Energy-Constrained Performance Optimizations For Real-Time Operating Systems

Tarek A. AlEnawy and Hakan Aydin
Computer Science Department
George Mason University
Fairfax, VA 22030
{thassan1,aydin}@cs.gmu.edu

ABSTRACT

In energy-constrained settings, most real-time operating systems take the approach of minimizing the energy consumption while meeting all the task deadlines. However, it is possible that the available *energy budget* is not sufficient to meet all deadlines and some deadlines will inevitably have to be missed. In this paper, we present a framework through which the operating system can select jobs for execution in order to achieve two alternative performance objectives: 1) maximizing the number of deadlines met , and 2) maximizing the total reward (utility) of jobs that meet their deadlines during the operation. We present an optimal algorithm that achieves the first objective. We prove that achieving the latter objective is NP-Hard and propose some fast heuristics for this problem. We evaluate the performance of the heuristics through simulation studies.

Keywords

Real-time scheduling, real-time operating systems, energy management, power-aware scheduling, power-aware systems.

1. INTRODUCTION

With the proliferation of wireless, portable and embedded devices, power-aware systems have moved to the forefront of Computer and Electrical Engineering research in recent years. A prominent energy management technique for real-time/embedded systems is based on **Dynamic Voltage Scaling (DVS)**. With DVS, it is possible to obtain significant savings in CPU energy consumption by simultaneously reducing the supply voltage and the clock frequency (CPU speed) at the expense of increased latency. Thus, in recent years, the *real-time* DVS (RT-DVS) research has been dominated by numerous papers that address the problem of meeting task deadlines while minimizing the CPU energy consumption under various task/system models and off-line/on-line scheduling techniques [1,2,3,6,9,10,11,12,14,15,16].

However, an almost invariable assumption of this line of research is that the "hard" performance constraint for the operating system is still to meet *all* the task deadlines, and that minimizing the energy consumption subject to hard real-time constraints is a highly desirable, yet "secondary", performance objective. In this paper, we look at the problem from the other perspective: we consider systems with "hard" *energy constraints*. That is, we assume that we are given a limited/scarce *energy budget* (E_{budget}) and a targeted system run time *(mission time)* during which the system *must* remain *functional*. Such a situation can arise when the computing device depends on the battery power supply during the operation, example scenarios include embedded control applications as well as military, space, and disaster recovery missions where a predictable real-time system behavior is necessary but battery re-charging during the mission is not possible (or feasible) at all. In fact, this is exactly one of the main arguments of [9], where the authors focus on the case of Mars Rover system to argue about the need to promote the 'hard' energy constraint to a first-class design consideration.

Having the knowledge of the workload and the energy limitations, the operating system is often the entity that is in best position to plan for delivering maximum performance within the available energy budget. Clearly, the operating system can still make use of well-known power-aware real-time scheduling techniques (such as DVS or predictive shutdown) and compute the minimum energy (E_{bound}) required to meet *all* the deadlines during the mission time. However, in scarce-energy settings where $E_{budget} < E_{bound}$, we have a serious problem: *some deadlines will inevitably have to be missed*. As we show in section 3 through a simple example, without any provisions, the system may run out of energy in the middle of the operation with no control whatsoever on the specific deadlines missed. In contrast, in our approach the operating system *selectively* labels task instances as 'skipped' or 'selected', prior to task execution, in order to stay within the energy budget while maximizing the performance. At run-time, only the task instances (jobs) that have been previously selected by the operating system are dispatched.

In this research effort, we propose three performance metrics to determine the task instances (jobs) to be executed during the system operation, namely:

- maximize the number of met deadlines,
- maximize the number of met deadlines while providing a minimum performance guarantee for each task, and
- maximize the reward (utility) of the system by giving preference to more important/critical tasks.

Each of the metrics above may be more appropriate for different systems and "mission" objectives. We underline that our framework is applicable to systems with or without DVS capability. In systems where the CPU does not have the DVS capability, our framework provides a way to maximize the system performance in terms of timing constraints and criticality, while staying within the energy budget. On the other hand, *DVS capability by itself does not guarantee that all the timing constraints will be met while staying within a given fixed energy budget.* As an example, consider a task set with total utilization 0.9. A well-known result from RT-DVS theory states that the minimum CPU speed that minimizes the CPU energy consumption and still meets all the deadlines is equal to 90% of maximum CPU speed [2] when the scheduling policy is Earliest-Deadline-First (EDF). Now suppose that this system has to remain functional for one hour, requiring K units of energy with speed 0.9. If it is not possible to re-charge the battery during the operation and if the battery in use can deliver at most K/2 energy units, then clearly the operating system has to adopt additional energy saving strategies (besides using DVS) to yield the best performance possible in these circumstances.

We note that a recent study [13] has also addressed some aspects of real-time scheduling issues with a fixed energy budget. However, the model in that paper is based on tasks having identical periods, as opposed to our more common and general case of possibly different periods. In addition, we also explore the more common performance metric of minimizing the number of missed deadlines in addition to maximizing the system reward.

The rest of this paper is organized as follows. In section 2 we present the system model and our assumptions. In section 3 we define the problem formally and explain our performance metrics for energy-constrained settings. Section 4 discusses the proposed solutions. In section 5, we present our simulation results. In section 6 we conclude the paper.

2. SYSTEM MODEL

2.1 Task model

We consider a set of n periodic tasks $T=\{T_1, T_2, \ldots, T_n\}$ that are to be scheduled on a single-processor system. Each task T_i is characterized by its worst-case execution time C_i, relative deadline D_i, and period P_i which is assumed to be equal to D_i. The utilization U_i of task T_i is defined as C_i/P_i. We denote the j^{th} instance of task T_i by T_{ij}, which is also referred to as the job T_{ij}. The hyperperiod P of the task set is defined as the least-common multiple of all task periods. All the tasks are assumed to be independent and simultaneously ready at time $t=0$. Our framework assumes preemptive scheduling. Finally, we assume that the preemption overhead is negligibly small and, if needed, it can be incorporated into C_i.

2.2 Power and energy consumption model

Throughout the paper we distinguish between two operational modes: **normal (execution) mode** and **low-power (stand-by) mode**. We say that the system is in normal (execution) mode when the system is executing a job. Otherwise, the system switches to low-power (stand-by) mode when the CPU is idle. The power consumptions of the normal mode and low-power modes are assumed to be g_e and g_{low} watts (joules per second), respectively[1]. In an interval $[t_1,t_2]$, the total energy consumption of the CPU is the sum of the execution mode energy E_e and the low-power mode energy E_{low}:

$$E(t_1,t_2) = E_e + E_{low} = g_e t_e + g_{low} t_{low} \quad (1)$$

where t_e is the total time during which the system runs in the execution mode and t_{low} is the total time during which the system runs in the low-power mode in the specified interval.

In our settings, the system has a limited CPU energy budget E_{budget}, and moreover, it has to remain operational in the interval [0,X]; that is, the total CPU energy consumption should not exceed a pre-determined threshold E_{budget} for X time units. X is also called the *mission time* throughout the paper. In other words, the system is subject to a *hard energy constraint*, which is a measure of available energy resources. The *minimum* energy amount necessary to complete *all* the task instances in a timely manner (i.e. before their deadlines) is denoted by E_{bound}. Throughout the paper we will assume that the task set is schedulable, that is all deadlines would be met according to a policy of designer's choice (e.g. EDF or RMS), if the available energy were greater than or equal to E_{bound} (i.e., if $E_{budget} \geq E_{bound}$). We are making this assumption because our goal is not to study real-time schedulability problem in the absence of any energy constraints, an intensively studied problem [8]. **Instead, our focus will be on the systems with hard energy constraints where the available energy is not sufficient to execute all the workload; that is, we consider systems where $E_{budget} < E_{bound}$.**

[1] The energy overhead due to switching between the two modes can be incorporated into g_{low}, if necessary. Similarly, the time overhead due to switching can be incorporated into C_i.

Applicability to systems with multiple speed levels: In DVS-enabled systems with multiple frequency and power levels, it is realistic to set g_e to the power consumption level of the *lowest* CPU speed that would be able to meet *all* the deadlines with the given scheduling policy. E_{bound} would be then equal to the energy consumption corresponding to this speed/power level during the entire mission, and our objective is again to provide a task selection framework for systems with scarce energy budget (where $E_{budget} < E_{bound}$).

3. MAXIMIZING SYSTEM REWARD IN ENERGY-CONSTRAINED SETTINGS

As explained in Introduction, for some applications it may be vital that the system remain functional and does not run out of energy during the mission time interval [0,X]. Moreover, if the available energy is scarce, blindly trying to execute all the jobs may result in a rather poor performance when we consider timing constraints, as the following example illustrates.

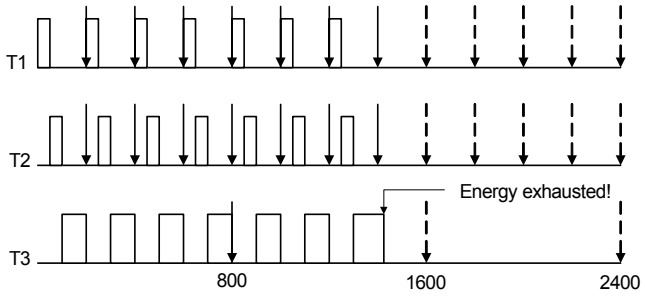

Figure 1: Naïve schedule generated by EDF

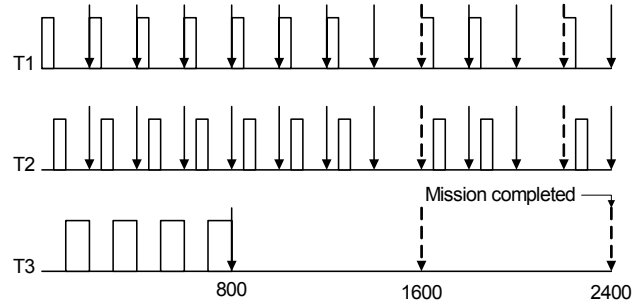

Figure 2: Improved schedule

Example 1

Consider a task set that consists of three tasks T_1 , T_2 , and T_3 with the following parameters: $U_1 = U_2 = 0.25$, $P_1 = P_2 = 200$, $C_1 = C_2 = 50$; $U_3 = 0.5$, $P_3 = 800$, $C_3 = 400$. Let $E_{budget} = 1425$, $X = 2400$, $g_e = 1$, $g_{low} = 0.025$. Based on these parameters E_{bound} can be computed as *2400*. The task set is schedulable using Earliest-Deadline-First (EDF) since the total utilization does not exceed 1.00 [7], if we do not take into account the hard energy constraint. Figure 1 shows the schedule produced by EDF (solid arrows denote the deadlines met, whereas the dashed arrows indicate the missed deadlines). Observe that since the hard energy constraint is not taken into account, *the system runs out of energy* at t=1425, just before T_{32} completes. Due to inefficient energy management the system energy budget is exhausted just before 60% of the mission time X has elapsed, about 23% of E_{budget} is wasted by executing T_{32} which never completes, and only 15 deadlines (about 56 % of the total) are met. Figure 2 shows an "improved" schedule (for the same problem) that intelligently uses the available energy to yield better performance. In this case, the system remains operational throughout the entire mission time and we are able to meet 21 deadlines (about 78% of the total, with 40% improvement compared to the naive schedule of Figure 1) without exceeding E_{budget}. Note also that all the jobs completed in the naive schedule are still completed in the improved schedule.

As this example illustrates, if the energy budget of the system cannot sustain the operation during the entire mission, then the aim must be to provide maximum predictability/utility for applications imposing timing constraints. We consider two cases of maximizing system performance in energy-constrained settings as follows.

3.1 Maximizing the number of deadlines met during mission time

One natural performance objective might be to maximize the total number of deadlines met, within the limited energy budget, while sustaining the system operation during [0,X]. This is particularly suitable for applications where occasional deadline misses may be acceptable, such as real-time communications, tracking and multimedia [6,8]. We denote this objective by **O1**. We examine two variations of this objective.

- **No minimum task–level requirements:**

In the absence of any additional constraints, we can simply try to maximize the total number of deadlines met across all tasks during mission time [0,X].

- **Minimum task deadline meet ratio:**

The operating system may attempt to ensure that a minimum percentage of each task's deadlines are met during mission time [0,X], thus providing a minimum performance guarantee for each task. This can be done by requiring that each task T_i meets its **minimum deadline meet ratio** M_i such that $n_i / N_i \geq M_i$, where n_i is the number of instances of task T_i that completed before their deadlines during [0,X] and $N_i = \lfloor X / P_i \rfloor$ is the number of all the instances of the same task whose deadlines lie within the same interval. Note that the aim is still maximizing the

number of deadlines met; only we are adding the minimum performance constraints. In fact, the "improved" schedule of Figure 2 maximized the total number of deadlines met for the task set discussed in Example 1, subject to the constraint $M_i = 30\%$ for each task.

3.2 Maximizing the total reward (utility) of the system

Objective **O1** given in section 3.1 does not distinguish between the **importance** of different tasks. However, some tasks may be deemed more important than others in the presence of limited energy budget. For example, real-time communication tasks may be more important than display update tasks. To this aim, we can associate a *weight (reward)* w_i with each task T_i. If an instance T_{ij} is completed before its deadline, then the total system reward is increased by w_i units (Note that all the jobs of T_i have the same reward). Otherwise, if an instance is skipped altogether or misses its deadline then the total reward remains the same. Thus, we have the objective **O2: selectively execute jobs so as to maximize the total system reward.**

The two variations discussed in **O1**, either guaranteeing a minimum deadline meet ratio for each task or the absence thereof, can still be applied to this case. Also observe that, Objective **O1** is a special case of Objective **O2** where all the tasks have the same weight $w_i = w$.

3.3 Our approach

In energy-constrained settings where $E_{budget} < E_{bound}$, the task instances to be executed must be carefully selected in order to *make best use of available energy*. Clearly, objectives O1 and O2 will provide the guidelines depending on the performance metric under consideration. To achieve this aim, the operating system can undertake a job selection phase in which each task instance (job) is labeled as **skipped** or **selected**. Clearly the operating system must consider the parameters of the task set, as well as available energy and operation duration information when making decisions in light of the performance metric of choice. Our solutions are general in the sense that they work with any scheduling policy α (e.g. RMS or EDF), providing maximum flexibility for the operating system[2].

Let Π be the set of all task instances (jobs) that must complete by (i.e. with deadlines on or before) the mission time X. We associate with each job T_{ij} a label S_{ij} where $S_{ij} = 0$ indicates that the job is skipped and $S_{ij} = 1$ indicates that it is selected for execution. **At run-time, only jobs whose labels are set to "selected" are dispatched**. Let $\Pi' = \{ T_{ij} \mid S_{ij} = 1 \}$ denote the set of selected instances. Thus the problem becomes, **choose $\Pi' \subseteq \Pi$ to achieve the performance objective (O1 or O2) while making sure that the energy budget is not exceeded**. Note that the energy consumed by the selected task instances plus the energy consumed in the stand-by mode should not exceed E_{budget}. That is:

$$\sum_{T_{ij} \in \Pi'} g_e C_i + \left(X - \sum_{T_{ij} \in \Pi'} C_i \right) g_{low} \le E_{budget} \quad (2)$$

Or equivalently,

$$g_{low} X + (g_e - g_{low}) \sum_{T_{ij} \in \Pi'} C_i \le E_{budget} \quad (3)$$

Note that the quantity $(g_e - g_{low})C_i$ is the excess energy consumed by each instance of task T_i in normal mode beyond the idle energy. Once selected, the set Π' can be scheduled by the algorithm α and the feasibility of *selected instances* is guaranteed since the superset Π is assumed to be initially schedulable under Algorithm α when the energy budget is not taken into account.

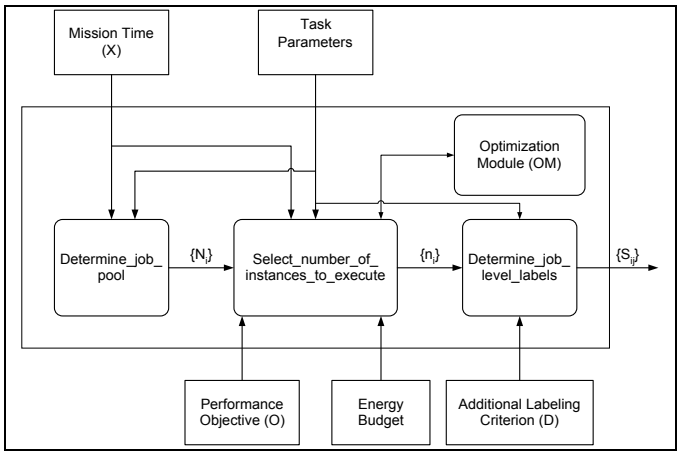

Figure 3. The generic job selection framework for energy-constrained real-time operating systems

The generic job selection framework that we propose for energy-constrained real-time operating systems consists of three main modules (Figure 3):

- ***Determine_job_pool***$(T, X, \{N_i\})$:

For each task T_i, calculates the total number of instances N_i whose deadlines are smaller than or equal to X. Note that N_i is simply equal to $\lfloor X / P_i \rfloor$, since this quantity gives the number of jobs belonging to T_i that must complete by X.

- ***Select_number_of_instances_to_execute***$(T, O, X, \{N_i\}, \{n_i\}, E_{budget})$:

Uses an auxiliary optimization module *OM*, which can be an optimal algorithm or a heuristic, to decide on the number of instances n_i to be executed for each task T_i

[2] Recall that we assume the feasibility of the schedule is guaranteed if all the jobs run in normal-mode and scheduled by Algorithm α when there are no constraints on energy budget.

during [0,X] to achieve the specific objective *O* under consideration, without exceeding the energy budget. This is the only module that needs to be modified if the performance objective changes.

- ***Determine_job_labels***($\{n_i\}$, *D*, $\{S_{ij}\}$):

Given the number of instances n_i to be executed for each task T_i , determine specific instances to be actually scheduled using an additional labeling criterion *D*. In other words, determine the label S_{ij} (skipped or executed) for each job. These labels will be used in conjunction with the original scheduling algorithm α at run-time, to maximize the system performance while staying within the system energy budget.

The module *Select_number_of_instances_to_execute* is highly dependent on the specific performance metric (objective) under consideration; we will give the specific pseudo-code for this module corresponding to different metrics when discussing the solutions for optimization problems.

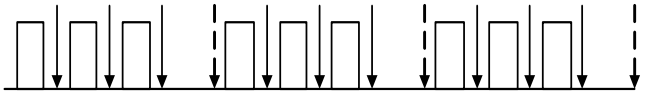

Figure 4. 'Balanced' distribution of skipped instances

Note that determining the number of instances to be executed during the mission time, namely n_i, for each task is only a part of the problem. We need to determine *which specific instances* will be labeled as selected or skipped. As an example, suppose that a given task T_i has 200 instances during the mission, but only 150 have been selected for execution. Which 150 will be selected? The first 150, the last 150, or *any* 150? The labeling criterion *D* essentially provides the algorithm to be used for making this decision. In the simplest case, one can adopt trivial techniques such as selecting the first n_i instances. However, many applications including multimedia and real-time communication tasks, would still deliver a performance of acceptable quality if the deadline misses are relatively few and distantly spaced [5,6]. Thus, in the above example, one can select three task instances out of each four consecutive instances for execution to obtain a 'balanced' distribution of 50 skipped instances during mission time. Part of this schedule is shown in Figure 4 (solid arrows and dashed arrows denote the met and missed deadlines, respectively).

We note that the approach of meeting m_i deadlines of each k_i consecutive instances was previously adopted in overloaded (but not energy-constrained) real-time scheduling approaches [5,6]. We believe that this is a noteworthy parallelism between overloaded and energy-constrained systems.

4. SOLUTIONS TO ENERGY CONSTRAINED PERFORMANCE OPTIMIZATION PROBLEMS

In this section we discuss the solutions to maximize the performance objectives stated earlier, **O1** and **O2**.

4.1 Objective O1: maximizing the number of deadlines met during the mission time

To maximize **O1** while staying within E_{budget} , then the problem can be solved efficiently. Consider the policy that first orders tasks according to execution time per instance C_i and then chooses for execution the instances of tasks with smallest execution times until E_{budget} is exhausted. We refer to this policy as **Favor Shortest Job (FSJ)**[3].

Optimization Module for FSJ

Input: T, E_budget, X, g_e, g_{low}
Output: $\{n_i\}$

1. Order tasks according to their execution time (T_1 represents the task with the smallest execution time);
2. In_progress = **true**;
3. Available = E_budget – X.g_{low};
4. i = 1;
5. **while** (In_progress==**true**) and (i ≤ n)
6. **begin**
7. z = (g_e - g_{low}). C_i;
8. **if** z ≤ Available **then**
9. **begin**
10. $N_i = \lfloor X / P_i \rfloor$;
11. n_i = min{ N_i , $\lfloor$ Available / z $\rfloor$ };
12. Available = Available - n_i.z;
13. **end**
14. **else** In_progress = **false**;
15. i = i + 1;
16. **end**

Figure 5. Pseudo-code for FSJ optimization module

Consider the *select_number_of_instances_to_execute* module which uses the optimization module FSJ_OP corresponding to FSJ (Figure 5). FSJ_OP starts off by setting aside enough energy to sustain the system in low-power mode in [0,X]. All the instances of a given task T_i have the same execution time C_i , thus we can start with the task T_1 having the smallest execution time C_1. Then,

[3] Note that this scheme is different from the well-known scheduling algorithm Shortest Job First (SJF), because FSJ selects task instances for execution in off-line fashion, and then these instances are scheduled using the algorithm α chosen by the operating system.

FSJ_OP determines the *maximum number of instances* n_1 of task T_1 that can be executed within the current budget E_{budget}. It then updates E_{budget} by deducting the excess energy consumed by the newly selected jobs. This process is repeated for tasks with next smallest C_i until either E_{budget} is exhausted or all task instances are selected.

THEOREM 1. *FSJ is optimal for the problem of maximizing the number of deadlines met.*

Proof: We will proceed by a proof by contradiction. Assume that FSJ is not optimal. This implies that there is an instance I of the problem of maximizing the number of deadlines met, for which FSJ selects m jobs for execution, while there exists another algorithm A that succeeds in meeting $M > m$ deadlines in I without exceeding E_{budget}. Let σ_{FSJ} be the set of jobs selected by FSJ for execution and σ_A be the set of jobs whose deadlines are met by algorithm A. Without loss of generality, assume both σ_{FSJ} and σ_A are ordered according to the execution times of jobs in non-decreasing fashion.

Let E_{low} be the energy amount required to sustain the system in stand-by mode during the mission time X; that is:

$$E_{low} = X \,.\, g_{low} \tag{4}$$

Also, let $E(H)$ be the excess energy required to execute all the jobs of a given set H, beyond E_{low} :

$$E(H) = (g_e - g_{low}) \sum_{i \in H} C_i \tag{5}$$

Let us denote the first m jobs of σ_A (i.e. those having the smallest m execution times) by $\sigma_{A,m}$. Note that the k^{th} job in σ_{FSJ} has an execution time that *does not exceed* that of the k^{th} job in $\sigma_{A,m}$, due to the very characteristic nature of the set generated by Favor Shortest Jobs (FSJ). This clearly translates to the energy requirements of the jobs in the two sets when compared pairwise. In other words, the energy required to execute $\sigma_{A,m}$ cannot be smaller than that required to execute σ_{FSJ}: $E(\sigma_{A,m}) \geq E(\sigma_{FSJ})$. Note that the $(m+1)^{\text{th}}$ job J_y in σ_A (such a job should exist if $M > m$) must have a larger execution time, and hence larger energy requirement, than the job J_x having $(m+1)^{th}$ smallest execution time in the job set. Since FSJ stopped after m smallest jobs, we must have:

$$E_{low} + E(\sigma_{FSJ}) + E(\{J_x\}) > E_{budget} \tag{6}$$

On the other hand, the total energy required to execute the jobs in σ_A is definitely at least as large as the energy required to execute only the first m+1 jobs of σ_A, that is:

$$E_{low} + E(\sigma_A) \geq E_{low} + E(\sigma_{A,m}) + E(\{J_y\}) \tag{7}$$

But the right-hand side of (7) cannot be smaller than the left-hand side of (6) (see the discussion above). Thus, we have:

$$E_{low} + E(\sigma_{A,m}) + E(\{J_y\}) \geq E_{low} + E(\sigma_{FSJ}) + E(\{J_x\}) \tag{8}$$

Hence:

$$E_{low} + E(\sigma_A) > E_{budget} \tag{9}$$

This implies that the algorithm A has *exceeded* the energy budget while meeting $M > m$ deadlines; clearly a contradiction.

Hence, FSJ must be optimal. □

Note that FSJ_OP needs to sort tasks by their execution times C_i which can be achieved in time $O(n \log n)$. Once sorted, FSJ_OP selects for execution task instances with smaller execution times which can be achieved in time $O(n)$. The total complexity of FSJ_OP is then $O(n \log n)$.

In the case where we have *minimum deadline meet ratio* M_i for each task, we must first reserve enough energy to meet the minimum requirements of each task. After this reservation, we update our energy budget. If we have excess energy beyond this point, then we can still use the same technique to maximize the total number of deadlines met as described above. The optimality of this approach can be proved in a similar way as in Theorem 1 but is omitted due to space limitations. Observe that for any "hard" real-time task that must meet its deadline at every instance, this ratio can be simply set to 100%.

4.2. Objective O2: maximizing the total reward (utility) of the system

When we associate a weight (reward) w_i with each task instance, a natural objective may be maximizing the total reward of the jobs which are executed (which meet their deadlines). Let us refer to the problem of maximizing total system reward over [0,X] as *REWARD*. Formally, *REWARD* is defined as to determine $\Pi' \subseteq \Pi$ so as to:

$$\text{maximize} \sum_{T_{ij} \in \Pi'} w_i \tag{10}$$

$$\text{subject to } (g_e - g_{low}) \sum_{T_{ij} \in \Pi'} C_i + X g_{low} \leq E_{budget} \tag{11}$$

That is, our aim is to maximize the total reward of the executed jobs subject to the condition that the system energy consumption in normal-mode (needed to execute the selected jobs) plus any additional idle energy needed during the mission time does not exceed the system energy budget. Blindly selecting task instances using FSJ, as in O1, may yield a low reward value, as the following example illustrates.

Example 2

Consider the schedule generated by FSJ when the individual deadline meet ratios (30%) are taken into account, given in Figure 2. Assume that T_3 represents a critical task for the system, and that we have the following task weights: $w_1 = w_2 = 1$ and $w_3 = 20$. In this case the schedule of Figure 2 would yield a total reward of 40.

However, by scheduling two instances of T_3 and six instances of each of T_1 and T_2 it is possible to obtain a total reward of 52 (a 30% improvement over FSJ) without exceeding E_{budget} .

Unfortunately, the problem of maximizing the total system reward with hard energy constraints turns out to be intractable in the general case:

THEOREM 2. *REWARD is NP-Hard.*

Proof: We will prove the theorem by showing that it is at least as hard as the KNAPSACK problem, which is NP-Hard [4].

KNAPSACK: Given a set of items $S = \{s_1, s_2, \ldots, s_n\}$ where each item s_i has an integer value v_i and an integer cost z_i, an integer V representing the total value we aim at, and an integer Z representing the total cost within which we must stay, is there a subset $S' \subseteq S$ such that:

$$\sum_{s_i \in S'} v_i \geq V \text{ and } \sum_{s_i \in S'} z_i \leq Z \, ? \tag{12}$$

We will consider a special ("easy") instance of the REWARD problem, called S-REWARD and prove that KNAPSACK can be reduced to it. S-REWARD is defined in the same way as REWARD, however all tasks are restricted to have the same period P. Furthermore, mission time X is set to P; i.e. $P_1 = P_2 = \ldots = P_n = P = X$. Solving REWARD in these limited settings is equivalent to solving S-REWARD which is to select $\Pi' \subseteq \Pi$ so as to:

$$\text{maximize } \sum_{T_{ij} \in \Pi'} w_i \tag{13}$$

$$\text{subject to } (g_e - g_{low}) \sum_{T_{ij} \in \Pi'} C_i + X g_{low} \leq E_{budget} \tag{14}$$

where Π' is the set of jobs chosen for execution.

Given an instance of the KNAPSACK problem with value limit V, cost limit Z, set of items $S = \{s_1, s_2, \ldots, s_n\}$, an integer value v_i and an integer cost z_i for each item; we construct the corresponding instance of S-REWARD problem as follows: We have n tasks $T_1, T_2, \ldots, T_n$ in Π. Further, for $1 \leq i \leq n$, we set $w_i = v_i$, $C_i = z_i$. We further set $g_e = 1$, $g_{low} = 0$, and $E_{budget} = Z$. The special instance of S-REWARD thus becomes:

$$\text{maximize } \sum_{T_{ij} \in \Pi'} w_i \tag{15}$$

$$\text{subject to } \sum_{T_{ij} \in \Pi'} C_i \leq Z \tag{16}$$

This transformation can be clearly performed in polynomial time. Now, suppose that there is a polynomial-time solution to S-REWARD. Given an instance of KNAPSACK, apply the transformation given above, solve the corresponding S-REWARD instance and compute the quantity $\sum_{T_{ij} \in \Pi'} w_i$.

Now, if $\sum_{T_{ij} \in \Pi'} w_i \geq V$, then clearly $\sum_{s_i \in S'} v_i = \sum_{T_{ij} \in \Pi'} w_i \geq V$ and S' is a solution to the KNAPSACK problem, since $\sum_{T_{ij} \in \Pi'} C_i = \sum_{s_i \in S'} z_i \leq Z$ is satisfied by construction. Otherwise, if $\sum_{T_{ij} \in \Pi'} w_i < V$, then clearly there is no solution to the KNAPSACK problem; since by construction, S-REWARD selects the subset Π' such that $\sum_{T_{ij} \in \Pi'} w_i = \sum_{s_i \in S'} v_i$ is maximized, subject to $\sum_{T_{ij} \in \Pi'} C_i = \sum_{s_i \in S'} z_i \leq Z$.

This shows that the problem S-REWARD, which is a special case of REWARD, is at least as hard as the KNAPSACK problem, and, hence, it is NP-Hard. Thus, REWARD must be NP-Hard as well. □

5. EXPERIMENTAL RESULTS

Since the problem of maximizing system reward subject to a fixed energy budget is NP-Hard, we explored several fast heuristics. The sub-optimal heuristics we explored are all fast (heuristic-based optimization module has a time complexity of *O(n log n)*), but also greedy in the sense that each heuristic gives preference to tasks that have higher values of a specific (combination of) task parameter(s) such as execution time, period, and weight. The optimization module of each heuristic is essentially the same as that of FSJ (Figure 5), the only difference is that, in Step 1, tasks are first ordered according to the specific task parameter (combination) instead of the execution time. We limit our discussion to the five heuristics that consistently yielded the best results, in addition to FSJ.

- **LRD (Larger Reward Density):** This heuristic is an improved version of FSJ in the sense that it considers both the reward and execution time (thus, the energy cost) of tasks. Specifically, LRD favors jobs with larger w_i / C_i value (i.e. jobs with larger reward and shorter execution time).
- **LRSP (Larger Reward Smaller Period):** favors tasks with larger w_i / P_i value, relying on the fact that choosing tasks with large reward and *large* number of instances during mission time may be more beneficial for the system.
- **LRDSP (Larger Reward Density Smaller Period):** This heuristic favors tasks with larger w_i / $P_i C_i$ values, combining the ideas behind LRD and LRSP techniques.
- **LRSU (Larger Reward Smaller Utilization):** gives preference to tasks with larger w_i / U_i values, considering that tasks that yield larger rewards while using a small *fraction* of CPU time should improve the overall system utility.

- **LR (Larger Reward):** gives preference to tasks with larger rewards w_i. Note that this scheme is not an optimal one, because it does not consider the execution time of (hence, the energy required to execute) tasks.
- **FSJ (Favor Shortest Jobs):** This technique, whose optimality is proven for the case where all task rewards are equal in Section 4, favors jobs with smaller execution times regardless of the reward accrued by their execution.

We ran simulation experiments to evaluate the performance of the heuristics discussed above. We measured the total system reward R as a function of 4 parameters:

- *Total system utilization*: $U = \sum C_i / P_i$.
- *System energy budget*: E_{budget} as a percentage of the total energy required to meet all deadlines E_{bound}. Note that as this percentage decreases, the system becomes more energy-constrained and it becomes more crucial to judiciously manage the available energy to deliver the best possible performance.
- *Task weight ratio*: $WR = max\{w_i\} / min\{w_i\}$. This ratio represents the relative variance of task rewards; as WR increases the weights of tasks in the system exhibit more variance.
- *Ratio of normal-mode power to stand-by mode power* (*gain ratio* $= g_e / g_{low}$): By changing this ratio, it is possible to model various system settings where the energy savings obtained by switching to low-power mode vary over different ranges. Clearly, the larger this ratio, the more significant the energy savings in low-power mode. Note that a given CPU may have multiple modes corresponding to inactivity periods, such as *sleep*, *doze*, and *shutdown*. Each of these levels may have different power characteristics and different overheads associated with switches to/from normal mode. Analyzing the system performance for different possible gain ratios may lead to better trade-offs.

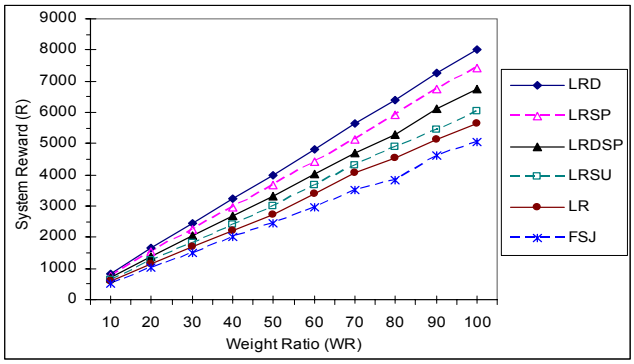

Figure 6. System reward as a function of task weight ratio for U=0.7, E_{budget}=30%, and gain ratio =100

We generated 100,000 generic task sets and then for each task set, we changed the above parameters over the following ranges: U ranged from 0.1 to 1.0 in steps of 0.1, E_{budget} (as a percentage of E_{bound}) ranged from 10% to 100% in steps of 10%, WR ranged from 10 to 100 in steps of 10, and *gain ratio* ranged from 5 to 1000. Then we computed average reward achieved by each heuristic for each system configuration given by U, E_{budget}, WR, and *gain ratio*. Each task set had 30 tasks. Task periods were generated according to a uniform probability distribution where $P_{min} = 10{,}000$ and $P_{max} = 648{,}000$. The mission time X was 10 times the hyperperiod P. Similarly, task weights were generated according to a uniform probability distribution between 1 and WR. We used EDF policy to schedule the 'selected' jobs.

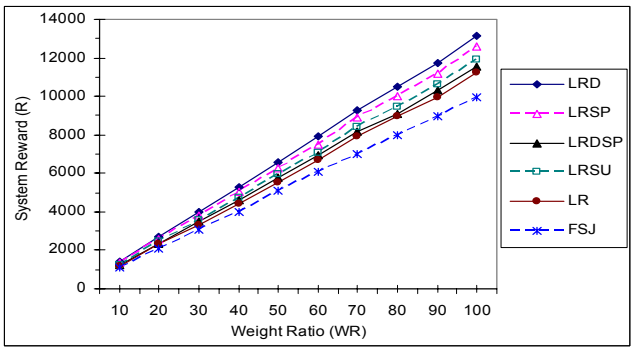

Figure 7. System reward as a function of task weight ratio for U=0.7, E_{budget} =60%, and gain ratio =100

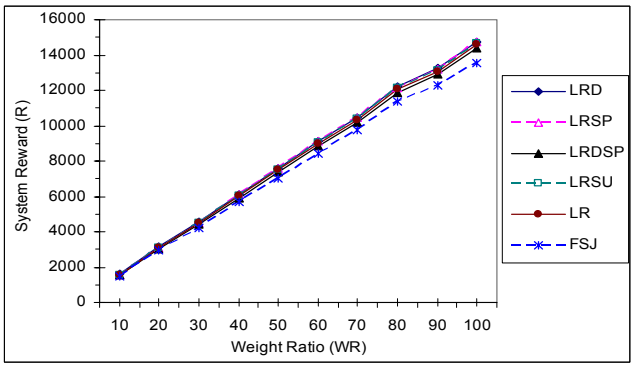

Figure 8. System reward as a function of task weight ratio for U=0.7, E_{budget} =90%, and gain ratio =100

Figure 6 through Figure 8 shows the system reward R as a function of task weight ratio WR for U=0.7, *gain ratio* =100 and for E_{budget} set to 30%, 60%, and 90% respectively. As it can be seen, **LRD** is the best performing heuristic throughout the spectrum, followed by **LRSP**. R increases roughly linearly with WR. When we keep U, *gain ratio*, and E_{budget} constant and vary WR, we are effectively linearly rescaling w_i for each task. As WR increases w_i increases linearly which increases the overall system reward R. Moreover, as WR increases the range

over which w_i varies increases which also makes the difference between the performances of heuristics more significant. On the other hand, as E_{budget} increases the performance margin between the different heuristics diminishes because more deadlines can be met and the system reward becomes closer to its maximum possible value.

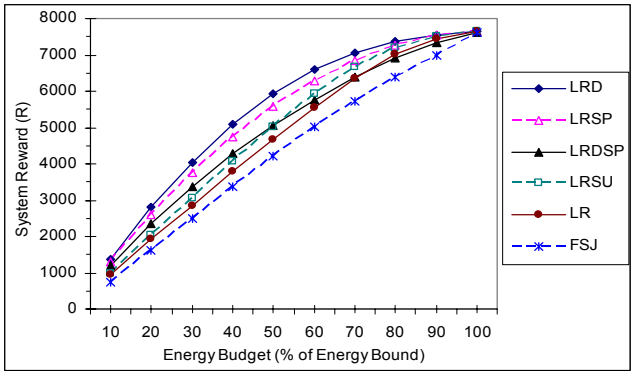

Figure 9. System reward as a function of E_{budget} for U=0.7, WR=50, and gain ratio =100

Figure 9 shows the system reward R as a function of E_{budget} for $U=0.7$, *gain ratio=100* and $WR=50$ for each heuristic. Again, **LRD** outperforms other heuristics. When E_{budget} is small, say 10%, the system is very energy-constrained and only a very small number of jobs can be executed. Under such conditions the difference in performance between the different heuristics is small. As E_{budget} increases the system becomes less energy constrained: more task instances can be executed which increases the overall system reward and the difference between the heuristics becomes more significant. As E_{budget} approaches 100%, the reward achieved by the different heuristics converge and they become exactly equal when $E_{budget}=100\%$, since in this case, the system has enough energy to meet all the deadlines and R reaches its maximum value.

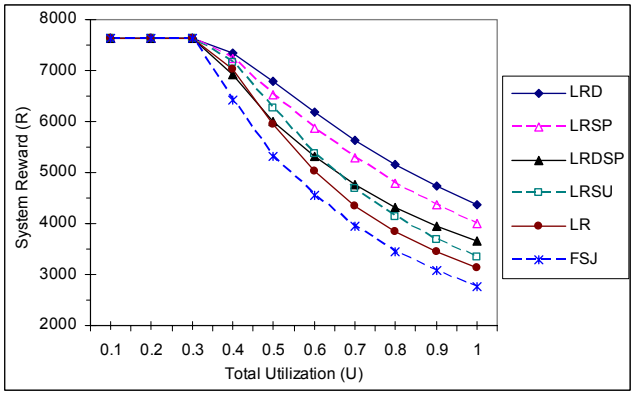

Figure 10. System reward as a function of total utilization for $E_{budget}=E_{bound}(U=0.3)$, WR=50, and gain ratio =100

Figure 10 shows R as a function of U for $E_{budget}=E_{bound}(U=0.3)$, *gain ratio=100*, and $WR=50$ for each heuristic. Unlike Figure 6 through Figure 9 where E_{budget} is recalculated as a percentage of E_{bound} which is also a function of the total utilization U, in this set of experiments E_{budget} is set to a fixed value, namely the energy required to meet all the deadlines when $U=0.3$ (i.e. $E_{bound}(U=0.3)$). When $U \leq 0.3$ the system has enough energy budget to meet all deadlines (i.e. $E_{budget} \geq E_{bound}$) and all the heuristics yield the same reward, which is simply the maximum possible system reward. As U increases from 0.3 to 1.0, R starts to decrease until it reaches its minimum value when $U=1.0$ and the differences in performance between the different heuristics become more significant. As U increases C_i increases linearly for each task while w_i remains constant. Hence, as U increases, more energy has to be spent to execute the same number of jobs, (thus to get the same R), compared to a smaller value of U. Consequently, as U increases R decreases.

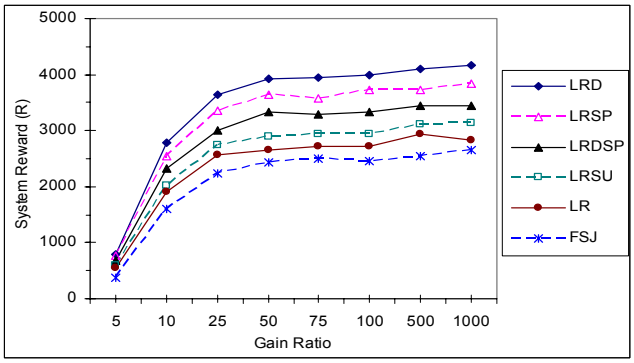

Figure 11. System reward as a function of gain ratio for U=0.7, E_{budget}=30%, and WR=50

Figure 11 shows system reward R as a function of the logarithm of the ratio of normal-mode power to standby-mode power (namely, *log*(*gain ratio*)) for $U=0.7$, $E_{budget}=30$ and $WR=50$ for each heuristic. As *gain ratio* increases the standby-mode power consumption g_{low} decreases and so does the idle energy. Consequently, there is more energy available to be used for executing jobs in the normal mode which increases the system reward. Hence, R increases with *gain ratio*. Note that Figure 11 shows a *cut-off gain ratio* of *50* beyond which the system reward remains practically constant. This is an important result since it shows that decreasing the stand-by power consumption beyond a certain threshold does not provide significant advantage. This behavior can be explained by noting that as *gain ratio* increases (as g_{low} becomes very small) the increase in the available energy for normal mode

becomes practically too small to be used for executing any additional jobs and the system reward saturates.

6. CONCLUSION

In this paper, we proposed a generic performance optimization framework for energy-constrained real-time operating systems. Our approach entails selecting jobs for execution to maximize the number of met deadlines, or alternatively maximize the reward (utility) of the system. We presented an optimal algorithm FSJ that achieves the first objective in time $O(n \log n)$, where n is the number of tasks. We proved that achieving the second objective is NP-Hard. We proposed some fast heuristics for this problem and presented experimental results that showed the relative performance of these heuristics. The best performing heuristic is *LRD* which favors tasks with higher w_i/C_i ratio, which represents the reward return per unit energy spent in executing a task.

REFERENCES

[1] H. Aydin, R. Melhem, D. Mosse and P.M. Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. *Proceedings of the 13th EuroMicro Conference on Real-Time Systems (ECRTS'01)*, pages 225-232, 2001.

[2] H. Aydin, R. Melhem, D. Mosse and P.M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. *Proceedings of the 22nd Real-Time Systems Symposium (RTSS'01)*, pages 95-105, 2001.

[3] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, *Workshop on Parallel and Distributed Real-Time Systems*, 2003.

[4] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, 1979.

[5] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Trans. Computers*, 44(12): 1995.

[6] P. Kumar and M. Srivastava. Power-aware multimedia systems using run-time prediction. *International Conference on Computer Design*, pages 64-69, 2001.

[7] C.L. Liu, J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real time environment, *Journal of the ACM*, 17(2). 1973.

[8] J. Liu. *Real-Time Systems*. Prentice Hall, NJ, 2000.

[9] J. Liu, P.H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints for mission-critical embedded systems. *Design Automation Conference*, pages 840-845, 2001.

[10] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, 2001.

[11] P. Pillai and K.G. Shin. Real-time dynamic voltage scaling for low power embedded operating systems. *Symposium on operating systems principles*, pages 89-102, 2001.

[12] G. Quan and X. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. *Design Automation Conference*, pages 828-833, 2001.

[13] C. Rusu, R. Melhem and D. Mosse. Maximizing the system value while satisfying time and energy constraints. *Real-Time Systems Symposium*, pages 246 -255, 2002.

[14] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. *Design Automation Conference*, pages 134–139, 1999.

[15] Y. Shin, S. Lee and J. Kim. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2): 20-30, 2001.

[16] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. IEEE Annual Foundations of Computer Science, pages 374–382, 1995.