 0$. To obtain the minimum slack at any time t, the jobs with minimum slack is each subset Z_i need to be examined. This results in a lookup of n jobs.

Now, for an efficient retrieval of the jobs with minimum slack in each subset, Tia's method is as follows. Jobs $J_1, J_2, \ldots J_N$ are assumed to be arranged in the non-decreasing order of their deadlines. Now, each subset Z_i will contain the jobs $J_{c_i}, J_{c_i+1}, J_{c_i+2} \ldots J_{c_n}$ An N x N table containing the initial slack of all jobs is created at time $t = 0$ with entries as $\chi(i, j)$. An entry $\chi(i, j)$ is the minimum slack of all jobs J_k, for $k = i, i+1, \ldots, j-1, j$. Conceptually, $\chi(i, j)$ is the minimum slack of all jobs with deadlines in the range $[D_{c_i}, D_{c_j}]$. The system slack at time $t = 0$ can be calculated as, $\Omega(0) = \chi(1, N)$. The initial slack of a job J_k is given as $\chi(k, k)$.

This pre-computed slack of jobs in a subset Z_i is updated as slack is consumed by CPU idling and execution of lower priority jobs. The minimum slack at time t, of all jobs in the subset Z_i is computed as,

$$\chi_i(t) = \chi(c_i, c_{i+1} - 1) - I(t) - \sum_{k=i+1}^{n} f_{c_k}(t)$$

$$\forall i = 1, 2, ...n - 1$$

Minimum slack of all jobs in the subset Z_n is given by,

$\chi_n(t) = \chi_n(0) - I(t)$

Thus the slack of the system at t is the minimum slack of all the jobs at time t, given by

$$\Omega(t) = min(\chi_i(t))_{\forall 1 \leq i \leq n}$$

Since, only the minimum slack in n subsets need to be computed, we can compute the system slack with O(n) time complexity.

4.2.2. The SURE Algorithm

With SURE, if a device is in the *active* state, ready jobs requiring the device are executed in succession such that few device state changes occur. Alternatively, if a device is in the *idle* state, the execution of jobs is delayed as long as possible so that the jobs do not miss their deadlines but also allows the device to be in the *idle* state for a longer duration. The heuristic here is that a device state change from *active* to *idle* or vice-versa is delayed as long as possible while still upholding temporal constraints.

For instance, if an I/O device is *idle* and a job requiring that device is released, then if there is system slack at that time, the device is allowed to stay idle till system slack becomes zero. After this, the job has to be executed to meet its deadline. Similarly, suppose a device was *active*, and the job with the nearest deadline, i.e., the highest EDF priority job, did not require the device (henceforth, we use the term priority to mean the priority assigned by the EDF scheduling algorithm). At this time, there could be another lower priority job requiring the same device. If there is slack in the system, the higher priority job could be deferred and the lower priority job is executed till there is no more slack in the system. Now, the higher priority job has to execute to meet its deadline. The overall result of the algorithm is that smaller chunks of device idle times and usage times are grouped together. This results in reducing the total number of state transitions in the hyperperiod. The algorithm is presented in Figure 1.

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

```
scheduler( ):
Initialize at t= 0: {
    $noSlack \leftarrow$ false ;
    $J_{curr} \leftarrow \phi$;
    $B_{curr} \leftarrow 0$;
    $computeSlack \leftarrow$ false;
    $devShare \leftarrow \phi$;
    return;
}
If ($t$: instance when job is released) {
    If ($J_{curr} == \phi$) // the CPU is idle
        $computeSlack \leftarrow$ true
    else
        $computeSlack \leftarrow$ false
    If ($noSlack$) { // the system has no slack
        do_EDF();
        return;
    }
}
If ($t$: instance when job finishes its execution budget) {
    If (job queue is empty) {
        $J_{curr} \leftarrow \phi$; // make CPU idle
        $B_{curr} \leftarrow 0$;
        return;
    }
    $computeSlack \leftarrow$ true; // need to recompute slack
}
If ($computeSlack$) {
    Compute $\Omega(t)$;
    If ($\Omega(t) > 0$)
        do_SURE ();
    else
        do_EDF();
}
```

```
do_EDF(){
    If ($J_{curr} \neq J_{high}$) {
        $devShare \leftarrow \{\Lambda_{T(curr)} \cap \Lambda_{T(high)}\}$; // $devShare$ is the set of devices shared by $J_{curr}$ and $J_{sh}$
        Make devices in $\{\Lambda_{T(curr)} - devShare\}$ idle; // the set of active devices not required by $J_{sh}$ is made idle
        Make devices in $\{\Lambda_{T(high)} - devShare\}$ active; // the set of idle devices required by $J_{sh}$ are made active
        $J_{curr} \leftarrow J_{high}$; // execute the highest priority job
        $B_{curr} \leftarrow e_{high}$;
        $noSlack \leftarrow$ true;
    }
    return;
}
do_SURE () {
    If ($J_{curr} \neq \phi$) {
        // Determine the job $J_{sh}$ which shares the maximum number of devices with $J_{curr}$
        $devShare \leftarrow \{\Lambda_{T(curr)} \cap \Lambda_{T(sh)}\}$;
        If ($|devShare| == 0$) { // no job share any device with $J_{curr}$
            Make devices in $\{\Lambda_{T(curr)}\}$ idle; // Make all devices used by $J_{curr}$ idle
            $J_{curr} \leftarrow \phi$; // Make CPU idle
        }
        else {
        Make devices in $\{\Lambda_{T(curr)} - devShare\}$ idle;
        Make devices in $\{\Lambda_{T(sh)} - devShare\}$ active;
        $J_{curr} \leftarrow J_{sh}$;
        }
    }
    $B_{curr} \leftarrow \Omega(t)$;
    $noSlack \leftarrow$ false;
    return;
}
```

Figure 1. SURE Scheduling Algorithm.

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

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

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

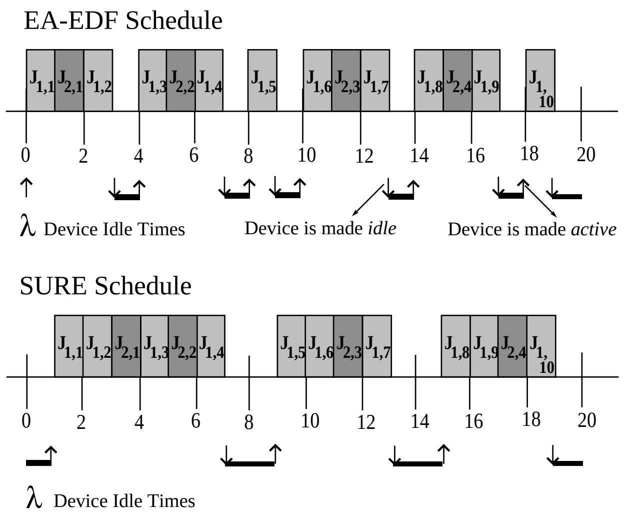


Figure 2. EA-EDF and SURE Schedules for $\{T_1, T_2\}$**,** $T_1 = (2, 1, \{\lambda\})$**,** $T_2 = (5, 1, \{\lambda\})$**.**

For non-preemptive or blocking versions of SURE, one simply needs to change the *do_EDF()* and slack computation routines to support non-preemptive or partially-preemptive EDF scheduling and to correctly compute slack under these circumstances.

Example: Consider the task set $\{T_1, T_2\}$, $T_1 = (2, 1, \{\lambda\})$, $T_2 = (5, 1, \{\lambda\})$ where the deadline is equal to the period and release time is 0. Both tasks require device λ. The hyperperiod is 10. Fig. 2 shows both the EA-EDF schedule and the SURE schedule for the task set where the device λ is idle whenever the CPU is idle (since all tasks use device λ). At $t = 0$, the device λ is *idle*. With EA-EDF the idle times in a hyperperiod are $\{1, 1, 1\}$ time units and the total number of switches is 6. Since the task sets have zero phase, the EA-EDF schedule will be the same in all subsequent hyperperiods. With the SURE schedule, at $t = 0$, the device λ is idled for 1 time unit. At $t = 1$, the highest EDF priority job is executed. Subsequent eligible jobs which require λ are all executed in succession. At $t = 7$, the ready job queue becomes empty and the CPU is idled. The device is changed to the *idle* state. At $t = 8$, $J_{1,5}$ is released. But, since λ is already in the *idle* state, execution of this job is delayed as much as possible. The device remains in the *idle* state till $t = 9$. The remaining jobs all execute in time and complete within their deadline. With slack utilization, the device idle time is 1 time unit in the beginning of the first hyperperiod. Subsequently, longer idle times of 2 time units are obtained. The total number of switches in a hyperperiod is reduced to 3 and the total idle time, of course, remains constant.

4.3. Feasibility

This section presents a necessary and sufficient feasibility condition for the preemptive SURE scheduling algorithm. The condition is the same utilization condition used for preemptive EDF scheduling of synchronous periodic task sets: $U \leq 1$, where U represents the processor utilization of the task set. (The non-preemptive and partially preemptive versions of SURE require different scheduling conditions, which are beyond the scope of this paper.)

The feasibility condition assumes that before a job is executed, it must be ensured that the devices requested by the task are in the *active* state. If the job release time is known in advance, a timer can be used to switch the devices to the *active* state. Alternatively, device switch times can be incorporated within the slack computation to ensure that the devices are in the *active* state before the execution of the job. In this section, we assume either one of these methods are employed to ensure that before a job executes, all devices in the DRS of its task are in the *active* state. Another point to note is that, although it is not explicitly mentioned in the algorithm, if a device is idle for a time less than $2P_{sw}$, SURE will not switch the device to the *idle* state. This is to ensure that the device has sufficient time to transition back from the *idle* state to the *active* state before job execution begins.

Intuitively, the reason the simple utilization test of $U \leq 1$ is a necessary and sufficient condition for the preemptive SURE algorithm is that SURE reduces to EDF scheduling whenever there is no system slack. However, the following lemmas are required before we can actually prove the temporal correctness of the SURE algorithm.

Let the symbol J_i denote a job, for $i = 1$ to N, where N is the total number of jobs released in the hyperperiod.

Lemma 4.1. *If $U \geq 1$, there is no slack, that is, $\forall t \geq 0, \Omega(t) = 0$.*

Proof: Let the slack of a job J_i at t be $\omega(t)$. At $t = 0$, $\omega(0) = D_i - \sum_{D_j \leq D_i} e_j$, where D_i is the absolute deadline of J_i. At $t = 0$,

$$\omega_1(0) = D_1 - \sum_{D_j \leq D_1} e_j$$

$$\omega_2(0) = D_2 - \sum_{D_j \leq D_2} e_j$$

$$\cdots$$

$$\omega_i(0) = D_i - \sum_{D_j \leq D_i} e_j$$

Since the tasks are synchronous and $H = lcm(p_i)$, every task has one job with the deadline as H. Thus, there will be n jobs with deadlines as H. This set of jobs has the slack as,

$$\begin{aligned}
\omega_k(0) &= H - \sum_{D_j \leq H} e_j \\
&= H - \{e_1 + e_2 + \ldots + e_N\} \\
&= H - \{\text{sum of execution times of all jobs in H}\} \\
\text{But, } \{\text{sum of execution times of all jobs in H}\} &= \sum_{i=1}^{n} e_i \cdot \frac{H}{p_i} \\
&= U \cdot H \\
\text{Thus, } \omega_k(0) &= H - U \cdot H \\
&= H \cdot (1 - U) \\
\text{Since, } U \geq 1, \omega_k(0) &\leq 0
\end{aligned}$$

Since slack gets consumed as time progresses, the slack of a job at any time cannot be greater than its slack at time zero. Thus we have,

$$\forall t \geq 0, \omega_k(t) \leq \omega_k(0) \leq 0$$
$$\Rightarrow \omega_k(t) \leq 0$$

At any time $t \geq 0$, from the definition of system slack,

$$\Omega(t) = 0$$

Hence, if $U \geq 1$, there is no slack at any time $t \geq 0$. □

Lemma 4.2. *When there is no slack, SURE reduces to EDF.*

Proof: In the algorithm presented in Section 4.2.2, whenever the system slack is zero, control goes to *do_EDF()* procedure. If the scheduler was invoked because a new job was released, then control goes directly to *do_EDF()*. If the scheduler was invoked because a job finished execution, $computeSlack$ is made *true*. When there is no slack and there are jobs to execute, the control goes to $do\_EDF()$.

In $do\_EDF()$, if the CPU had been idle till now, then a newly released job would immediately execute. If a lower priority job had been executing, it would be preempted by the newly released job. If there are no jobs to execute, then the CPU is idled. Thus, at all times, if there is no slack, the highest EDF priority job is executed each time the scheduler is invoked. In addition, the job is executed immediately since there is no slack at any time to defer the execution of the job. Hence, the algorithm reduces to EDF. $\square$

Lemma 4.3. *In an interval $[t_1, t_2]$ where the CPU is never idled, if for some arbitrary job J_k with absolute deadline at $D_k \geq t_2$ and its release time $r_k \leq t_1$ and $\omega_k(t_2) < 0$, then $\omega_k(t) < 0, \forall t, t_1 \leq t \leq t_2$; the system slack is never positive in the interval $[t_1, t_2]$.*

Proof: We prove this Lemma by contradiction. Suppose that t_i is the last instant before t_2 such that $\Omega(t_i) \geq 0$. Given that slack of job J_k at t_2 is less than zero and from the definition of the slack of a job, we have,

$$\omega_k(t_2) < 0$$

$$\Rightarrow \omega_k(t_2) = D_k - \sum_{D_i \leq D_k} e_i - I(0, t_2) - \sum_{D_i > D_k, r_i < t_2} f_i(t_2) < 0 \tag{2}$$

Since CPU is never idled in $[t_1, t_2]$, $I(0, t_i) = I(0, t_2)$. In addition,

$$\sum_{D_i > D_k, r_i < t_2} f_i(t_2) = \sum_{D_i > D_k, r_i < t_i} f_i(t_2) + \sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2)$$

Substituting in Equation (2),

$$D_k - \sum_{D_i \leq D_k} e_i - I(0, t_i) - \sum_{D_i > D_k, r_i < t_i} f_i(t_2) - \sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2) < 0$$

But, $D_k - \sum_{D_i \leq D_k} e_i - I(0, t_i) - \sum_{D_i > D_k, r_i < t_i} f_i(t_2) = \omega_k(t_i)$

$$\Rightarrow \omega_k(t_i) - \sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2) < 0$$

$$\Rightarrow \omega_k(t_i) < \sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2) \tag{3}$$

The term $\sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2)$ denotes the CPU demand by jobs with lower priority than J_k. The lower priority jobs are given CPU time by the SURE scheduler only if there is system slack. From the assumption that t_i is the last point before t_2 such that $\Omega(t_i) \geq 0$, we have,

$$\begin{aligned} \sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2) &\leq \Omega(t_i) \\ \Rightarrow \omega_k(t_i) < \sum_{D_i > D_k, t_i \leq r_i < t_2} f_i(t_2) &\leq \Omega(t_i) \\ \Rightarrow \omega_k(t_i) &< \Omega(t_i) \end{aligned}$$

By definition of slack, this is possible only if $\Omega(t_i) = 0$ and $\omega_k(t_i) < 0$. Thus, we have proved that $\omega_k(t) < 0$, $\forall t, t_1 \leq t \leq t_2$.

Hence, by definition of system slack,$\Omega(t) = 0$, $\forall t, t_1 \leq t \leq t_2$. This means that the system slack is never positive in the interval $[t_1, t_2]$. □

Lemma 4.4. *Consider an interval $[t_0, t_2]$ where $\omega_k(t_2) < 0$ and t_0 is the last instant before t_2 at which the CPU has no jobs to execute. Suppose t_1 is the last CPU idle instant before t_2, $t_0 \leq t_1 < t_2$, and $t_0 \leq r_k \leq t_1$, where r_k is the release time of some job J_k with absolute deadline at $D_k \geq t_2$ then $t_0 = t_1$.*

Proof: In the interval $[t_1, t_2]$, the CPU is never idled since t_1 is the last instant before t_2 at which the CPU is idle. It is also given that $r_k \leq t_1$ and $\omega_k(t_2) < 0$. Thus, from Lemma 4.3, $\omega_k(t) < 0$, $\forall t, t_1 \leq t \leq t_2$. And the system slack is never positive in the interval $[t_1, t_2]$. Thus at t_1, slack of job J_k is less than zero,

$$\begin{aligned} \omega_k(t_1) &< 0 \\ \Rightarrow \omega_k(t_1) = D_k - \sum_{D_i \leq D_k} e_i - I(0, t_1) - \sum_{D_i > D_k, r_i < t_1} f_i(t_1) &< 0 \end{aligned} \tag{4}$$

$$\sum_{D_i > D_k, r_i < t_1} f_i(t_1) = \sum_{D_i > D_k, r_i < t_1} f_i(t_0) + \sum_{D_i > D_k, r_i < t_1} f_i(t_0, t_1)$$

The term $\sum_{D_i > D_k, r_i < t_1} f_i(t_0)$ refers to the amount of time jobs with deadline greater than D_k, have executed till t_0. This implies that these jobs have to be released before t_0. Hence, we can change the notation to $\sum_{D_i > D_k, r_i < t_0} f_i(t_0)$. Also, since t_0 is the last instant before t_2 at which the CPU has no jobs to execute, $\sum_{D_i > D_k, r_i < t_1} f_i(t_0, t_1)$ can be made equal to $\sum_{D_i > D_k, t_0 \leq r_i < t_1} f_i(t_0, t_1)$. This term $\sum_{D_i > D_k, t_0 \leq r_i < t_1} f_i(t_0, t_1)$ refers to the CPU time allotted to jobs with priority lower than J_k but released in $[t_0, t_1]$. Thus,

$$\sum_{D_i > D_k, r_i < t_1} f_i(t_1) = \sum_{D_i > D_k, r_i < t_0} f_i(t_0) + \sum_{D_i > D_k, t_0 \leq r_i < t_1} f_i(t_0, t_1)$$

Since the CPU is idle till t_1, these jobs will not be allotted any CPU time i.e, $\sum_{D_i > D_k, t_0 \leq r_i < t_1} f_i(t_0, t_1) = 0$.

$$\sum_{D_i > D_k, r_i < t_1} f_i(t_1) = \sum_{D_i > D_k, r_i < t_0} f_i(t_0)$$

Substituting this in Equation (4)

$$D_k - \sum_{D_i \leq D_k} e_i - I(0, t_1) - \sum_{D_i > D_k, r_i < t_0} f_i(t_0) < 0$$

Also, $I(0, t_1) = I(0, t_0) + I(t_0, t_1)$. Substituting this and rearranging the terms,

$$\Big\{ D_k - \sum_{D_i \leq D_k} e_i - I(0, t_0) - \sum_{D_i > D_k, r_i < t_0} f_i(t_0) \Big\} - I(t_0, t_1) < 0 \tag{5}$$

But, $D_k - \sum_{D_i \leq D_k} e_i - I(0, t_0) - \sum_{D_i > D_k, r_i < t_0} f_i(t_0) = \omega_k(t_0)$

Substituting in Equation (5),

$$\omega_k(t_0) - I(t_0, t_1) < 0$$

$$\Rightarrow \omega_k(t_0) < I(t_0, t_1)$$

But, the CPU is idled from t_0 to t_1 by the SURE scheduler only if there is system slack at t_0.

$$I(t_0, t_1) \leq \Omega(t_0)$$

$$\Rightarrow \omega_k(t_0) < \Omega(t_0)$$

But, if the system slack is positive at t_0, $\omega_k(t_0) \geq \Omega(t_0)$, a contradiction. Thus, it must be that $\Omega(t_0) = 0$ and

$\omega_k(t_0) < 0$. Thus, there is no positive slack in $[t_0, t_1]$. This also implies that CPU will not be idled from t_0 to t_1 and t_1 is the last instant at which the CPU is idle before t_2, $t_0 = t_1$. Thus, it follows that slack is never positive in $[t_0, t_2]$. $\square$

Thus, as long as system slack is computed correctly, processing can be delayed until there is no more slack in the schedule. At that point, the task set is scheduled using EDF. The challenge in proving that SURE generates a correct schedule arises when tasks are executed "out of order" with respect to EDF when there is non-zero system slack.

Theorem 4.5. *A set of synchronous periodic tasks $T = \{T_1, T_2, T_3, ...T_n\}$, with deadlines equal to their periods, can be feasibly scheduled on a single processor with preemptive SURE if and only if* $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$.

Proof: For the proof of necessity of the Theorem, suppose that $U > 1$. From Lemma 4.1, we know that if $U \geq 1$, there is no slack. From Lemma 4.2, we know that if there is no slack at any time, then SURE reduces to EDF. Hence, we can conclude that since $U > 1$, SURE reduces to EDF. We know that if $U > 1$, EDF will fail to find a schedule. Since SURE reduces to EDF, SURE will also fail when $U > 1$. Thus, necessity is proved.

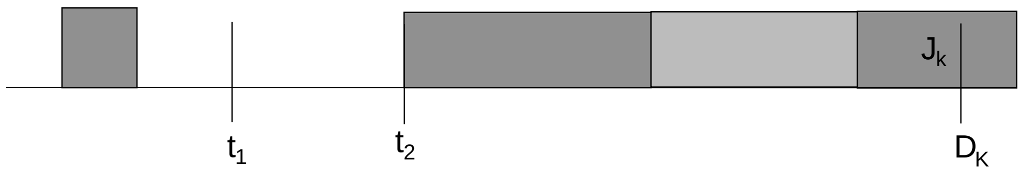


Figure 3. Instance of the SURE schedule when J_k misses a deadline at time D_k.

For sufficiency, assume that $U \leq 1$, but tasks cannot be feasibly scheduled. In Fig. 3, let J_k be the first job to miss its deadline at D_k and t_0 be the last instant before D_k at which the CPU has no jobs to execute. t_0 can be traced back to 0 if there are no idle instants thereafter. Let t_1 be the last instant before D_k at which the CPU is idle. Since SURE is a non-work-conserving algorithm, it does not guarantee that a job will be scheduled immediately if the CPU is free. Hence, the last idle instant of the CPU maybe equal to or after the last instant at which the CPU has no jobs to execute. Thus, $t_0 \leq t_1$.

Let r_k be the absolute release time of J_k. Since t_0 is the last instance before D_k at which the CPU has no jobs to execute, it follows that

$$t_0 \leq r_k < D_k$$

Thus, with reference to the release time of J_k, we can define two cases.

a) $t_0 \leq r_k \leq t_1 < D_k$

b) $t_0 \leq t_1 < r_k < D_k$

Case (a): $t_0 \leq r_k \leq t_1 < D_k$

In the interval (t_1, D_k), the CPU is never idled since t_1 is the last instant before D_k at which the CPU is idle. It is also given that $r_k \leq t_1$. Since J_k misses its deadline at D_k, $\omega_k(D_k) < 0$. Thus, from Lemma 4.4, $\omega_k(t) < 0$, $\forall t, t_0 \leq t \leq D_k$. This also means that the system slack is never positive in the interval $[t_0, D_k]$ and that CPU is never idled in $[t_0, D_k]$.

$$\omega_k(D_k) < 0$$

$$\Rightarrow \omega_k(D_k) = D_k - \sum_{D_i \leq D_k} e_i - I(0, D_k) - \sum_{D_i > D_k, r_i < D_k} f_i(D_k) < 0 \tag{6}$$

Since CPU is never idled in $[t_0, D_k]$, $I(0, t_0) = I(0, D_k)$. In addition,

$$\sum_{D_i \leq D_k} e_i = \sum_{D_i \leq t_0} e_i + \sum_{t_0 < D_i \leq D_k} e_i$$

Substituting in Equation (6),

$$D_k - \sum_{D_i \leq t_0} e_i - \sum_{t_0 < D_i \leq D_k} e_i - I(0, t_0) - \sum_{D_i > D_k, r_i < D_k} f_i(D_k) < 0 \tag{7}$$

But, $$\sum_{D_i > D_k, r_i < D_k} f_i(D_k) = \sum_{D_i > D_k, r_i < D_k} f_i(t_0) + \sum_{D_i > D_k, r_i < D_k} f_i(t_0, D_k)$$

To have the CPU allocated to them, these jobs have to be released before t_0,

$$\sum_{D_i > D_k, r_i < D_k} f_i(t_0) = \sum_{D_i > D_k, r_i < t_0} f_i(t_0)$$

$$\Rightarrow \sum_{D_i > D_k, r_i < D_k} f_i(D_k) = \sum_{D_i > D_k, r_i < t_0} f_i(t_0) + \sum_{D_i > D_k, r_i < D_k} f_i(t_0, D_k)$$

The term $\sum\limits_{D_i>D_k, r_i<D_k} f_i(t_0, D_k)$ is the CPU time allocated in (t_0, D_k) to ready jobs with lower priority than J_k. Since there is no system slack in $[t_0, D_k]$, these jobs will not be allocated any CPU time. Thus, $\sum\limits_{D_i>D_k, r_i<D_k} f_i(t_0, D_k) = 0$. Substituting in Equation (7),

$$D_k - \sum_{D_i \le t_0} e_i - \sum_{t_0<D_i\le D_k} e_i - I(0,t_0) - \sum_{D_i>D_k, r_i<t_0} f_i(t_0) < 0$$

Adding and subtracting t_0 and rearranging the terms,

$$(D_k - t_0) + \Big\{ t_0 - \sum_{D_i \le t_0} e_i - I(0,t_0) - \sum_{D_i>D_k, r_i<t_0} f_i(t_0) \Big\} - \sum_{t_0<D_i\le D_k} e_i < 0 \qquad (8)$$

But, $t_0 - \sum\limits_{D_i \le t_0} e_i - I(0,t_0) - \sum\limits_{D_i>D_k, r_i<t_0} f_i(t_0) \qquad = \omega_j(t_0)$

where, $\omega_j(t_0)$ is the slack of the job with absolute deadline at t_0, the instant before the CPU's job queue becomes non-empty. Since the tasks are synchronous with deadlines equal to periods, if t_0 marks the instant at which some job is released, it must also mark the absolute deadline of some job, J_j. Since J_k was the first job to miss its deadline, $\omega_j(t_0) = 0$. The slack of this job J_j is positive when it finishes executing and when the CPU becomes idle before t_0. This slack then decreases and at t_0, the slack of the job is zero. So, (8) becomes,

$$(D_k - t_0) - \sum_{t_0<D_i\le D_k} e_i < 0 \qquad (9)$$

But,
$$\begin{aligned}\sum_{t_0<D_i\le D_k} e_i &= \sum_{t_0<D_i\le D_k, r_i<D_k} e_i \\ &= \sum_{t_0<D_i\le D_k, r_i<t_0} e_i + \sum_{t_0<D_i\le D_k, t_0\le r_i<D_k} e_i\end{aligned}$$

Substituting in Equation (9),

$$(D_k - t_0) - \sum_{t_0<D_i\le D_k, r_i<t_0} e_i - \sum_{t_0<D_i\le D_k, t_0\le r_i<D_k} e_i < 0 \qquad (10)$$

The term $\sum\limits_{t_0 < D_i \leq D_k, r_i < t_0} e_i$ refers to the CPU demand of the jobs released before t_0 and with deadlines in $(t_0, D_k]$. Since t_0 is the last instant at which the CPU has no jobs to execute, there can be no jobs released prior to t_0 with deadlines in $(t_0, D_k]$. Thus, $\sum\limits_{t_0 < D_i \leq D_k, r_i < t_0} e_i = 0$. So Equation (10) becomes,

$$(D_k - t_0) - \sum_{t_0 < D_i \leq D_k, t_0 \leq r_i < D_k} e_i < 0$$

The term $\sum\limits_{t_0 < D_i \leq D_k, t_0 \leq r_i < D_k} e_i$ is the CPU demand of the jobs with deadlines in $(t_0, D_k]$ and released in $[t_0, D_k)$. Thus,

$$\begin{aligned}
(D_k - t_0) &< \sum_{j=1}^{n} \left\lfloor \frac{D_k - t_0}{p_j} \right\rfloor \cdot e_j \\
&\leq \sum_{j=1}^{n} \left(\frac{D_k - t_0}{p_j} \right) \cdot e_j \\
&= (D_k - t_0) \sum_{j=1}^{n} \frac{e_j}{p_j} \\
(D_k - t_0) &< (D_k - t_0) \cdot U \\
1 &< U, \text{ a contradiction}
\end{aligned}$$

This implies that in this case, if $U \leq 1$, SURE will find a valid schedule.

Case (b): $t_0 \leq t_1 < r_k < D_k$

In the interval $[r_k, D_k]$ the CPU is never idled since t_1 is the last instant before D_k at which the CPU is idle. Since J_k misses its deadline at D_k, $\omega_k(D_k) < 0$. Thus, from Lemma 4.3, $\omega_k(t) < 0$, $\forall t, r_k \leq t \leq D_k$. This also means that the system slack is never positive in the interval $[r_k, D_k]$.

Suppose that t_3 is the last instant before D_k such that $\omega_k(t_3) \geq 0$. We have intentionally avoided using the notation t_2 to avoid confusion from the previous definition in Lemma 4.3, and Lemma 4.4. Let t_4 be the first instant after t_3 where slack of job J_k is negative i.e, $\omega_k(t_4) < 0$ where $t_4 \leq r_k < D_k$. This also means that, $\forall t$, $t_4 \leq t \leq D_k$, $\omega_k(t) < 0$. Figure 4 illustrates this case. Note that as assumed here and by the earlier definition of the slack of a job, a job can have slack less than zero before it is released. Now, at t_4, the slack of job J_k is less

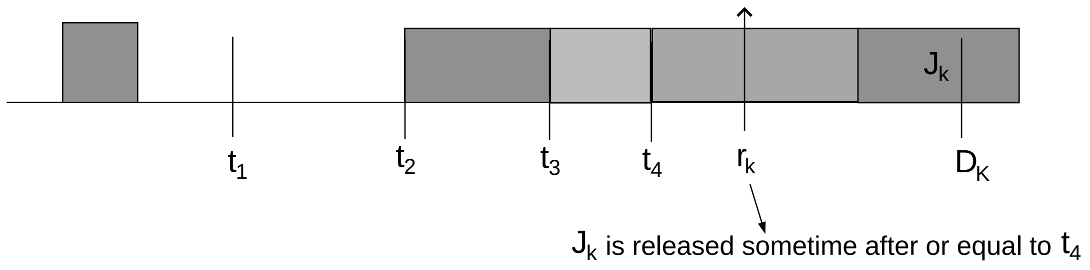


Figure 4. Instance of the SURE schedule when J_k misses a deadline at time D_k. t_3 is the last instant before D_k such that $\omega_k(t_3) \geq 0$ and t_4 be the first instant after t_3 where slack of job J_k is negative.

than zero.

$$\omega_k(t_4) < 0$$

$$\Rightarrow \omega_k(t_4) = D_k - \sum_{D_i \leq D_k} e_i - I(0,t_4) - \sum_{D_i > D_k, r_i < t_4} f_i(t_4) < 0$$

Since CPU is never idled in $[t_1, D_k]$, $I(0,t_4) = I(0,t_3)$. In addition,

$$\begin{aligned} \sum_{D_i > D_k, r_i < t_4} f_i(t_4) &= \sum_{D_i > D_k, r_i < t_4} f_i(t_3) + \sum_{D_i > D_k, r_i < t_4} f_i(t_3,t_4) \\ &= \sum_{D_i > D_k, r_i < t_3} f_i(t_3) + \sum_{D_i > D_k, r_i < t_4} f_i(t_3,t_4) \end{aligned}$$

Substituting in Equation (11),

$$\Rightarrow D_k - \sum_{D_i \leq D_k} e_i - I(0,t_3) - \sum_{D_i > D_k, r_i < t_3} f_i(t_3) - \sum_{D_i > D_k, r_i < t_4} f_i(t_3,t_4) < 0$$

$$\text{But, } D_k - \sum_{D_i \leq D_k} e_i - I(0,t_3) - \sum_{D_i > D_k, r_i < t_3} f_i(t_3) = \omega_k(t_3)$$

$$\Rightarrow \omega_k(t_3) - \sum_{D_i > D_k, r_i < t_4} f_i(t_3,t_4) < 0 \tag{11}$$

$$\omega_k(t_3) < \sum_{D_i > D_k, r_i < t_4} f_i(t_3,t_4) \tag{12}$$

Thus, a portion of the CPU time from from (t_3, t_4) is allocated to lower priority jobs with deadlines greater than D_k. Thus slack of J_k is consumed by lower priority jobs and this can happen because of two reasons:

(i) The lower priority jobs are scheduled by SURE.

(ii) The lower priority jobs are scheduled because no higher priority jobs are released yet.

We will consider both cases.

Case (i): Slack of J_k is consumed by lower priority jobs because of SURE scheduler.

If slack is consumed because of the SURE scheduler, we can be assured that the slack consumed is always less than or equal to the system slack at that time. Hence,

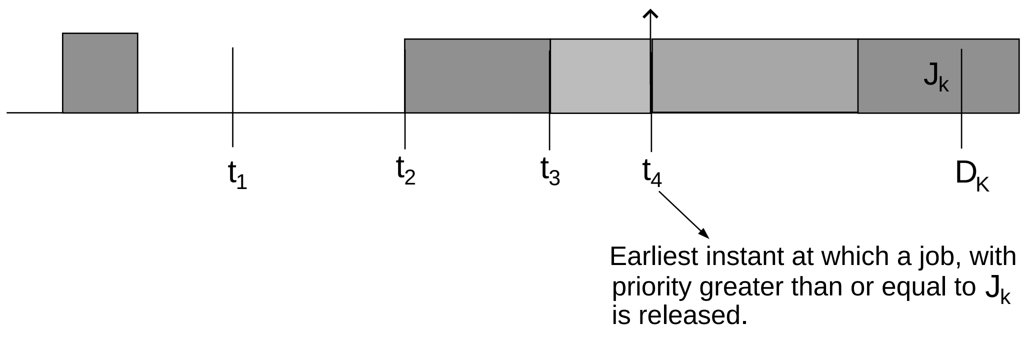
$$0 \leq \omega_k(t_3) < \sum_{D_i > D_k, r_i < t_4} f_i(t_3, t_4) \leq \Omega(t_3)$$

$$\Rightarrow 0 \leq \omega_k(t_3) < \Omega(t_3)$$

By definition of slack, this is possible only if $\Omega(t_3) = 0$ and $\omega_k(t_3) < 0$. So, t_3 cannot be the last instant before D_k where slack of job J_k is non-negative and this case reduces to Case (a).

Case (ii): Slack of J_k is consumed by lower priority jobs because higher priority jobs are not yet released.

Slack of a job can be consumed by execution of lower priority jobs or by CPU idling. Since CPU is not idled in $[t_1, D_k]$, it must be that lower priority jobs consumed the slack. If slack of J_k is not allocated by the SURE scheduler it must be that slack of J_k is consumed by lower priority jobs only because higher priority jobs are not released before t_4. Thus, t_4 marks the instant when J_k or a higher priority job i.e., a job with deadline lesser than D_k is released. This is in fact similar to the proof of EDF when there are some jobs, which start their current period earlier than the release time of the job which misses its deadline. Figure 5 illustrates this case.



J_k
t_1
t_2
t_3
t_4
D_K
Earliest instant at which a job, with priority greater than or equal to J_k is released.

Figure 5. Instance of the SURE schedule when J_k misses a deadline at time D_k and t_4 marks the instant when J_k or a higher priority job is released.

We know from Lemma 4.3, $\omega_k(t) < 0$, $\forall t, r_k \leq t \leq D_k$. Now, at D_k, slack of the job J_k is less than zero.

$$\begin{aligned} \omega_k(D_k) &< 0 \\ \Rightarrow \omega_k(D_k) = D_k - \sum_{D_i \leq D_k} e_i - I(0, D_k) - \sum_{D_i > D_k, r_i < D_k} f_i(D_k) &< 0 \end{aligned} \tag{13}$$

Since the CPU is never idled in $[t_1, D_k]$, $I(0, D_k) = I(0, t_4)$. In addition,

$$\begin{aligned} \sum_{D_i > D_k, r_i < D_k} f_i(D_k) &= \sum_{D_i > D_k, r_i < D_k} f_i(t_4) + \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) \\ &= \sum_{D_i > D_k, r_i < t_4} f_i(t_4) + \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) \\ &= \sum_{D_i > t_4, r_i < t_4} f_i(t_4) - \sum_{t_4 \leq D_i \leq D_k, r_i < t_4} f_i(t_4) + \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) \end{aligned}$$

The term $\sum_{t_4 \leq D_i \leq D_k, r_i < t_4} f_i(t_4)$ refers to the CPU time allocated till t_4 to jobs with deadlines in $[t_4, D_k]$. But, we know that all jobs with priority equal to and higher than J_k are not released before t_4. Thus, $\sum_{t_4 \leq D_i \leq D_k, r_i < t_4} f_i(t_4) = 0$. Substituting in Equation (13),

$$D_k - \sum_{D_i \leq D_k} e_i - I(0, t_4) - \sum_{D_i > t_4, r_i < t_4} f_i(t_4) - \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) < 0$$

Adding and subtracting t_4,

$$(D_k - t_4) + \left\{ t_4 - \sum_{D_i \leq D_k} e_i - I(0, t_4) - \sum_{D_i > t_4, r_i < t_4} f_i(t_4) \right\} - \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) < 0 \tag{14}$$

$$\begin{aligned} \text{But, } \sum_{D_i \leq D_k} e_i &= \sum_{D_i \leq D_k, r_i < D_k} e_i \\ &= \sum_{D_i \leq t_4, r_i < D_k} e_i + \sum_{t_4 < D_i \leq D_k, r_i < D_k} e_i \\ &= \sum_{D_i \leq t_4, r_i < t_4} e_i + \sum_{t_4 < D_i \leq D_k, r_i < D_k} e_i \end{aligned}$$

Substituting in Equation (14) and rearranging the terms,

$$(D_k - t_4) - \sum_{t_4 < D_i \leq D_k, r_i < D_k} e_i + \left\{ t_4 - \sum_{D_i \leq t_4, r_i < t_4} e_i - I(0, t_4) - \sum_{D_i > t_4, r_i < t_4} f_i(t_4) \right\} - \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) < 0 \tag{15}$$

$$\text{But, } t_4 - \sum_{D_i < t_4, r_i < t_4} e_i - I(0, t_4) - \sum_{D_i > t_4, r_i < t_4} f_i(t_4) = \omega_j(t_4)$$

where, $\omega_j(t_4)$ is the slack of the job, J_j, released by some task that releases another job at t_4. Since the tasks are synchronous and deadlines are equal to periods, t_4 marks the absolute deadline of the job J_j. As J_k was the first job to miss its deadline, $\omega_j(t_4) \geq 0$. Equation (15) becomes,

$$(D_k - t_4) - \sum_{t_4 < D_i \leq D_k, r_i < D_k} e_i - \sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) < 0 \tag{16}$$

The term $\sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k)$ refers to the CPU time allocated in (t_4, D_k) to jobs with lower priority than J_k. Since there is no positive slack in $[t_4, D_k]$, the CPU will never be allocated to lower priority jobs. Hence, $\sum_{D_i > D_k, r_i < D_k} f_i(t_4, D_k) = 0$. So, Equation (16) reduces to,

$$(D_k - t_4) - \sum_{t_4 < D_i \leq D_k, r_i < D_k} e_i < 0$$

$$(D_k - t_4) - \sum_{t_4 < D_i \leq D_k, r_4 \leq r_i < D_k} e_i - \sum_{t_4 < D_i \leq D_k, r_i < r_4} e_i < 0$$

The term $\sum_{t_4 < D_i \leq D_k, r_i < r_4} e_i$ refers to the CPU demand of the jobs with deadlines in $(t_4, D_k]$ but released before t_4. We know that there are no such jobs released before t_4. So, this CPU demand is zero. Hence,

$$(D_k - t_4) - \sum_{t_4 < D_i \leq D_k, r_4 \leq r_i < D_k} e_i < 0$$

$$(D_k - t_4) < \sum_{t_4 < D_i \leq D_k, r_4 \leq r_i < D_k} e_i$$

The term $\sum_{t_4 < D_i \leq D_k, r_4 \leq r_i < D_k} e_i$ signifies the CPU demand of jobs with deadlines in $(t_4, D_k]$ and released in

$[t_4, D_k)$.

$$\begin{aligned}
(D_k - t_4) &< \sum_{j=1}^{n} \left\lfloor \frac{D_k - t_4}{p_j} \right\rfloor \cdot e_j \\
&\le \sum_{j=1}^{n} \left(\frac{D_k - t_4}{p_j} \right) \cdot e_j \\
&= (D_k - t_4) \sum_{j=1}^{n} \frac{e_j}{p_j} \\
(D_k - t_4) &< (D_k - t_4) \cdot U \\
1 &< U \text{ , a contradiction}
\end{aligned}$$

This implies that in this case, if $U \le 1$, a feasible schedule will be produced. Hence, sufficiency is proved. □

5. Evaluation

In this section, we present evaluation results for the EA-EDF, EEA-EDF and SURE algorithms. Section 5.1 describes the the evaluation methodology used in this study. Section 5.2 reports on the performance of the algorithms when the CPU is only shared device. Section 5.3 describes the evaluation of the algorithms on a system with multiple shared devices, and Section 5.4 compares SURE with the minimum energy schedule generated using a brute-force algorithm that explores the entire state space to generate a feasible schedule with minimal energy consumption.

5.1. Methodology

We evaluated the EA-EDF, EEA-EDF and SURE algorithms using a simulator written in the C programming language. This approach is consistent with the evaluation methodologies of [5] for leakage power and [14, 15, 16] for I/O devices, which were all offline scheduling algorithms. Moreover, since the main concern in the evaluation of an energy-saving algorithm is the amount of energy it saves, we evaluated EA-EDF, EEA-EDF and SURE as offline algorithms. An online implementation of EA-EDF, EEA-EDF and SURE will generate the same amount of energy savings as the offline simulation as long as the tasks execute with there worst case execution time. In the case of the SURE algorithm, the primary difference in the online version is the additional overhead incurred in the adjustments to the pre-computed slack table due to inserted idle time, which has cost $O(n)$. For the purposes of this study, however, this overhead was not measured. Instead each of the algorithms is evaluated as an offline algorithm.

The power requirements and state switching times for devices were obtained from data sheets provided by the manufacturer when available, or measured experimentally when the data was not provided by the manufacturer. For example, the Rabbit 3000 was chosen as the CPU device because it is used in many embedded systems, including several of our own research projects. Rabbit Semiconductor provides a data sheet on the processor that provides average power requirements for the processor in its various power states. However, the vendor does not provide the average time taken to change power states. Thus, a series of experiments were conducted to measure the average power state switch times as well as the average energy consumed at each power state. The measured amount of energy consumed was very close to the amount computed using the data sheet, given the measured power state switch times. Thus, the power requirements provided in the data sheet for the device were used for the simulation experiments. The average power state switch time used in the simulations, however, is based on the average measured times since the vendor does not provide this information.

The *normalized energy savings* is used to evaluate the energy savings of the three algorithms. The normalized energy savings is the amount of energy saved under a DPM algorithm relative to the case when no DPM technique is used. It is computed using Equation (17).

$$\textit{Normalized Energy Savings} = \frac{\textit{Energy with No DPM} - \textit{Energy with Energy Saving Algorithm}}{\textit{Energy with No DPM}} \tag{17}$$

In all of the experiments, 500 task sets were generated randomly with random utilization (where $U \leq 1$). The number of tasks in any task set was also a random number between 1 and 20.

To ensure that the a device changes its state before the start of the execution of the job utilizing the device, the absolute time at which the device should be switched to the *active* state is recorded from the schedule. During execution, an interrupt is issued at this time to switch the device state to the *active* state. Similarly, when a job finishes execution and the algorithm detects that a device should be switched to the *idle* state, the power state switch is carried out. When no DPM technique is used, all devices specified in the DRS of the tasks will remain in the *active* state over the entire hyperperiod. Note that only the first hyperperiod must be simulated since a synchronous periodic task set is assumed.

Device	P_{active} (mW)	P_{idle} (mW)	P_{sw} (mW)	t_{sw} (ms)
Rabbit 3000	198	.3729	30.3	12.1

Table 1. CPU Specifications.

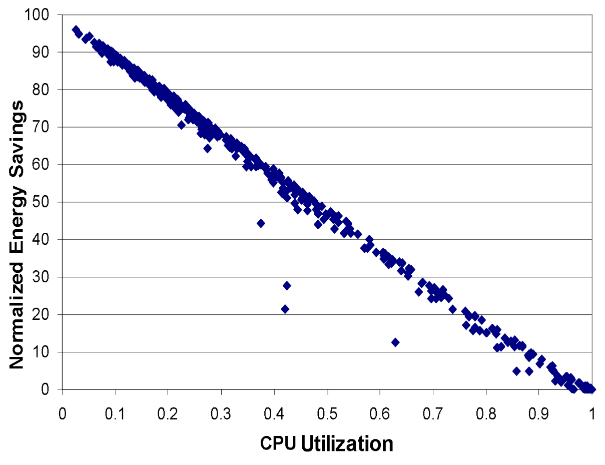
5.2. CPU as the Only Shared Device

In this experiment we examined the performance of EA-EDF, EEA-EDF and SURE as if the CPU was the only shared device. The Rabbit 3000 processor was selected for evaluation purposes. This processor is an 8-bit 29.4MHz embedded processor running the MicroC/OS-II real-time operating system. The power specifications for the Rabbit 3000 are shown in Table 1, where P_{active} and P_{idle} were obtained from the product data sheets provided by the manufacturer while t_{switch} and P_{switch} were measured experimentally. The Rabbit 3000 processor has a main oscillator that runs the processor in active mode and a low power 23kHz oscillator that runs the processor in idle mode. The switching time between the main oscillator and the 32kHz oscillator is 0.9 ms while the switching time from the 32kHz to the main oscillator is 23.3 ms. The average of these two values was used for t_{switch} since the energy conservation model presented in Section 3 assumes that the switching overhead is symmetric for a device. (Note that most vendors report only a single switching time, if they report any at all.)

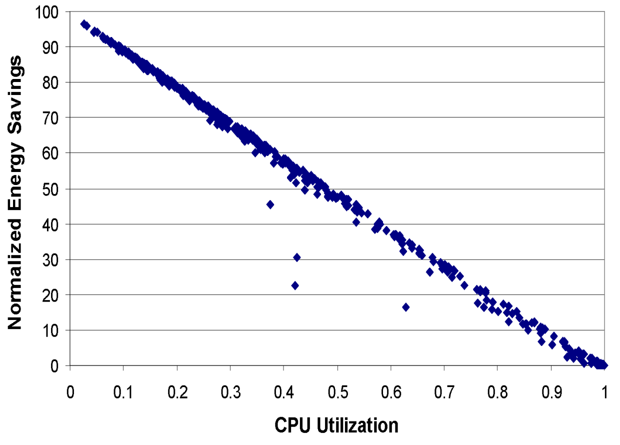
In this experiment, the EA-EDF and EEA-EDF algorithms perform exactly the same because the CPU is the only device used, and both algorithms switch it to the idle state whenever there is no job to execute. Figure 6(a) shows the normalized energy savings for these two algorithms. Each of the 500 randomly generated task sets results in a different energy savings. The general trend follows the expectation that the normalized energy savings should decrease linearly with increases in CPU utilization.

Figure 6(b) plots the normalized energy savings of the SURE algorithm against CPU utilization. The SURE algorithm performs only slightly better than the EA-EDF and EEA-EDF algorithms in total energy savings. There is not a significant performance improvement because the switching energy consumed by the Rabbit 3000 processor is relatively low, and all three algorithms place the processor in the low power state for the same amount of time—sans switching time.

The obvious conclusion to draw from this experiment is that the EA-EDF algorithm should be selected when the CPU is the only shared device and the cost of switching power states is low; it has the lowest overhead and results in nearly the same energy savings as the other two algorithms.



(a) Normalized energy savings with either EA-EDF or EEA-EDF.



(b) Normalized energy savings with SURE.

Figure 6. Normalized energy savings with only the CPU as a device.

Device	P_{active} (W)	P_{idle} (W)	P_{sw} (W)	t_{sw} (ms)
SST Flash SST39LF020 [13]	0.125	0.001	0.05	1
SimpleTech Flash Card [12]	0.225	0.02	0.1	2
TI DSP TMS320C6411 (Digital Signal Processor)[17]	0.63	0.2	0.4	500

Table 2. Device Specifications.

5.3. Multiple Devices

In this experiment we used the same randomly generated task sets created for the prior experiment (i.e., when the CPU was the only shared device). However, we changed the DRS of the tasks so that each task uses the CPU and a random number of devices from the set of devices shown in Table 2. The parameters for these devices were obtained from the product data sheets provided by the manufacturer.

In this case we plotted the normalized energy savings against the *total device utilization* (TDU), which is the sum of the *device utilization factors* for all devices specified in the DRS of the tasks. The TDU can be more than 1 since, if a task uses several devices, the device utilization factor of each requested device increases by an amount equal to the CPU utilization of that task. Figure 7 shows the a plot of the normalized energy savings against the TDU for EEA-EDF and SURE. In this case, the energy savings of the EEA-EDF algorithm is always at least as great as the savings for the EA-EDF. Thus, to simply the presentation, only results for the EEA-EDF and SURE algorithms are plotted together.

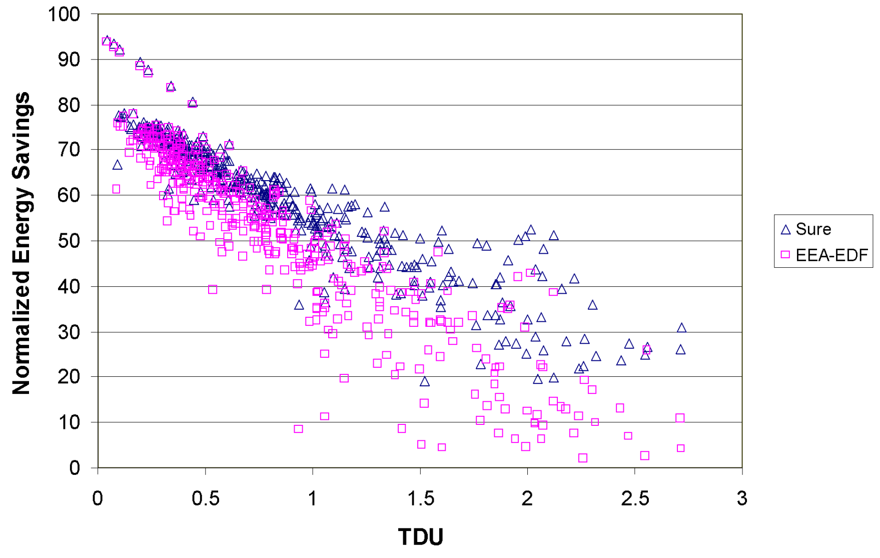


Figure 7. Normalized energy savings with EEA-EDF and SURE.

On average, SURE saves more energy than EEA-EDF (or the EA-EDF). In most cases, as the TDU increases, the normalized energy savings decreases. The rationale for this is that as devices are utilized more, the amount of time they can be kept in *idle* mode decreases. There are a few instances in which the SURE algorithm actually results in more energy being consumed than the EEA-EDF algorithm. This is because SURE tries to minimize the number of switches at a given time, but it does not ensure that the total number of switches is minimized. Thus, a locally optimal choice may result in a globally suboptimal result, which is not uncommon in heuristic algorithms.

Another metric used to compare the performance of SURE with EEA-EDF is the *percentage of switch reductions* computed using Equation (18).

$$\text{Percentage of Switch Reductions} = \frac{\text{Num of Device Switches with EEA-EDF} - \text{Num of Device Switches with SURE}}{\text{Num of Device Switches with EEA-EDF}} \tag{18}$$

Figure 8 shows the percentage of total device switch reductions performed by SURE as compared to EEA-EDF. A general trend is that, as TDU increases, the percentage of switch reductions increases. This is expected since, as device utilization increases, there is a higher probability of SURE finding a ready job requiring the devices which are already in the *active* state. Hence, the SURE algorithm can more effectively rearrange the jobs to reduce the number of switches. However as in the earlier cases, the actual value of the percentage of switch reductions is dependent on parameters of the task set.

Perhaps the most remarkable result from this experiment is not that SURE saves more energy, on average, than

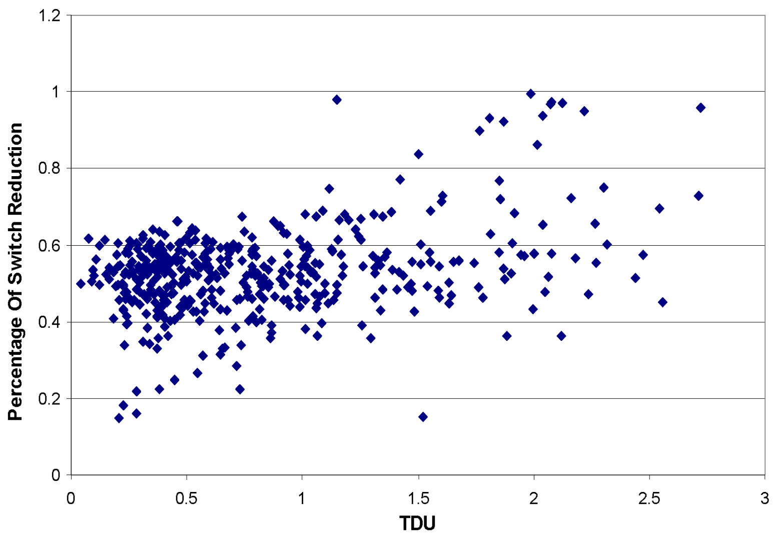


Figure 8. Percentage of switch reductions performed by SURE as compared to EEA-EDF.

either EA-EDF or EEA-EDF; it is that the savings is not larger. From Figure 8, it is clear that SURE has the potential to save much more energy than EA-EDF or EEA-EDF. However, the devices used in this experiment all have relatively low switching costs, which reduces the benefit of lowering the number of switches. Devices with much higher switching costs, such as disks, will result in much greater energy savings for SURE compared with EA-EDF or EEA-EDF.

5.4. Comparison with Minimum Energy Schedule (MES)

To compare the SURE algorithm with the *minimum energy schedule* (MES) for a task set, we implemented the brute force method of constructing the entire tree of preemptive schedules for a given task set and selecting the schedule with the minimum energy. We implemented this by means of a Depth-First-Search method by examining the tree at each time-tick, which translates into a tree of H levels. Since preemptive schedules are considered, at every level, we also consider CPU idling for one time-tick as a job with infinite deadline and execution time. If a job misses a deadline, then no further jobs at that level of the tree are examined. If a job has not yet been released, we move on to the next job in the same level without further branching. This simply means that we look at scheduling only ready jobs at any time. The energy is computed at every level. When all the schedules have been exhausted, we retrieve the minimum energy value for the given task set. The complexity of this algorithm is $O((n+1)^{H-1})$, where n is the number of tasks in the task set and H is the length of the hyperperiod.

H	Execution Time		Energy Consumed (W)	
	MES	SURE	MES	SURE
≤ 10	$< 1s$	$< 1s$	74	78
≤ 20	$> 30min$	$< 1s$	121	123
≤ 30	$> 30min$	$< 1s$	116	120
≤ 40	$> 30min$	$< 1s$	176	180
≤ 50	$> 30min$	$< 1s$	138	138
≤ 60	$> 1day$	$< 1s$	164	164

Table 3. Comparison with the Minimum Energy Scheduler.

We ran experiments for H ranging from 10 to 60 time units with varying task temporal parameters and DRS. Table 3 shows the worst case values for the time taken to execute the Minimum Energy algorithm and the corresponding value of the minimum energy for the task set as compared to the SURE algorithm. We stopped at $H > 60$, because for some task sets it was taking several days to complete. In fact, for $H = 32$, $n = 4$, $N = 6$, the Minimum Energy algorithm executed for more than 5 days on a Sun workstation.

It can be observed from Table 3, that as we approach higher orders of the hyperperiod, the energy savings with the SURE schedule is more than 90% of the optimal solution. More importantly, the time taken to compute the SURE schedule is many orders of magnitude less than the time to compute the MES schedule.

6. Conclusion

Three real-time scheduling algorithms were presented for conserving energy in processors, I/O devices, or device subsystems. For preemptive versions of all three algorithms, $U \leq 1$ is a necessary and sufficient schedulability condition. The EA-EDF and EEA-EDF algorithms, though relatively simple extensions to EDF scheduling, provide remarkable power savings when the cost of switching power states is low. As the cost of switching power states increases, so does the energy savings produced by the SURE algorithm—especially with respect to EA-EDF and EEA-EDF. Ultimately, the choice of which energy saving algorithm to choose, if any, depends on the temporal parameters of the task set and devices utilized.

In Section 5, we illustrated that the SURE algorithm does not result in the minimum energy schedule when multiple devices are shared; none of the three algorithms are optimal in this case. The problem of finding a feasible schedule which consumes minimum I/O device energy is NP-hard. Hence, our focus was not to find the optimal solution but to create algorithms that reduce the energy consumption of multiple shared devices and that can be executed online to adapt to the work load. The preemptive SURE algorithm provides the foundation for a family of general, non-work-conserving, online energy saving algorithms that can be applied to systems with hard temporal constraints.

References

[1] Advanced configuration & power interface specification. Advanced Configuration & Power Interface, August 2003. http://www.acpi.info/DOWNLOADS/ACPIspec-2-0c.pdf.

[2] S. Borkar. Low power design challenges for the decade. In *Proc. Asia South Pacific Design Automation Conference*, pages 293–296, 2001.

[3] L. R. Carley, G. R. Ganger, D. F. Guillou, and D. Nagle. System design considerations for mems-actuated magnetic-probe-based mass storage. *IEEE Transactions on Magnetics*, 37(2 Part 1):657–662, March 2001.

[4] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness if not sloth. In *Proceedings of the Winter USENIX Conference*, 1996.

[5] Y. H. Lee, K. P. Reddy, and C. M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, 2003.

[6] Y. Lin, S. A. Brandt, D. D. E. Long, and E. L. Miller. Power conservation strategies for mems-based storage devices. In *Proceedings of the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2002)*, Oct. 2002. Fort Worth, TX.

[7] Y. H. Lu, L. Benini, and G. Micheli. Operating-system directed power reduction. In *International Symposium on Low Power Electronics and Design*, 2000.

[8] Y. H. Lu, L. Benini, and G. Micheli. Requester-aware power reduction. In *International Symposium on System Synthesis*, pages 18–23. Stanford University, September 2000.

[9] Y. H. Lu, L. Benini, and G. Micheli. Power-aware operating systems for interactive systems. *IEEE Transactions on Very Large Scale Integration Systems*, 10(2):119–134, April 2002.

[10] G. Marsh. Data storage gets to the point. In *Materials Today*, Feb. 2003.

[11] Microsoft OnNow power management architecture. http://www.microsoft.com/whdc/hwdev/tech/onnow/OnNowApp_Print.mspx.

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

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

[14] V. Swaminathan and K. Chakrabarty. Dynamic I/O power management in real-time systems. In *Proc. International Conference on Information Fusion (FUSION 2002)*, pages 965–972, 2002.

[15] V. Swaminathan and K. Chakrabarty. Pruning-based energy-optimal device scheduling in hard real-time systems. In *Proc. International Symposium on Hardware/Software Co-Design*, pages 175–180, 2002.

[16] V. Swaminathan and K. Chakrabarty. Energy-conscious, deterministic I/O device scheduling in hard real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, 22:847–858, July 2003.

[17] Texas instruments tms320c6411 dsp. http://focus.ti.com/lit/an/spra373a/spra373a.pdf.

[18] T. S. Tia. *Utilizing Slack Time for aperiodic and sporadic requests scheduling in real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computing Science, 1995. J. W.-S. Liu, adviser.

[19] P. Vettiger, M. Despont, U. Drechsler, U. Durig, W. Haberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The 'millipede' - more than one thousand tips for future afm data storage. *IBM Journal of Research and Development*, 44(3):323–340, 2000.

[20] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.