

# Pruning-Based, Energy-Optimal, Deterministic I/O Device Scheduling for Hard Real-Time Systems

VISHNU SWAMINATHAN and KRISHNENDU CHAKRABARTY  
Duke University

---

Software-controlled (or dynamic) power management (DPM) in embedded systems has emerged as an attractive alternative to inflexible hardware solutions. However, DPM via I/O device scheduling for hard real-time systems has received relatively little attention. In this paper, we present an offline I/O device scheduling algorithm called energy-optimal device scheduler (EDS). For a given set of jobs, it determines the start time of each job such that the energy consumption of the I/O devices is minimized. EDS also ensures that no real-time constraint is violated. The device schedules are provably energy optimal under hard real-time job deadlines. Temporal and energy-based pruning are used to reduce the search space significantly. Since the I/O device scheduling problem is  $\mathcal{NP}$ -complete, we also describe a heuristic called maximum device overlap (MDO) to generate near-optimal solutions in polynomial time. We present experimental results to show that EDS and MDO reduce the energy consumption of I/O devices significantly for hard real-time systems.

Categories and Subject Descriptors: B.8.7 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Performance Attributes; D.4.7 [**Operating Systems**]: Organization and Design; Real-Time Systems and Embedded Systems

General Terms: Performance, Theory

Additional Key Words and Phrases: Schedulability analysis, device scheduling, I/O devices, hard real-time systems

---

This research was supported in part by DARPA under grant no. N66001-001-8946, and in part by a graduate fellowship from the North Carolina Networking Initiative. It was also sponsored in part by DARPA, and administered by the Army Research Office under Emergent Surveillance Plexus MURI award no. DAAD19-01-1-0504. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

A preliminary version of this paper appeared in *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, CO, May 2002, 175–181.

Authors' address: V. Swaminathan and K. Chakrabarty, Department of Electrical and Computer Engineering, Duke University, 130 Hudson Hall, Box 90291, Durham, NC 27708; email:vishnus@ee.duke.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2005 ACM 1539-9087/05/0200-0141 \$5.00

## 1. INTRODUCTION

Computing systems can be broadly classified into two distinct categories—general purpose and embedded. Embedded systems are usually intended for a specific application. Many such systems are I/O intensive, and most of them require real-time guarantees during operation. Examples of embedded systems include remote sensors, digital cellular phones, audio and video disk players, sonar, radar, magnetic resonance imaging (MRI) medical systems, video telephones, and missile systems.

Power consumption is an important design issue in embedded systems. The power and energy consumptions of the various components of an embedded system directly influence battery lifetime, and hence the lifetime of the system. Therefore, system lifetime can be extended by reducing energy consumption in an embedded system. Decreased power consumption also results in higher component reliability. Many embedded systems tend to be situated at remote locations; the cost of replacing battery packs is high when the batteries that power these systems fail.

Power reduction techniques can be viewed as being static or dynamic. Power can be reduced statically by applying compile-time optimizations to generate restructured, power-conscious machine code [AbouGhazaleh et al. 2001]. Static techniques can also be applied at design time to synthesize power-optimized hardware [Alidina et al. 1994]. Dynamic power reduction techniques are used during run-time to take advantage of variations in run-time workload.

The use of static power reduction techniques only can result in a system that is relatively inflexible to changes in the operating environment. Although the use of static techniques result in significant energy savings, recent research has focused more on dynamic power management techniques. These techniques usually take advantage of the features provided by the underlying hardware to obtain further energy savings.

*Dynamic power management* (DPM) refers to the methodology in which power management decisions are made at run-time to take advantage of variations in system workload and resources. Modern hardware designs provide several features to support DPM. These features include multiple *power states* in I/O devices and variable-voltage processors. Given these features, a DPM scheme can make intelligent decisions about changing the operating voltage of a processor, called dynamic voltage scaling (DVS), and switching devices to low-power *sleep* states during periods of inactivity. DVS has emerged as an effective energy-reduction technique by utilizing the fact that energy consumption of a CMOS processor is quadratically proportional to its operating voltage. I/O-centric DPM techniques identify time intervals where I/O devices are not used and switch these devices to low-power modes during these intervals. Such techniques can be implemented at both the hardware and software levels.

More recently, DPM at the operating system (OS) level has gained importance due to its flexibility and ease of use. The OS has a global view of system resources and workload and can therefore can make intelligent power-management decisions in a dynamic and flexible manner. The Advanced Configuration and Power Interface (ACPI) standard, introduced in 1997, allows

hardware power states to be controlled by the OS through system calls, effectively transferring the power reduction responsibility from the hardware (BIOS) to the software (OS).

A number of embedded systems are designed for real-time use. These systems must be designed to meet both functional *and* timing requirements [Buttazzo 1997]. Thus, the correct behavior of these systems depends not only on the accuracy of computations but also on their timeliness. Systems in which the violation of these timing requirements can result in catastrophic consequences are termed *hard real-time systems*. Any real-time scheduling algorithm must guarantee timeliness and schedule tasks so that the deadline of every task is met. Energy minimization adds a new dimension to these design issues.

In modern computer systems, the CPU and the I/O subsystem are among the major consumers of power. While reducing CPU power results in significant energy savings, the I/O subsystem is also a potential candidate to target for power reduction in I/O-intensive systems. However, switching between device power states has associated time and power penalties, that is, a device takes a certain amount of time and power to transition between its power states. In real-time systems where tasks have associated deadlines, this switching must be performed with great caution to avoid the consequences of tasks missing their deadlines. Current-day practice involves keeping devices in hard real-time systems powered up during the entirety of system operation; the critical nature of I/O devices operating in real-time prohibits the shutting down of devices during runtime in order to avoid the catastrophic consequences of missed task deadlines.

In this paper, we present a nonpreemptive optimal offline scheduling algorithm to minimize the energy consumption of the I/O devices in hard real-time systems. In safety-critical applications, offline scheduling is often preferred over priority-based run-time scheduling to achieve high predictability [Xu and Parnas 2000]. In systems where offline scheduling is used, the problem of scheduling tasks for minimum I/O energy can be readily addressed through the approach presented here. We refer to the algorithm that is described in this paper as the energy-optimal device scheduler (EDS). For a given job set, EDS determines the start time of each job such that the energy consumption of the I/O devices is minimized, while guaranteeing that no real-time constraint is violated. EDS uses a tree-based branch-and-bound approach to identify these start times. In addition, EDS provides a sequence of states for the I/O devices, referred to as the I/O device schedule, that is provably energy-optimal under hard real-time job deadlines. Temporal and energy-based pruning are used to reduce the search space significantly. We show that the I/O device scheduling problem is  $\mathcal{NP}$ -complete, and we present a heuristic called maximum device overlap (MDO) to generate near-optimal solutions in polynomial time. Experimental results are presented to show that EDS and MDO reduce the energy consumption of I/O devices significantly for hard real-time systems.

The rest of the paper is organized as follows. In Section 2, we review related prior work on DVS and I/O device scheduling. In Section 3 we present a formal statement of our problem, including the terminology used in the paper and our assumptions. We prove that the minimum-energy I/O device scheduling problem for hard real-time systems is  $\mathcal{NP}$ -complete. In Section 4, we explain

the pruning technique that plays a pivotal role in the reduction of the search space. In Section 5, we describe the energy-optimal EDS algorithm. In Section 6, we describe the MDO heuristic with  $O(pH^2)$  complexity, where  $p$  is the number of devices in the system and  $H$  is the hyperperiod. In Section 7 we present our experimental results. Finally, in Section 8, we summarize the paper and outline directions for future research.

## 2. RELATED PRIOR WORK

The past decade has seen a significant body of research on low-power design methodologies. This research has focused primarily on reducing the power consumption of the CPU and I/O devices. We first review DPM methods for the CPU. In Yao et al. [1995], a minimum-energy, offline preemptive task scheduling algorithm is presented. This method identifies *critical intervals* (intervals in which groups of tasks must run at a constant maximum voltage in any optimal schedule) in an iterative fashion. These tasks are then scheduled using the EDF scheduling policy [Liu and Layland 1973]. An online scheduling algorithm for the preemptive task model is presented in Hong et al. [1998]. The algorithm guarantees that all periodic task deadlines are met. It also accepts aperiodic tasks whose deadlines are guaranteed to be met (the guarantee is provided by an acceptance test). In Ishihara and Yasuura [1998], the authors consider the problem of statically assigning voltages to tasks using an integer linear programming formulation. They show that energy is minimized only if a task completes exactly at its deadline and that at most two voltages are required to emulate an ideal voltage level (in the case where only discrete frequencies are allowed). In Shin and Choi [1999], an online DVS technique based on the rate-monotonic algorithm (RMA) is presented. This approach uses the fixed-priority implementation model described in Katcher et al. [1993]. The method presented in Shin and Choi [1999] identifies time instants at which processor speed can be scaled down to reduce power consumption, while guaranteeing that no task deadlines are missed. This work is extended in Shin et al. [2000]. The authors improve upon their prior work by first performing an offline schedulability analysis of the task set and determining the minimum possible speed at which all tasks meet their deadlines. An online component then takes advantage of run-time slack that is generated, for example, when tasks do not all run at their estimated worst-case computation times. In Quan and Hu [2001], a near-optimal offline fixed-priority scheduling scheme is presented. This is extended in Quan and Hu [2002] to generate optimal solutions for the DVS problem. An online slack estimation method is presented in Kim et al. [2002] that is used to dynamically vary processor voltage under dynamic priority scheduling schemes.

Almost all prior work on DPM techniques for I/O devices has focused primarily on scheduling devices in a non-real-time environment. I/O-centric DPM methods 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 next request generated by a task for a device that has been shut down wakes it up. The

device then proceeds to service the request. Predictive schemes are more readily adaptable to changing workloads than timeout-based schemes. Predictive schemes such as the one described in Hwang and Hu [1997] attempt to predict the length of the next idle period based on the past observation of requests. In Lu et al. [2000], a device-utilization matrix keeps track of device usage and a processor-utilization matrix keeps track of processor usage of a given task. When the utilization of a device falls below a threshold, the device is put into the sleep state. In Chung et al. [1999], devices with multiple sleep states are considered. Here too, the authors use a predictive scheme to shut down devices based on adaptive learning trees. Stochastic methods usually involve modeling device requests through different probabilistic distributions and solving stochastic models (Markov chains and their variants) to obtain device switching times [Benini et al. 1999; Simunic et al. 2001]. In Irani et al. [2002], a theoretical approach based on the notion of competitive ratio is developed to compare different DPM strategies. The authors also present a probabilistic DPM strategy where the length of the idle period is determined through a known probabilistic distribution.

An important observation that we make here is that none of the above I/O-DPM methods are viable candidates for use in real-time systems. Due to their inherently probabilistic nature, the applicability of the above methods to real-time systems falls short in one important aspect—real-time temporal guarantees cannot be provided. Shutting down a device at the wrong time can potentially result in a task missing its deadline (this is explained in greater detail in Section 3. Although significantly prolonging battery life, most methods that have been described in the literature thus far target non-real-time systems, where average task response time (rather than deadline) is an important design parameter. In non-real-time systems, a small delay in computation can be tolerated. In hard real-time systems, meeting deadlines is of critical importance, and therefore, it becomes apparent that new algorithms that operate in a more deterministic manner are needed in order to guarantee real-time behavior.

A recent approach for I/O device scheduling for real-time systems relies on the notion of a mode dependency graph (MDG) for multiple processors and I/O devices [Li et al. 2002]. An algorithm based on topological sorting is used to generate a set of valid mode combinations. A second algorithm then determines a sequence of modes for each resource such that all timing constraints are met and max-power requirements are satisfied for a given task set. A schedule generated in Li et al. [2002] is not necessarily an energy-optimal schedule for the task set. Furthermore, the work in Li et al. [2002] does not distinguish between I/O devices and processors. On the other hand, the model we assume is that of a set of periodic tasks executing on a single processor. These tasks use a given set of I/O devices. We only consider offline device scheduling; two online I/O-based DPM algorithms are described in Swaminathan and Chakrabarty [2003].

DVS for real-time multiprocessor systems has been studied in Luo and Jha [2001, 2002], and Zhang et al. [2002]. In Luo and Jha [2001], the authors first perform a static voltage assignment to a set of real-time tasks with precedence constraints. A dynamic voltage scheme is also proposed that handles soft aperiodic tasks, while also adjusting clock frequency at run-time to utilize any

excess slack that is generated. The authors of Zhang et al. [2002] follow an approach similar to the one described in Luo and Jha [2001]. They use a variant of the latest-finish-time heuristic to perform task allocation, and then present an integer linear programming model to optimally identify the clock frequencies for the task set. Minimizing communication power in real-time multiprocessor systems has been considered recently in Liu et al. [2002]. The authors assume a multiprocessor system, with each node having a voltage-scalable processor, and a communication channel. The network interface at each node can transmit and receive at different speeds, with corresponding power levels. Each process on a node consists of three segments—a receive segment, a local processing segment, and a send segment. Each process also has an associated deadline. The problem addressed in Liu et al. [2002] is to identify a communication speed and a processor speed for each node such that the global energy consumption is minimized.

In the next section, we present our problem statement and describe the underlying assumptions.

### 3. NOTATION, PROBLEM STATEMENT, AND COMPLEXITY ANALYSIS

In this section, we present the problem statement, our notation, and underlying assumptions. We also show that the problem we address is  $\mathcal{NP}$ -complete.

#### 3.1 Notation and Problem Statement

We are given a task set  $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_n\}$  of  $n$  periodic tasks. Associated with each task  $\tau_i \in \mathcal{T}$  are the following parameters:

- its *release* (or arrival) time  $a_i$ ,
- its period  $p_i$ ,
- its deadline  $d_i$ ,
- its execution time  $c_i$ , and
- a *device usage list*  $L_i$ , consisting of all the I/O devices used by  $\tau_i$ .

The hyperperiod  $H$  of the task set is defined as the least common multiple of the periods of all tasks. We assume that the deadline of each task is equal to its period, that is,  $p_i = d_i$ . Associated with each *task set*  $\mathcal{T}$  is a *job set*  $\mathcal{J} = \{j_1, j_2, \dots, j_l\}$  consisting of all the instances of each task  $\tau_i \in \mathcal{T}$ , arranged in ascending order of arrival time, where  $l = \sum_{k=1}^n H/p_k$ . Except for the period, a job inherits all properties of the task of which it is an instance. This transformation of a pure periodic task set into a job set does not introduce significant overhead because optimal I/O device schedules are generated offline, where scheduler efficiency is not a pressing issue.

The system also uses a set  $K = \{k_1, k_2, \dots, k_p\}$  of  $p$  I/O devices. Each device  $k_i$  has the following parameters:

- two power states—a low-power sleep state  $ps_{l,i}$  and a high-power working state  $ps_{h,i}$ ,
- a transition time from  $ps_{l,i}$  to  $ps_{h,i}$  represented by  $t_{wu,i}$ ,
- a transition time from  $ps_{h,i}$  to  $ps_{l,i}$  represented by  $t_{sd,i}$ ,

- power consumed during wake-up  $P_{wu,i}$ ,
- power consumed during shutdown  $P_{sd,i}$ ,
- power consumed in the working state  $P_{w,i}$ , and
- power consumed in the sleep state  $P_{s,i}$ .

We assume that the worst-case execution times of the tasks are greater than the transition time of the devices. We make this assumption to ensure that the states of the I/O devices are clearly defined at the completion of the jobs. Although we assume here that the devices have only a single sleep state, our algorithms can also generate energy-optimal device schedules for devices with multiple low-power states. We explain this in greater detail in Section 5.

We assume, without loss of generality, that for a device  $k_i$ ,  $t_{wu,i} = t_{sd,i} = t_{0,i}$  and  $P_{wu,i} = P_{sd,i} = P_{0,i}$ . The energy consumed by device  $k_i$  is given by  $E_i = P_{w,i}t_{w,i} + P_{s,i}t_{s,i} + mP_0t_0$ , where  $m$  is the number of state transitions,  $t_{w,i}$  is the total time spent by device  $k_i$  in the working state, and  $t_{s,i}$  is the total time spent in the sleep state, assuming that all devices possess only two power states. The problem  $\mathcal{P}_{io}$  that we address in this paper is formally stated below:

- $\mathcal{P}_{io}$ : Given a job set  $\mathcal{J}$  that uses a set  $\mathcal{K}$  of I/O devices, identify a set of start times  $\mathcal{S} = \{s_1, s_2, \dots, s_l\}$  for the jobs such that the total energy consumed  $\sum_{i=1}^p E_i$  by the set  $\mathcal{K}$  of I/O devices is minimized and all jobs meet their deadlines.

This set of start times, or schedule, provides a minimum-energy device schedule. Once a task schedule has been determined, a corresponding device schedule is generated by determining the state of each device at the start and completion of each job based on its device-usage list.

Requests can be processed by the devices only in the working state. All I/O devices used by a job must be powered-up before it starts execution. There are no restrictions on the time instants at which device states can be switched. The I/O device schedule that is computed offline is loaded into memory, and a timer controls the switching of the I/O devices at run-time. Such a scheme can be implemented in systems where tick-driven scheduling is used. We assume that all devices are powered up at time  $t = 0$ .

Incorrectly switching power states can cause increased, rather than decreased, energy consumption for an I/O device. This leads to the concept of *breakeven time*, which is the time interval for which a device in the powered-up state consumes an energy exactly equal to the energy consumed in shutting a device down, leaving it in the sleep state and then waking it up Hwang and Hu [1997]. Figure 1 illustrates this concept. If any idle time interval for device  $k_i$  is greater than its breakeven time  $t_{be,i}$ , energy is saved by powering  $k_i$  down. For idle intervals that are less than the breakeven time interval, energy is saved by keeping the device in the powered-up state. Device switching for reduced energy consumption therefore results in latencies of  $2t_{be,i}$  because devices cannot be used during state-transitions. However, by performing task scheduling to minimize device energy, we adjust the task schedule such that the number of idle intervals of length  $2t_{be,i}$  or greater are maximized.

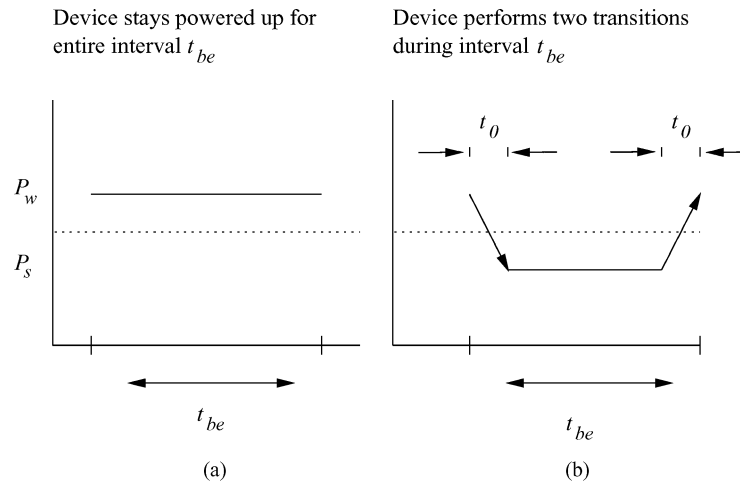


Fig. 1. Illustration of breakeven time. The time interval for which the energy consumptions are the same in (a) and (b) is called the breakeven time.

It is easy to show that the decision version of problem  $\mathcal{P}_{io}$  is  $\mathcal{NP}$ -complete. We first show that  $\mathcal{P}_{io} \in \mathcal{NP}$  and then show that  $\mathcal{P}_{io}$  is  $\mathcal{NP}$ -hard using the method of restriction. We restate  $\mathcal{P}_{io}$  in the form of a decision problem.

- **INSTANCE:** Set  $\mathcal{J}$  of jobs that uses a set  $\mathcal{K}$  of I/O devices and a positive constant  $B$ .
- **QUESTION:** Is there a feasible schedule for  $\mathcal{J}$  such that the energy consumed by the set  $\mathcal{K}$  of I/O devices is at most  $B$ ?

A nondeterministic algorithm can generate a job schedule and compute the energy consumed by the set of devices, and also check in polynomial time if the energy consumption is at most  $B$ . To show that  $\mathcal{P}_{io}$  is  $\mathcal{NP}$ -hard, we use the method of restriction. Consider a special case where  $K = \phi$ , that is, no devices are used. The decision problem  $\mathcal{P}_{io}$  then reduces to the *sequencing within intervals* problem, which is known to be  $\mathcal{NP}$ -complete [Garey and Johnson 1977]. Thus  $\mathcal{P}_{io}$  is  $\mathcal{NP}$ -complete.

Although  $\mathcal{P}_{io}$  is  $\mathcal{NP}$ -complete, it can be solved optimally for moderate-sized problem instances. In the following section, we present our approach to solving  $\mathcal{P}_{io}$  and the underlying theory.

#### 4. PRUNING TECHNIQUE

We generate a schedule tree and iteratively prune branches when it can be guaranteed that the optimal solution does not lie along those branches. The schedule tree is pruned based on two factors—time and energy. Temporal pruning is performed when a certain partial schedule of jobs causes a missed deadline deeper in the tree. The second type of pruning—which we call *energy pruning*—is the central idea on which EDS is based. The remainder of this section explains the generation of the schedule tree and the pruning techniques that are employed. We illustrate these through the use of an example.

Table I. Example Task Set  $\mathcal{T}1$ 

Task	Arrival Time	Completion Time	Period (Deadline)	Device-Usage List
$\tau_1$	0	1	3	$k_1$
$\tau_2$	0	2	4	$k_2$

 Table II. List of Jobs for Task Set  $\mathcal{T}1$  from Table I

	$j_1$	$j_2$	$j_3$	$j_4$	$j_5$	$j_6$	$j_7$
$a_i$	0	0	3	4	6	8	9
$c_i$	1	2	1	2	1	2	1
$d_i$	3	4	6	8	9	12	12

A vertex  $v$  of the tree is represented as a 3-tuple  $(i, t, e)$ , where  $i$  is a job  $j_i$ ,  $t$  is a valid start time for  $j_i$ , and  $e$  represents the energy consumed by the devices until time  $t$ . An edge  $z$  connects two vertices  $(i, t, e)$  and  $(k, l, m)$  if job  $j_k$  can be successfully scheduled at time  $l$  given that job  $j_i$  has been scheduled at time  $t$ . A path from the root vertex to any intermediate vertex  $v$  has an associated order of jobs that is termed a *partial schedule*. A path from the root vertex to a leaf vertex constitutes a *complete schedule*. A *feasible schedule* is a complete schedule in which no job misses its associated deadline. Every complete schedule is a feasible schedule (temporal pruning eliminates all *infeasible* partial schedules).

An example task set  $\mathcal{T}1$  consisting of two tasks is shown in Table I. Each task has an arrival time, a worst-case execution time, and a period. We assume that the deadline for each task is equal to its period. Task  $\tau_1$  uses device  $k_1$ , and task  $\tau_2$  uses device  $k_2$ . Table II lists the instances of the tasks, arranged in increasing order of arrival. In this example, we assume a working power of 6 units, a sleep power of 1 unit, a transition power of 3 units, and a transition time of 1 unit.

We now explain the generation of the schedule tree for the job set shown in Table II. The root vertex of the tree is a dummy vertex. It is represented by the 3-tuple  $(0, 0, 0)$  that represents dummy job  $j_0$  scheduled at time  $t = 0$  with an energy consumption of 0 units. We next identify all jobs that are released at time  $t = 0$ . The jobs that are released at  $t = 0$  for this example are  $j_1$  and  $j_2$ . Job  $j_1$  can be scheduled at times  $t = 0, t = 1$ , and  $t = 2$  without missing its deadline. We also compute the energy consumed by all the devices up to times  $t = 0, t = 1$ , and  $t = 2$ . The energy values are 0, 8, and 10 units, respectively (Figure 2 explains the energy calculation procedure). We therefore draw edges from the dummy root vertex to vertices  $(1, 0, 0)$ ,  $(1, 1, 8)$ , and  $(1, 2, 10)$ . Similarly, job  $j_2$  can be scheduled at times  $t = 0, t = 1$ , and  $t = 2$  and the energy values are 0, 8, and 10 unit, respectively. Thus, we draw three more edges from the dummy vertex to vertices  $(2, 0, 0)$ ,  $(2, 1, 8)$ , and  $(2, 2, 10)$ . Note that job  $j_2$  would miss its deadline if it were scheduled at time  $t = 3$  (since it has an execution time of 2 units). Therefore, no edge exists from the dummy node to node  $(2, 3, e)$ , where  $e$  is the energy consumption up to time  $t = 3$ . Figure 3 illustrates the tree after one job has been scheduled. Each level of depth in the tree represents one job being successfully scheduled.

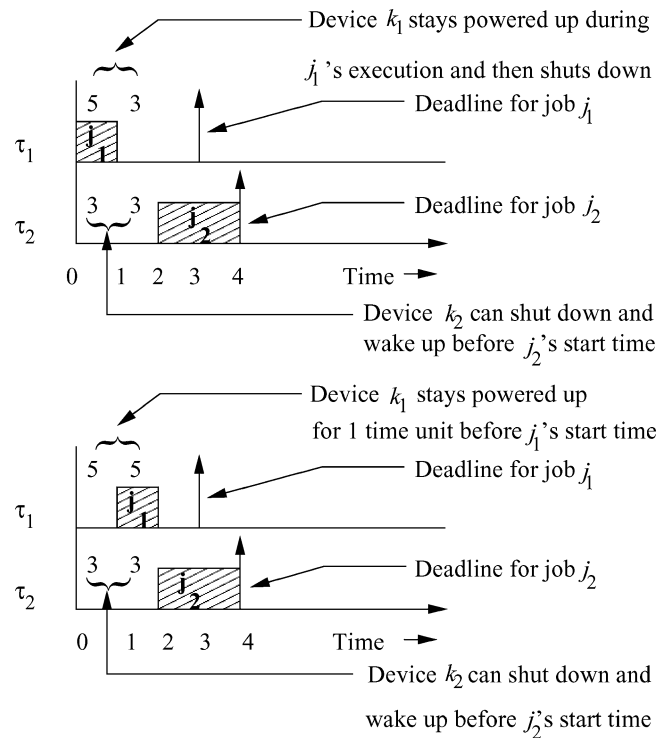


Fig. 2. Calculation of energy consumption.

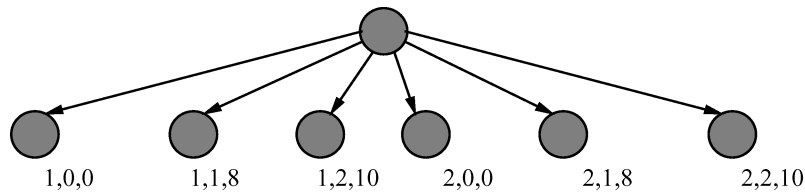


Fig. 3. Partial schedules after 1 scheduled job.

We then proceed to the next level. We examine every vertex at the previous level and determine the jobs that can be scheduled next. By examining node  $(1, 0, 0)$  at level 1, we see that job  $j_1$  would complete its execution at time  $t = 1$ . The only other job that has been released at  $t = 1$  is job  $j_2$ . Thus,  $j_2$  can be scheduled at times  $t = 1$  and  $t = 2$  after job  $j_1$  has been scheduled at  $t = 0$ . The energies for these nodes are computed, and edges are drawn from  $(1, 0, 0)$  to  $(2, 1, 10)$  and  $(2, 2, 14)$ . Similarly, examining vertex  $(1, 1, 8)$  results in vertex  $(2, 2, 16)$  at level 2. The next vertex at level 1—vertex  $(1, 2, 10)$ —results in a missed deadline at level 2. If job  $j_1$  were scheduled at  $t = 2$ , it would complete execution at time  $t = 3$ . The earliest time at which  $j_2$  could be scheduled is  $t = 3$ ; however, even if it were scheduled at  $t = 3$ , it would miss its deadline. Thus, scheduling  $j_1$  at  $t = 2$  does not result in a feasible schedule. This branch can hence be pruned. Similarly, the other nodes at level 1 are examined, and

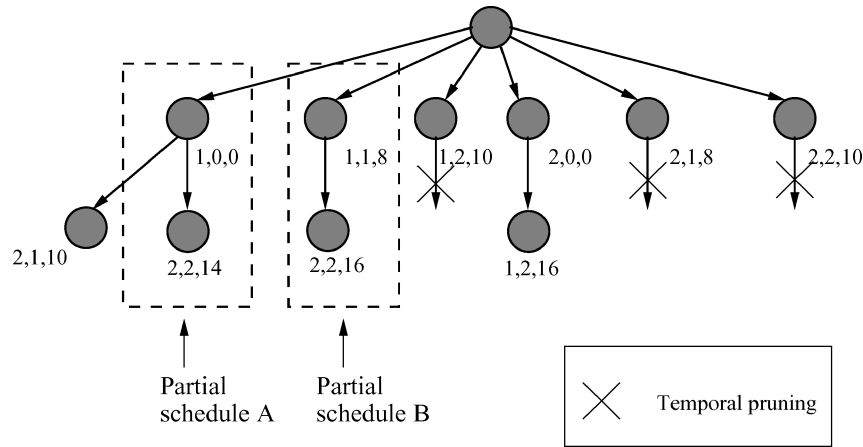


Fig. 4. Partial schedules after two scheduled jobs.

the unpruned partial schedules are extended. Figure 4 illustrates the schedule tree after two jobs have been scheduled. The edges that have been crossed out represent branches that are not considered due to temporal pruning.

At this point, we note that vertices  $(2, 2, 14)$  and  $(2, 2, 16)$  represent the same job ( $j_2$ ) scheduled at the same time ( $t = 2$ ). However, the energy consumptions for these two vertices are different. This observation leads to the following theorem:

**THEOREM 1.** *When two vertices at the same tree depth representing the same job being scheduled at the same time can be reached from the root vertex through two different paths, and the orders of the previously scheduled jobs along the two partial schedules are identical, then the partial schedule with higher energy consumption can be eliminated without losing optimality.*

**PROOF.** Let us call the two partial schedules at a given depth schedule A and schedule B, with schedule A having lower energy consumption than schedule B. We first note that schedule B has higher energy consumption than schedule A because one or more devices have been in the powered-up state for a longer period of time than necessary in schedule B. Assume that  $i$  jobs have been scheduled, with job  $j_i$  being the last scheduled job. Since we assume that the execution times of all jobs are greater than the maximum transition time of the devices, it is easy to see that the state of the devices at the end of job  $j_i$  will be identical in both partial schedules. By performing a time translation (mapping the end of job  $j_i$ 's execution to time  $t = 0$ ), we observe that the resulting schedule trees are identical in both partial schedules. However, all schedules in schedule B after time translation will have an energy consumption that is greater than their counterparts in schedule A by an energy value  $E_\delta$ , where  $E_\delta$  is the energy difference between schedules A and B. It is also easy to show that the energy consumed *during* job  $j_i$ 's execution in schedule A will always be less than or equal to  $j_i$ 's execution in schedule B. This completes the proof of the theorem.  $\square$

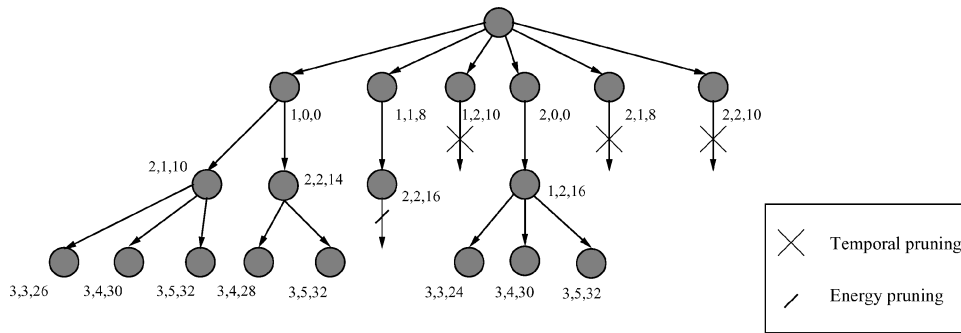


Fig. 5. Partial schedules after three scheduled jobs.

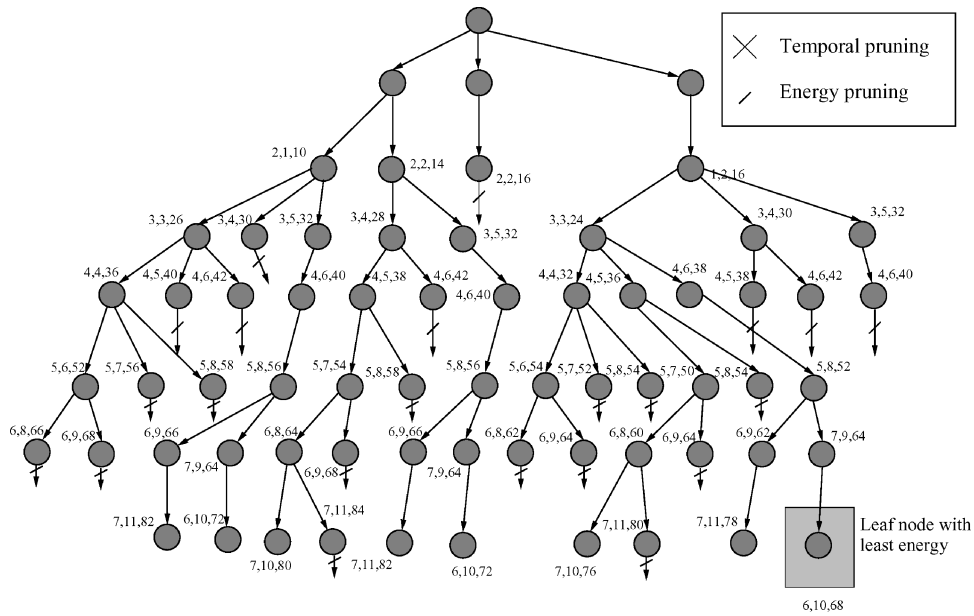


Fig. 6. Complete schedule tree.

The application of this theorem to the above example results in partial schedule B in Figure 4 being discarded. As one proceeds deeper down the schedule tree, there are more vertices such that the partial schedules corresponding to the paths to them from the root vertex are identical. It is this “redundancy” that allows for the application of Theorem 1, which consequently results in tremendous savings in memory while still ensuring that an energy-optimal schedule is generated. By iteratively performing this sequence of steps (vertex generation, energy calculation, vertex comparison, and pruning), we generate the complete schedule tree for the job set. Figure 5 illustrates the partial schedules after three jobs have been scheduled for our example. The complete tree is shown in Figure 6. We have not shown paths that have been temporally pruned. The edges that have been crossed out with horizontal slashes represent

**Procedure** EDS( $\mathcal{J}, l$ )  
 $\mathcal{J}$ : Job set.  
 $l$ : Number of jobs.  
openList: List of unexpanded vertices.  
currentList: List of vertices at the current depth.  
 $t$ : time counter.

1. Set  $t = 0$ ; Set  $d = 0$ ;
2. Add vertex  $(0,0)$  to openList;
3. **for each** vertex  $v = (j_i, time)$  in openList {
4.   Set  $t = time + c_i$ ;
5.   Find set of all jobs  $\mathcal{J}'$  released up to time  $t$ ;
6.   **for each** job  $j \in \mathcal{J}'$  {
7.     **if**  $j$  has been previously scheduled
8.     **continue**;
9.     **else** {
10.       Find all possible scheduling instants for  $j$ ; /\* Temporal pruning\*/
11.       Compute energy for each generated vertex;
12.       Add generated vertices to currentList;
13.     }
14.   }
15. **for each** pair of vertices  $v_1, v_2$  in currentList {
16.   **if**  $j_1 = j_2$  and  
      partial schedule( $v_1$ ) = partial schedule( $v_2$ ) {
17.     **if**  $E_{v_1} > E_{v_2}$
18.       Prune  $v_1$ ;
19.     **else**
20.       Prune  $v_2$ ;
21.   }
22. }
23. Add unpruned vertices in currentList to openList;
24. Clear currentList;
25. Increment  $d$ ;
26.   **If**  $d = l$
27.     **Terminate**.
28. }

Fig. 7. Pseudocode description of EDS.

energy-pruned branches. The energy-optimal device schedule can be identified by tracing the path from the highlighted node to the root vertex in Figure 6.

## 5. THE EDS ALGORITHM

The pseudocode for EDS is shown in Figure 7. EDS takes as input a job set  $\mathcal{J}$  and generates all possible nonpreemptive minimum energy schedules for the given job set. The algorithm operates as follows. The time counter  $t$  is set to 0, and openList is initialized to contain only the root vertex  $(0, 0, 0)$  (lines 1 and 2). In lines 3–10, every vertex in openList is examined and nodes are generated at the succeeding level. Next, the energy consumptions are computed for each of these newly generated vertices (line 11). Lines 15–20 correspond to the pruning technique. For every pair of replicated vertices, the partial schedules are

checked and the one with the higher energy consumption is discarded. Finally, the remaining vertices in `currentList` are appended to `openList`. `currentList` is then reset. This process is repeated until all the jobs have been scheduled, that is, the depth of the tree equals the total number of jobs (lines 25–28). Note that several schedules can exist with a given energy consumption for a given job set. EDS generates all possible unique schedules with a given energy for a given job set. One final comparison of all these unique schedules results in the set of schedules with the absolute minimum energy.

Devices with multiple low-power sleep states can be handled simply by iterating through the list of low-power sleep states and identifying the sleep state that results in the most energy savings for a given idle interval. However, the number of allowed sleep states is limited by our assumption that the transition time from a given low-power sleep state is less than the worst-case completion time of the task.

The EDS algorithm attempts to find an optimal solution for an  $\mathcal{NP}$ -complete problem. Hence, despite the pruning techniques that it employs, it can be expected to require excessive memory and computation time for large problem instances. We have therefore developed a heuristic method to generate near-optimal solutions in polynomial time. We refer to it as the maximum device overlap (MDO) heuristic.

## 6. MAXIMUM DEVICE OVERLAP HEURISTIC

The MDO algorithm uses a real-time scheduling algorithm to generate a feasible real-time job schedule and then iteratively swaps job segments to reduce energy consumption. MDO is efficient for large problem instances because, unlike EDS, it generates I/O device schedules for preemptive schedules. The preemptive scheduling with arrival times and deadlines has been shown to be solvable in polynomial time [Lawler 1973]. Thus, the MDO algorithm is also a polynomial time algorithm, with a computational complexity of  $O(pH^2)$ , where  $p$  is the number of devices used and  $H$  is the hyperperiod (note that  $H \leq \prod_{i=1}^n p_i$ , where  $p_i$  is the period of task  $\tau_i$ ). The pseudocode for the MDO heuristic is shown in Figure 8.

The algorithm takes as input a feasible schedule  $S$  of jobs. This feasible schedule  $S$  is generated using a real-time scheduling algorithm such as RM or EDF. The algorithm operates in the following manner. At the completion of each job, the algorithm finds the next schedulable job with a device-usage list closest to the device-usage list of the current job. We refer to the number of devices that are common to two jobs as device overlap, that is, for two jobs  $j_i$  and  $j_{i+1}$ , the device overlap  $do = L_i \cap L_{i+1}$ . Lines 2, 3, and 4 are initializations. In lines 5–19, we select the new schedulable job with the closest device-usage overlap with the current job. In line 8, a check is performed to ensure that job  $j_{i_2}$  is schedulable at time  $t + 1$ . We also check to ensure that swapping the two jobs does not cause a missed deadline. For each job that passes this test, the device-usage overlap with the current job is calculated (lines 9–13) and the one with the highest overlap is chosen for swapping. The two jobs are then swapped in line 20.

**Procedure** MDO( $S, L_i$ )

$S$ : Schedule of jobs.  $S$  is an array with each element in the array representing a unit time slot, with its value representing a pointer to the job (task) that is scheduled during that time slot.

$do$ : device overlap for current job

$mdo$ : maximum device overlap computed so far

$newtask$ : newly selected job with maximum overlap

```

1. for  $t = 1$  to  $H$  {
2.   Set  $j_t = S[t]$ 
3.   Set  $j_{t+1} = S[t+1]$ 
4.    $mdo = 0$ 
5.   for  $t_2 = t + 1$  to  $H$  {
6.      $newtask = S[t_2]$ 
7.     Set  $j_{t_2} = S[t_2]$ 
8.     if  $j_{t_2} \rightarrow a \leq t + 1$  and  $j_{t+1} \rightarrow d \geq t_2$  {
9.       for  $k = 1$  to  $p$  {
10.        if  $k \in L_t$  and  $k \in L_{t_2}$ 
11.           $do = do + 1$ 
12.        }
13.      if  $do > mdo$  {
14.         $mdo = do$ 
15.         $newtask = S[t_2]$ 
16.      }
17.    }
18.  }
19.  Swap( $S[t]$ ,  $newtask$ )
20. }
```

Fig. 8. Pseudocode description of the MDO heuristic.

In our implementation, when the MDO algorithm terminates, the array  $S[t]$  of time slots contains a new schedule of jobs with a lower I/O device energy consumption. It is easy to extract the device schedule from this job schedule. A procedure to extract a device schedule from a job schedule is shown in Figure 9 (this procedure can be used for both EDS and MDO). MDO can be used for devices with multiple power states.

Procedure Extract() takes as inputs the array  $S$  of time slots and the parameters of a device  $k$  (in Figure 9, we assume that the device parameters are implicitly available through the argument  $k$ , and that the power states are sorted in decreasing order of power values). At the start of each job (line 1), the algorithm first checks if device  $k$  is used by the current job (line 2). If it is, the procedure keeps the device in the powered up state. If the device is not used by the current job, there is a possibility that it can be shut down. The procedure then identifies the power state the device can be switched to. The identification of the correct power state is illustrated in lines 3–9. If the time difference between the start of the next job that uses  $k$  and the current scheduling instant is greater than the breakeven time corresponding to device state  $ds$  and also greater than twice the transition time to power state  $ds$ , then the variable state is set to  $ds$ . In line 10, the state of the device is recorded and in line 11, a

```

Procedure Extract( $S, k$ )
 $k$ : device under consideration
 $ps_1$ — $ps_m$ : set of device states for device  $k$ 
 $ds$ : iteration variable
state: device state to switch device  $k$  to
1. At  $s_i$ :
2.   if  $k \notin L_i$  {
3.     Find next job  $j_l$  that uses device  $k$ 
4.      $t_d = s_l - s_i$ 
5.     for  $ds = ps_1$  to  $ps_m$  {
6.       if  $t_d > t_{be,ds}$  and  $t_{be,ds} > 2t_{0,ds}$  {
7.         state =  $ds$ 
8.       }
9.     }
10.    Record shutdown and wakeup time for device  $k$ 
11.    Set timer for device  $k$ 's wakeup
12.  }
13. At  $s_i + c_i$ :
14.  Find next job  $j_l$  that uses device  $k$ 
15.   $t_d = s_l - s_i - c_i$ 
16.  for  $ds = ps_1$  to  $ps_m$  {
17.    if  $t_d > t_{be,ds}$  and  $t_{be,ds} > 2t_{0,ds}$  {
18.      state =  $ds$ 
19.    }
20.  }
21.  Record shutdown and wakeup time for device  $k$ 
22.  Set timer for device  $k$ 's wakeup
23.}

```

Fig. 9. Procedure to extract a device schedule from a given task schedule.

timer is set to wake  $k$  up just in time for job  $j_l$  to begin execution. Lines 13–22 correspond to the device state identification at the completion of job  $j_i$ . Note that here, no check is performed to see if  $k$  is in the current job's device-usage list.

Intuitively, MDO attempts to keep a device in a given state (sleep or powered-up) for as long as possible before switching it to a different state. This algorithm is similar to the one presented in Lu et al. [2000], where device requests are grouped together to keep devices powered-down for extended periods of time in order to reduce energy consumption. However, owing to real-time constraints, there is much less flexibility here than in Lu et al. [2000]. The authors of Lu et al. [2000] focus on device scheduling for interactive systems with no hard timing constraints. Their method, like MDO, attempts to schedule, at every scheduling instant, a task with the maximum device-usage overlap with the current task. However, since they do not consider a real-time task model with periodic arrivals and deadlines, their approach is less constrained than the MDO heuristic. Furthermore, in a hard real-time system, it is generally not advisable to power down devices when tasks that use them are being executed. Thus, MDO and EDS perform intertask device scheduling rather than intratask voltage scheduling, as is done in Lu et al. [2000].

Table III. Experimental Task Set  $T1$ 

Task	Execution Time	Period (Deadline)	Device List
$\tau_1$	1	4	$k_1, k_3$
$\tau_2$	3	5	$k_2, k_3$

While performing preemptive scheduling with I/O resources, task blocking becomes an important issue. Blocking refers to the phenomenon where a task that is executing a critical section of code gets preempted while holding I/O resources that are required by the preempting task. This can potentially result in missed deadlines. Several algorithms have been proposed that address the issue of blocking under fixed-priority and dynamic-priority scheduling policies [Baker 1991; Sha et al. 1990]. For example, the stack resource policy (SRP), described in Baker [1991], requires a preempting task to request all the resources it requires for execution prior to preemption. Preemption is not allowed if the required resources are unavailable. In our MDO algorithm, we do not explicitly address the issue of blocking—we generate a device schedule for a given task schedule and assume that the allocation of resources and the prevention of blocking is performed by an underlying algorithm such as SRP. However, MDO ensures that the start of a job is not delayed by devices that are powered-down and therefore unavailable. In other words, MDO ensures that all devices required by a job are powered-up and ready before the job begins execution, and an underlying blocking-prevention algorithm ensures that deadlock situations do not arise. Such a blocking test can be easily integrated into MDO and can be performed prior to swapping job slices. In the next section, we present experimental results for EDS and MDO.

## 7. EXPERIMENTAL RESULTS

We evaluated EDS and MDO for several periodic task sets with varying hyperperiods and number of jobs. We compare the memory requirement of the tree with the pruning algorithm to the memory requirement of the tree without pruning. Memory requirement is measured in terms of the number of nodes at every level of the schedule tree.

The first experimental task set, shown in Table III, consists of two tasks with a hyperperiod of 20. The device-usage lists for tasks were randomly generated. The device parameters were chosen from real devices that are currently deployed in the field. The devices and their parameters are listed in Table IV. These I/O devices are a representative set of devices commonly used in embedded applications. These I/O devices can be classified into two categories (see Figure 10)<sup>1</sup>. Type I devices are devices that stay in a given state until an explicit power management command is issued. Type II devices are devices that stay in the powered-up state only while processing requests, and then automatically transition to a lower-powered idle state when not in use (note, however, that this idle state is different from a fully powered-down sleep state). The DSP is a device that can be powered-down to a fully low-powered sleep state in a

<sup>1</sup>Figure 10 was provided by Reviewer 3.

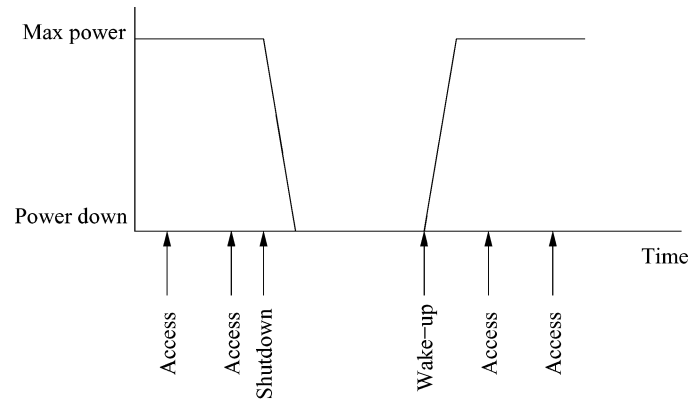
Table IV. Device Parameters Used in Evaluating EDS and MDO

Device $k_i$	Device Type	$P_{w,i}$ (W)	$P_{sd,i} = P_{wu,i} = P_{0,i}$ (W)	$t_{0,i}$ (s)	$P_{s,i}$ (W)
$k_1$	HDD <sup>1</sup>	2.3	1.5	0.02	1.0
$k_2$	NIC <sup>2</sup>	0.3	0.2	0.5	0.1
$k_3$	DSP <sup>3</sup>	0.63	0.4	0.5	0.25

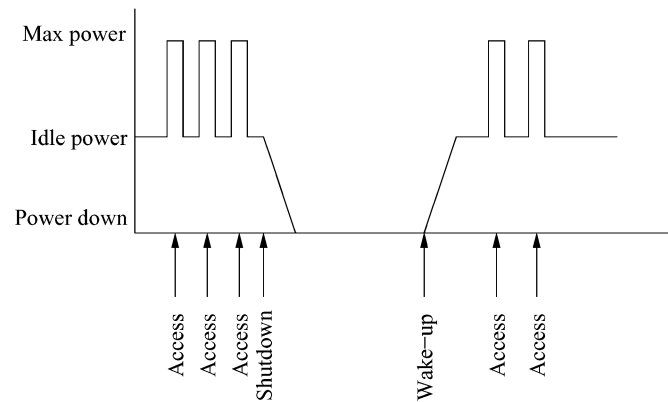
<sup>1</sup>After Fujitsu MHL2300AT Hard Disk Drive Product Manual.

<sup>2</sup>After AMD Am79C874 NetPHY-1LP Low Power 10/100 Tx/Rx Ethernet Transceiver Technical Data sheet.

<sup>3</sup>After TMS320C6411 Power Consumption Summary.



(a) Type I



(b) Type II

Fig. 10. An illustration of power models for I/O devices (courtesy Reviewer 3).

small amount of time and is a good example of a type I device. On the other hand, the disk drive (HDD) has three power states—an active state in which the disk reads and writes data, an intermediate-powered idle state where the spindle and disk platters are still spinning without read/write activity, and a low-power standby state where the spindle is stopped. This is an example of a

Table V. Job Set Corresponding to Experimental Task Set  $\mathcal{T}1$ 

	$j_1$	$j_2$	$j_3$	$j_4$	$j_5$	$j_6$	$j_7$	$j_8$	$j_9$
$a_i$	0	0	4	5	8	10	12	15	16
$c_i$	1	3	1	3	1	3	1	3	1
$d_i$	4	5	8	10	12	15	16	20	20

type II device, where “implicit” power management by the device switches it to an intermediate state that is different from a fully low-powered sleep state. For the disk drive, the power consumed in transitioning from standby to idle/active is 4.5 W and the transition time is typically around 5 s, resulting in a breakeven time of approximately 18 s. This interval of inactivity is rarely seen at run-time, and the disk drive stays in the idle state for the entire hyperperiod. Therefore, we assume here that the sleep power  $P_s$  is the power consumed when the device is in the idle state (spindle is spinning with no read/write activity). The active power  $P_w$  corresponds to the power consumed during actual reading and writing of data, and the transition power  $P_0$  represents the power consumption during a transition from the sleep state (idle) to the active state (read/write) (the transition time between these states is 22 ms [Fujitsu MHL2300AT]).

Although the transitions from the active to idle states are performed “implicitly” (i.e., by the hard disk, without an explicit power-down command), we use this device model since it provides a more accurate picture of energy consumption. During execution of a task, we assume that the disk drive consumes 2.3 W of power to read and write data, and during idle periods, the disk-drive transitions to the intermediate idle state where the platters are still spun-up, but without any read/write activity.

It is also important to note that any explicit device scheduling algorithm operates atop the implicit power management performed by the device itself. However, explicit power management yields greater energy savings than implicit power management because devices can be switched to lower-powered states earlier, thereby enabling them to save greater amounts of energy than implicit power management.

Expansion of the task set in Table III results in the job set shown in Table V. Figures 11(a) and 11(b) show the task and device schedules generated for the task set in Table III using the fixed-priority rate-monotonic scheduling algorithm Liu and Layland [1973]. Since device  $k_3$  is used by both tasks, it stays powered up throughout the hyperperiod. The device schedule for  $k_3$  is therefore not shown in Figure 11.

If all devices are powered up throughout the hyperperiod, the energy consumed by the I/O devices for any task schedule is 66 J. Figure 12 shows an optimal task schedule generated using EDS. The energy consumption of the optimal task (device) schedule is 44 J, resulting in a 33% reduction in energy consumption.

From Figure 11(b), we see that device  $k_2$  stays powered-up for almost the entire hyperperiod and device  $k_1$  performs 10 transitions over the hyperperiod. Moreover, device  $k_2$  stays powered up even when it is not in use due to the fact that there is insufficient time for shutting down and powering the device back up. By examining Figures 12(a) and 12(b), we deduce that minimum energy will

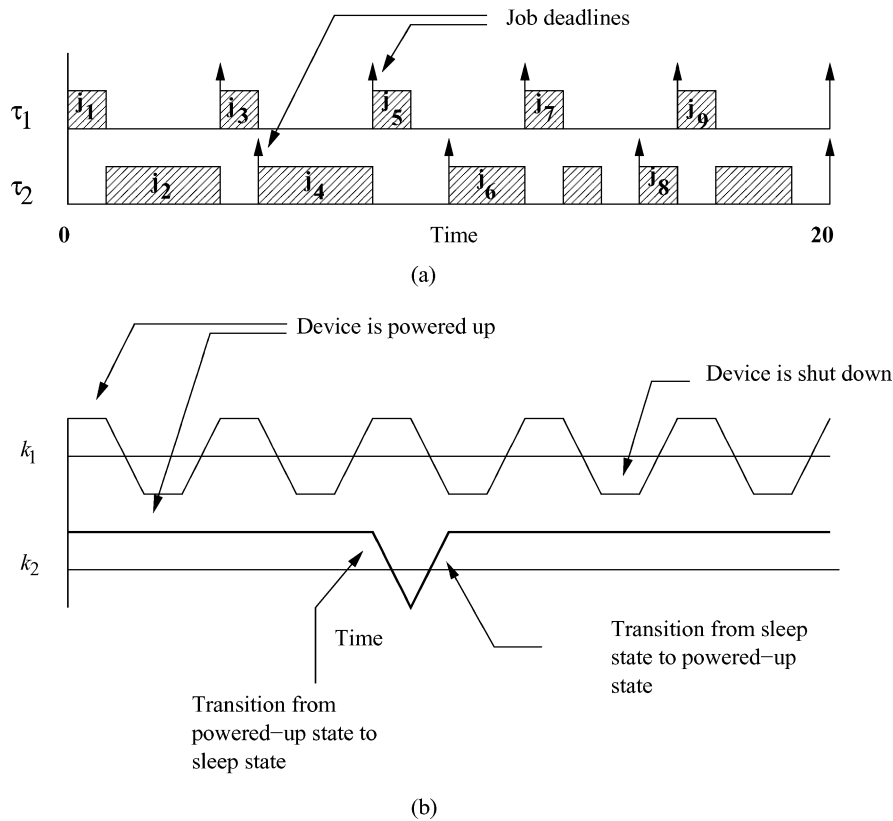


Fig. 11. Task schedule for task set in Table III using RMA.

be consumed if (i) the time for which the devices are powered up is minimized, (ii) the time for which the devices are shutdown is maximized, and (iii) the number of device transitions is minimized (however, if the transition power of a device  $k_i$  is less than its active (operating) power, then energy is minimized by forcing any idle interval for the device to be at least  $2t_{0,i}$ ). In Figure 12(b), no device is powered up when it is not in use. Furthermore, by scheduling jobs of the same task one after the other, the number of device transitions is minimized, resulting in the maximization of device sleep time. Our approach to reducing energy consumption is to find jobs with the maximum device-usage overlap and schedule them one after the other. Indeed, two jobs will have maximum overlap with each other if they are instances of the same task. This is the approach that EDS follows. An alternative approach that MDO takes is to use a precomputed job schedule and swap job slices in an intelligent manner in order to keep devices in a given state for as long a time as possible. It is for this reason that the MDO algorithm generates solutions that are close to optimal, within 4% of the optimal value in all of our experiments.

A side-effect of scheduling jobs of the same task one after the other is the maximization of task activation jitter (see Figure 12). In some real-time control systems, this is an undesirable feature, which reduces the applicability of

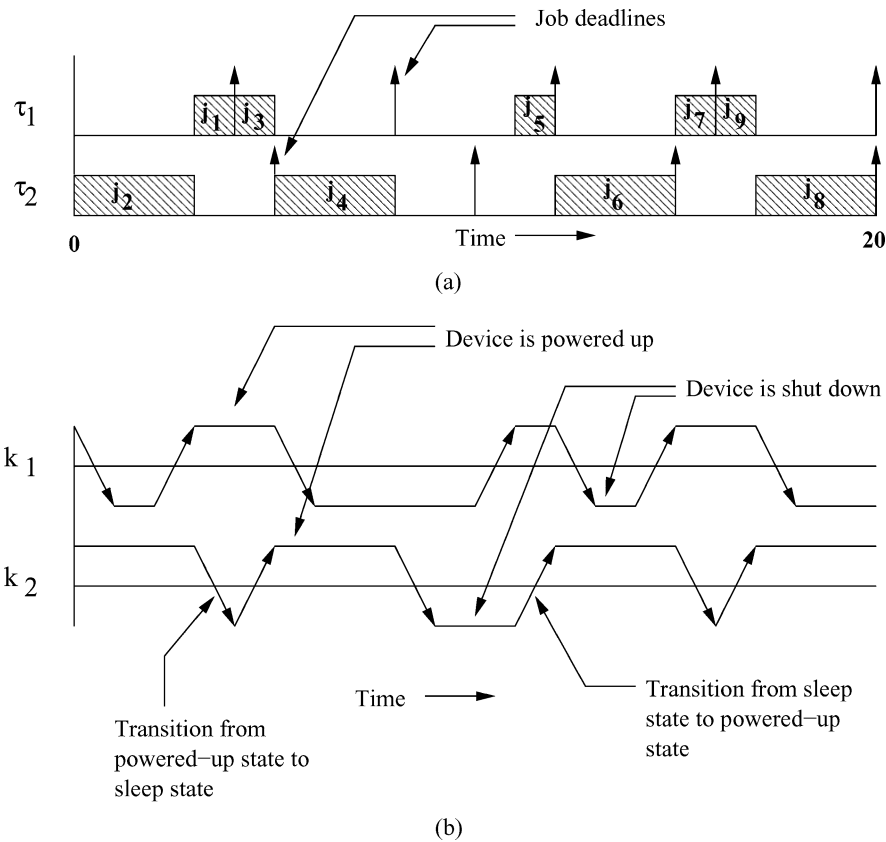


Fig. 12. Optimal task schedule for Table III.

EDS in such systems. However, it is clear that jobs of the same task must be scheduled one after the other in order to minimize device energy. It therefore appears that scheduling devices for minimum energy and minimizing activation jitter are not always compatible goals.

In order to illustrate the effectiveness of the pruning technique, we compare EDS with an exhaustive enumeration method (EE) which generates all possible schedules for a given job set. The rapid growth in the state space with EE is evident from Table VI. We see that the number of vertices generated by EE is enormous, even for a relatively small task set as in Table III. In contrast, EDS requires far less memory. The total number of vertices for EDS is 87% less than that of EE.

By changing the periods of the tasks in Table III, we generated several job sets whose hyperperiods ranged from  $H = 20$  to  $H = 40$  with the number of jobs  $J$  ranging from 9 to 13. For job sets larger than this, EE failed due to lack of computer memory. EE also took prohibitively large amounts of time to run to completion. These experiments were performed on a 500 MHz Sun workstation with 512 MB of RAM and 2 GB of swap space. The results are shown in Table VII.

Table VI. Percentage Memory Savings

Tree Depth $i$	No. of Vertices at Depth $i$		Memory Savings (%)
	EE	EDS	
1	7	7	0
2	4	4	0
3	20	14	30
4	18	12	61
5	76	24	68
6	156	26	83
7	270	18	93
8	648	24	96
9	312	8	97
Total	1512	158	90

Table VII. Comparison of Memory Consumption and Execution Time for EE and EDS

Job Set	No. of Vertices		Execution Time	
	EE	EDS	EE	EDS
$H = 20, J = 9$	1,512	158	<1 s	<1 s
$H = 30, J = 11$	252,931	1,913	2.3 s	<1 s
$H = 35, J = 12$	2,964,093	2,297	28.2 s	4.6 s
$H = 40, J = 13$	23,033,089	4,759	7 m 15 s	35.2 s
$H = 45, J = 14$	–	7,815	–	2 m 29.5 s
$H = 55, J = 16$	–	18,945	–	2 h 24 m 15 s
$H = 60, J = 17$	–	30,191	–	5 h 10 m 23.2 s

– Failed due to insufficient memory.

For job sets with the number of jobs being greater than 17 jobs, the EDS algorithm failed due to insufficient memory. We circumvent this problem by breaking up the vertices generated at level 1 into several separate subproblems. Energy pruning is then performed within and across each subproblem. This is explained in greater detail in the next paragraph.

Let us consider our running example for pruning. Figure 3 illustrates the partial schedule tree after one job has been scheduled. The original EDS algorithm expands each of these nodes in a breadth-first fashion and then performs energy-based pruning across all nodes at the second level, as shown in Figure 4. At deeper levels, the number of nodes increases tremendously, thereby making excessive demands on memory. An enhancement to EDS that addresses the memory consumption issue is to expand only a single level-1 vertex at a time and perform temporal and energy pruning within this single subproblem. The memory requirement is therefore reduced significantly. The minimum-energy schedule derived from solving this single subproblem is then recorded. When the next subproblem is solved, energy pruning is performed both within the current subproblem and across all previously solved subproblems. The solution of a single subproblem results in a minimum-energy schedule with a given level-1 job. This energy value is used as an additional bound that is used for further pruning, even at intermediate depths, in succeeding subproblems. With this enhancement, we were able to solve job sets of up to 26 jobs. Even larger problem instances can be solved by breaking the vertices at lower levels into independent subproblems. Here, however, we restrict ourselves only to level-1 subproblems.

Table VIII. Comparison of EDS with MDO

Job Set	Energy Consumption (J)			$\% \Delta E_1 = \frac{E_{eds} - E_{apu}}{E_{apu}}$	$\% \Delta E_1 = \frac{E_{mdo} - E_{eds}}{E_{eds}}$	Execution Time	
	Enhanced EDS	MDO	All Powered Up			Enhanced EDS	MDO
$H = 20, J = 9$	44.12	45.25	66.60	-33.7%	2.5%	<1 s	<1 s
$H = 30, J = 11$	60.92	62.72	96.9	-37.1%	2.9%	<1 s	<1 s
$H = 35, J = 12$	69.85	72.42	113.05	-38.2%	3.6%	<1 s	<1 s
$H = 40, J = 13$	78.17	80.68	129.20	-39.4%	3.2%	<1 s	<1 s
$H = 45, J = 14$	87.13	90.38	145.35	-40.0%	3.7%	<1 s	<1 s
$H = 55, J = 16$	104.33	106.88	177.65	-41.2%	2.4%	<1 s	<1 s
$H = 60, J = 17$	112.73	115.13	193.80	-41.8%	2.1%	3.98 s	<1 s
$H = 65, J = 18$	121.53	123.38	203.95	-40.4%	1.5%	19.15 s	<1 s
$H = 70, J = 19$	129.93	131.6	226.1	-42.5%	1.2%	58.8 s	<1 s
$H = 80, J = 21$	147.13	148.12	258.4	-43.0%	0.6%	7 m 31 s	<1 s
$H = 85, J = 22$	156.0	156.37	274.0	-43.0%	0.2%	30 m 45 s	<1 s
$H = 90, J = 23$	164.33	164.62	290.7	-43.4%	0.1%	2 h 39 m 35 s	<1 s
$H = 95, J = 24$	170.45	172.87	306.85	-44.5%	1.4%	8 h 9 m 17.3 s	<1 s
$H = 105, J = 26$	186.23	189.37	339.15	-45.0%	1.6%	50 h 0 m 26.6 s	<1 s

$E_{eds}$ : Energy consumption using EDS.

$E_{apu}$ : Energy consumption with devices all powered up.

$E_{mdo}$ : Energy consumption using MDO.

The results for the enhanced EDS algorithm, including a comparison to the MDO heuristic, are shown in Table VIII. For this set of experiments, we used a PC running at 1.4 GHz with 512 MB of RAM.

The MDO algorithm took under 1 s to run for each of the job sets. Furthermore, it results in solutions that differ from the optimal by less than 4%. The energy consumptions of EDS and MDO are also compared to the case where all devices are powered up. The minimum-energy schedules generated by EDS result in energy savings of up to 45% for the larger job sets listed in the table. The growth of the search space (and corresponding increase in execution time) is also evident from the table. An important point to note here is that the use of the energy value of a complete schedule obtained from solving a single subproblem as a bound results in significant pruning at lower levels in the tree. Therefore, the time taken to search the final set of complete schedules for a minimum energy schedule is significantly reduced. This results in faster execution times for the enhanced EDS algorithm.

Finally, we compare EDS with an online device scheduling algorithm for hard real-time systems called LEDES [Swaminathan and Chakrabarty 2003] and a simple timeout-based scheme. In the timeout-based scheme, a device is powered-down if it has not been used for a prespecified interval of time (here, we assume that the timeout interval is 1 unit). However, a timeout-based scheme cannot be used in hard real-time systems since it cannot guarantee that jobs complete execution before their deadlines. Nevertheless, we compare our algorithms with the timeout method to highlight the effectiveness of our algorithms. These results are presented in Table IX. EDS performs better than LEDES and timeout method for all experimental task sets. Moreover, the timeout method resulted in an average of 6.8 missed job deadlines over all our job sets.

Table IX. Comparison of EDS and LEDES [Swaminathan and Chakrabarty 2003]

Job Set	Energy Consumption ( $J$ )			$\frac{\Delta E_3}{E_{ledes}} = \frac{E_{eds} - E_{ledes}}{E_{ledes}}$ (%)
	EDS	LEDES	Timeout	
$H = 20, J = 9$	44.12	59.69	60.21	-26.0
$H = 30, J = 11$	60.92	75.29	85.23	-19.0
$H = 35, J = 12$	69.85	88.4	100.87	-20.0
$H = 40, J = 13$	78.17	102.65	108.76	-23.8
$H = 45, J = 14$	87.13	116.9	130.43	-25.4
$H = 55, J = 16$	104.33	145.4	155.5	-28.2
$H = 60, J = 17$	112.73	159.65	170.43	-29.3
$H = 65, J = 18$	121.53	173.9	192.76	-30.1
$H = 70, J = 19$	129.93	188.15	216.8	-30.9
$H = 80, J = 21$	147.13	216.65	240.98	-31.9
$H = 85, J = 22$	156.0	230.9	252.43	-32.4
$H = 90, J = 23$	164.33	245.15	270.32	-33.0
$H = 95, J = 24$	170.45	259.4	282.53	-34.3
$H = 105, J = 26$	186.23	287.9	315.76	-35.4

$E_{eds}$ : Energy consumption using EDS.

$E_{ledes}$ : Energy consumption using LEDES.

Finally, we discuss the impact of the assumption that  $P_{sd,i} = P_{wu,i}$  and  $t_{sd,i} = t_{wu,i}$  on energy consumption. If the shutdown power  $P_{sd,i}$  is not equal to the wakeup power  $P_{wu,i}$ , and  $t_{sd,i}$  is not equal to  $t_{wu,i}$ , the methods and analyses presented here can still be validated by setting  $P_{0,i} = \max\{P_{sd,i}, P_{wu,i}\}$  and  $t_{0,i} = \max\{t_{sd,i}, t_{wu,i}\}$ . For the case where  $P_{sd,i} < P_{wu,i}$  and  $t_{sd,i} < t_{wu,i}$ , we can expect to save more energy. If  $P_{sd} < P_{wu}$ , devices will not consume as much energy in transitioning between power states, and if  $t_{sd} < t_{wu}$ , devices can be powered-down sooner and can stay in the low-power sleep state for longer periods of time. Hence, without the assumption that  $P_{sd,i} = P_{wu,i}$  and  $t_{sd,i} = t_{wu,i}$ , we can obtain greater savings in energy.

## 8. CONCLUSIONS

Energy consumption is an important design parameter for embedded computing systems that operate under stringent battery lifetime constraints. In many embedded systems, the I/O subsystem is a viable candidate to target for energy reduction. In this paper, we have described an offline low-energy I/O device scheduling algorithm called EDS for hard real-time systems. Our experimental results show that energy savings of over 40% can be obtained using EDS. We have shown that the I/O device scheduling problem is  $\mathcal{NP}$ -complete and that EDS can optimally solve small to moderate-sized problem instances. To solve larger problem instances, we have presented the MDO heuristic that reorders task execution such that devices stay powered down for long periods of time. In all of our experiments, solutions generated by the MDO heuristic consume at most 4% more energy than the energy-optimal EDS solutions.

We next list a few possible extensions to the device scheduling problem and EDS.

- *Joint CPU/I/O-Device Optimization.* Dynamic voltage scaling (DVS) algorithms are currently used for energy minimization in many embedded

systems. Therefore, the effect of task scheduling for minimum device energy on DVS algorithms must be studied in greater detail. An extension to combine I/O-based DPM with DVS appears to be straightforward. The slack that is present in task schedules that are generated using EDS can be utilized with existing DVS algorithms to further reduce energy consumption. However, using DVS for power reduction results in longer execution times for application tasks which, in turn, causes devices to stay powered-up for longer periods of time. Therefore, care must be taken to ensure that the increased energy consumption of I/O devices does not nullify the energy savings using DVS. In this way, two of the major consumers of energy in embedded systems—the CPU and I/O subsystem—can be efficiently targeted for energy reduction.

- *Inclusion of Precedence Constraints.* In this paper, we have assumed an independent task model. However, application tasks often have precedence constraints between them, that is, some tasks (jobs) cannot begin before the completion of other tasks (jobs). Our algorithms can be extended to handle task sets with precedence constraints. With precedence constraints, temporal pruning in the EDS algorithm plays a more significant role in the elimination of redundant schedules since many jobs cannot be scheduled at time-points earlier than the completion of their predecessor jobs. Since the earliest start times and latest finish times of precedence-constrained jobs are restricted to fewer values than with independent jobs, a fewer number of vertices are generated at each level in the schedule tree. Hence EDS is more effective for precedence-constrained tasks. The MDO algorithm can also be extended to address jobs with precedence relations. MDO uses a real-time task scheduling strategy to generate a task schedule and then reorders task slices for reduced energy consumption. It is straightforward to incorporate an additional check within the MDO algorithm to ensure that swapping job slices does not violate precedence constraints between the slices.

#### ACKNOWLEDGMENTS

We thank Prof. Zebo Peng of Linköping University, Sweden, for suggesting the use of the additional energy bound from solving a single subproblem in the enhanced version of EDS. We thank Prof. Petru Eles of Linköping University, Sweden, for suggesting the comparison of EDS to a heuristic method. We also thank Srinivas Chakravarthula of Louisiana State University, Baton Rouge, for his contributions to the development and implementation of the MDO heuristic, as well as implementing extensions to the EDS algorithm. Finally, we thank the anonymous reviewers of this paper for their thoughtful critiques. In particular, we thank Reviewer 3 for providing us with a more realistic I/O device model (Figure 10).

#### REFERENCES

- ABOUGHAZALEH, N., MOSSE, D., CHILDERS, B., AND MELHAM, R. 2001. Toward the placement of power management points in real-time applications. In *Proceedings Workshop on Compilers and Operating Systems for Low Power (COLP)*.

- Advanced Configuration and Power Interface (ACPI). <http://www.teleport.com/~acpi>.
- ALIDINA, M., MONTEIRO, J., DEVADAS, S., GHOSH, A., AND PAPAETHYMIU, M. 1994. Precomputation-based sequential logic optimization for low power. *IEEE Trans. VLSI Syst.* 2, 426–436.
- AMD Am79C874 NetPHY-1LP Low-Power 10/100 Tx/Rx Ethernet Transceiver Technical Datasheet.
- BAKER, T. P. 1991. Stack-based scheduling of real-time processes. *Real-Time Syst. J.* 3, 1, 67–100.
- BENINI, L., BOGLIOLO, A., PALEOLOGO, G. A., AND DE MICHELI, G. 1999. Policy optimization for dynamic power management. *IEEE Trans. Comput.-Aided Des.* 16, 813–833.
- BUTTAZZO, G. C. 1997. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Norwell, MA.
- CHUNG, E.-Y., BENINI, L., AND DE MICHELI, G. 1999. Dynamic power management using adaptive learning tree. In *Proceedings of the International Conference on Computer-Aided Design*. 274–279.
- Fujitsu MHL2300AT Hard Disk Drive Product Manual, <http://www.fcpa.fujitsu.com/products/discontinued-products/index.html#hard-drives>.
- GAREY, M. R. AND JOHNSON, D. S. 1977. Two-processor scheduling with start times and deadlines. *SIAM J. Comput.* 6, 416–426.
- GOLDING, R., BOSH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. 1995. Idleness is not sloth. In *Proceedings of the Usenix Technical Conference on UNIX and Advanced Computing Systems*. 201–212.
- HONG, I., POTKONJAK, M., AND SRIVASTAVA, M. B. 1998. On-line scheduling of hard real-time tasks on variable-voltage processor. In *Proceedings of the International Conference on Computer-Aided Design*. 653–656.
- HWANG, C. AND C.-H. WU, A. 1997. A predictive system shutdown method for energy saving of event-driven computation. In *Proceedings of the International Conference on Computer-Aided Design*. 28–32.
- IRANI, S., SHUKLA, S. AND GUPTA, R. 2002. Competitive analysis of dynamic power management strategies for systems with multiple power states. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*. 117–123.
- ISHIHARA, T. AND YASUURA, H. 1998. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. 197–202.
- KATCHER, D., ARAKAWA, H., AND STROSNIDER, J. 1993. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Software Eng.* 19, 920–934.
- KIM, W., KIM, J., AND MIN, S. L. 2002. A dynamic voltage scaling algorithm for dynamic priority hard real-time systems using slack time analysis. In *Proceedings of the Design Automation and Test Conference in Europe*. 788–795.
- LAWLER, E. L. 1973. Optimal sequencing of a single machine subject to precedence constraints. *J. Management Sci.* 19, 544–546.
- LI, D., CHOU, P., AND BAGERZADEH, N. 2002. Mode selection and mode-dependency modeling for power-aware embedded systems. In *Proceedings of the Asia South Pacific Design Automation Conference*. 697–704.
- LIU, C. L., AND LAYLAND, J. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM.* 20, 1, 46–61.
- LIU, J. W. S. 2000. *Real-time Systems*, Prentice-Hall, Upper Saddle River, NJ.
- LIU, J., CHOU, P. H., AND BAGERZADEH, N. 2002. Communication speed selection for embedded systems with networked voltage-scalable processors. In *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*. 169–174.
- LU, Y.-H., BENINI, L., AND DE MICHELI, G. 2000. Low-power task scheduling for multiple devices. In *Proceedings of the International Workshop on Hardware/Software Codesign*. 39–43.
- LU, Y.-H., BENINI, L., AND DE MICHELI, G. 2000. Operating system directed power reduction. In *Proceedings of the International Conference on Low-Power Electronics and Design*. 37–42.
- LUO, J. AND JHA, N. K. 2001. Battery-aware static scheduling for distributed real-time embedded systems. In *Proceedings of the Design Automation Conference*. 444–449.
- LUO, J. AND JHA, N. 2002. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proceedings of the International Conference on VLSI Design*. 719–726.

- QUAN, G. AND HU, X. 2001. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the Design Automation Conference*. 828–833.
- QUAN, G. AND HU, X. 2002. Minimum-energy fixed-priority scheduling for variable-voltage processor. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*. 782–788.
- SHIN, Y. AND CHOI, K. 1999. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the Design Automation Conference*. 134–139.
- SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.* 39, 1175–1185.
- SHIN, Y., CHOI, K., AND SAKURAI, T. 2000. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design*. 365–368.
- SIMUNIC, T., BENINI, L., GLYNN, P., AND DE MICHELI, G. 2001. Event driven power management. *IEEE Trans. Comput.-Aided Des.* 20, 840–857.
- SWAMINATHAN, V. AND CHAKRABARTY, K. 2003. Energy-conscious, deterministic I/O device scheduling in hard real-time systems. *IEEE Transaction on Computer-Aided Design Integrated Circuits and Systems* 22 (July), 847–858.
- TMS320C6411 Power Consumption Summary. Available on-line at [www.s.ti.com/sc/techlit/spra373](http://www.s.ti.com/sc/techlit/spra373).
- XU, J. AND PARNAS, D. L. 2000. Priority scheduling vs. pre-run-time scheduling. *Int. J. Time-Critical Comput. Syst.* 18, 7–23.
- YAO, F., DEMERS, A., AND SHENKER, S. 1995. A scheduling model for reduced CPU energy. In *Proceedings of the IEEE Annual Foundations of Computer Science*. 374–382.
- ZHANG, Y., HU, X., AND CHEN, D. 2002. Task scheduling and voltage selection for energy minimization. In *Proceedings of the Design Automation Conference*. 183–188.

Received July 2002; revised May 2003; accepted October 2003