

Maximizing the System Value while Satisfying Time and Energy Constraints *

Cosmin Rusu, Rami Melhem, Daniel Mossé
Computer Science Department, University of Pittsburgh
Pittsburgh, PA 15260
(*rusu,melhem,mosse*)@cs.pitt.edu

Abstract

Typical real-time scheduling theory has addressed deadline and energy constraints as well as deadline and reward constraints simultaneously in the past. However, we believe that embedded devices with varying applications typically have three constraints that need to be addressed: energy, deadline, and reward. These constraints play important roles in the next generation of embedded devices, since they provide users with a variety of QoS-aware trade-offs. An optimal scheme would allow the device to run the most critical and valuable applications, without depleting the energy source while still meeting all deadlines. In this paper we propose a solution to this problem for typical control systems, such as frame-based task sets. We devise two algorithms that closely approximate the optimal solution while taking only a fraction of the runtime of an optimal solution.

1 Introduction

The current developments in embedded systems technology have been largely responsible for the promotion of mobile, wireless, systems-on-a-chip and other “computing-in-the-small” devices. Most of these devices have energy constraints, embodied by a battery that has a finite lifetime. Therefore, an essential element of these embedded systems is the way in which power is managed.

In addition to the power management needs, some of these devices execute real-time applications, in which producing timely results is typically as important as producing logically correct outputs. An admission control algorithm can be used to only accept tasks that will finish before their deadlines. The main problem with admission control algorithms is that they are conservative, and underutilize resources.

An alternative is to allow systems to run above the load restrictions imposed by real-time admission control algorithms. These overloaded systems lend themselves naturally to scenarios in which some applications are executed in lieu of more important applications; the value/reward can be assigned to each application. The problem, in this case, is how to choose applications that will maximize the overall reward given to the system, such that all applications chosen will execute within their respective deadlines.

The three constraints mentioned above, namely *energy*, *deadline*, and *reward* play important roles in the current generation of embedded devices. An optimal scheme would allow the device to run the most valued applications, without depleting the energy source while still meeting all deadlines.

Note that this problem differs from minimizing power consumption due to the extra constraints considered, namely deadlines and CPU utilization. Clearly, minimizing the energy consumption of applications is useful, but does not consider the value/reward characteristics of different applications. For example, it may be better to run an important application that consumes more energy than two less important applications that consume much less energy.

Considering these three constraints simultaneously (reward, energy, and deadlines) is important since it allows system designers to determine the most important components of their system, or allows them to emphasize a subset of the system over another in a dynamic fashion. An example of such flexibility is when one decides to maximize mission life-time instead of having a fixed mission time within which performance should be maximized.

The rest of this paper is organized as follows: We first describe related work. Section 2 explains in detail the task model and defines the problem. In Section 3 we present two algorithms that closely approximate the optimal solution. Section 4 presents experimental results obtained through simulation. In Section 5 we conclude the paper.

*This work has been supported by the Defense Advanced Research Projects Agency through the PARTS (Power-Aware Real-Time Systems) project under Contract F33615-00-C-1736.

1.1 Related Work

The issue of assignment of CPU cycles to different tasks has been studied through scheduling and operations research for decades. In the mid-80s, researchers started considering the tradeoff between time and other metrics, such as value/reward [5]. In the late-90s, researchers started considering a similar tradeoff, but focusing on the tradeoff between energy and time [27]. Below we describe representative works in these two fields. However, none of these works addressed the general framework with the three types of constraints we consider here, namely *energy*, *deadline*, and *reward/value*.

Rewards and real-time

The IC (Imprecise Computation) [17, 23] and IRIS (Increased Reward with Increased Service) [7, 16] models were proposed to enhance the resource utilization and provide graceful degradation in real-time systems. In the IC model every real-time task is composed of a mandatory part (which must finish before the task deadline to yield an output of minimal quality) and an optional part. The longer the optional part executes, the better the quality of the result. Several efficient algorithms have been proposed to solve the scheduling problem of aperiodic tasks [17, 23]. A common assumption in these studies is that the quality of the results produced is a linear function of the precision error; more general error functions are not usually addressed.

An alternative model is the IRIS model with no upper bounds on the execution times of the tasks and no separation between the mandatory and optional parts (i.e., tasks may be allotted no CPU time). Typically, a non-decreasing concave reward function is associated with each task's execution time. In [6, 7] the problem of maximizing the total reward in a system of aperiodic tasks was addressed and an optimal solution for static task sets was presented, as well as two extensions that include mandatory parts and policies for dynamic task arrivals. An optimal algorithm assuming concave reward functions and periodic real-time applications was presented in [3]. Both IC and IRIS focus on linear and concave (logarithmic for example) functions representing applications such as image and speech processing [4, 11, 26] or multimedia applications [21]. The case of real applications with no reward for partial executions or step functions has been shown in [17] to be NP-Complete. Furthermore, the reward-based scheduling problem for convex reward functions is NP-Hard [3].

In [21] a QoS-based resource allocation model (GRAM) was proposed for periodic applications. The reward functions are in terms of utilization of resources and an iterative algorithm was presented for the case of one resource and multiple QoS dimensions; the QoS dimensions may be either dependent or independent. In [22], the GRAM

work is continued by the authors with the solution for a particular audio-conferencing application with two resources (CPU cycles and network bandwidth) and one QoS dimension (sampling rate). Several resource tradeoffs (compression schemes to reduce network bandwidth while increasing the number of CPU cycles) are also investigated, assuming linear utility and resource consumption functions.

Variable voltage scheduling and real-time

The variable voltage-scheduling (VVS) framework, which involves dynamically adjusting the voltage and frequency of the CPU, has recently become a major research area. Cubic energy savings [27, 14] can be achieved at the expense of just linear performance loss. For real-time systems, VVS schemes focus on minimizing energy consumption in the system while still meeting the deadlines. Yao et al. [27] provided a static off-line scheduling algorithm, assuming aperiodic tasks and worst-case execution times (WCET). Heuristics for on-line scheduling of aperiodic tasks while not hurting the feasibility of periodic requests are proposed in [13]. Non-preemptive power aware scheduling is investigated in [12]. For periodic tasks with identical periods, the effects of having an upper bound on the voltage change rate are examined in [14]. Slowing down the CPU whenever there is a single task eligible for execution was explored in [24]. VVS in the context of soft deadlines was investigated in [18]. Cyclic and EDF scheduling of periodic hard real-time tasks on systems with two (discrete) voltage levels have been investigated in [15]. The static solution for the general periodic model where tasks have potentially different power characteristics is provided in [1]. Real-time applications exhibit a large variation in actual execution times [9] and WCET is too pessimistic. Thus, a lot of research was directed at dynamic slack-management techniques [2, 10, 20, 25]. Many other VVS papers appeared in recent conferences and workshops, such as COLP'01 or PACS'02.

It was proved in [2] that the problem of minimizing the energy consumption assuming WCET for tasks and convex power functions is equivalent to the problem of maximizing the rewards for concave reward functions assuming all the tasks run at the maximum speed.

In this work we address the problem of maximizing the rewards assuming the VVS framework and a limited energy budget for frame-based task sets. Our goal is to maximize the rewards without exceeding the deadline and the total energy available, which can be provided by an exhaustible source such as a battery. The algorithms we propose determine which tasks to execute and the speeds these selected tasks should run so that the total reward of the system is maximized while meeting both the timing and the energy

constraints.

Concurrently with our work similar research combined the three constraints (time, energy and reward) for the case of IRIS tasks in [8]. An algorithm was developed to maximize the system value by an energy-aware allocation of resources. However, the task model in [8] does not include voltage or frequency scaling.

2 Task model

We assume a frame-based task model, which we describe next. There are N available periodic tasks in the system, all ready at time zero. The task set is denoted by $\mathbf{T}=\{T_1, T_2, \dots, T_N\}$. All task periods are identical and all task deadlines are equal to their period. The common deadline/period (also known as frame length) is denoted by D . A frame consists of a subset of tasks which are selected for execution. The execution of the frame is to be repeated. It is not a requirement that all tasks must be scheduled. However, a task cannot be selected more than once during a frame.

The tasks are to be executed on a variable voltage processor with the ability to dynamically adjust its frequency and voltage on application requests. There are M available frequencies (clock rates or CPU speeds), $\{f_1, f_2, \dots, f_M\}$. Each task can run at any of the available speeds and we say that a task runs at speed level k if the speed of the task is set to f_k . By placing tasks that run at the same frequency next to each other, the maximum number of speed changes that can occur during a frame is $\min(M, N)$. We assume the overhead of $\min(M, N)$ speed changes is negligible compared to the deadline D , or that it was already subtracted from D .

We also assume that the task worst-case execution time and energy consumption are known for all tasks and all speed levels. The execution time of task T_i running at speed level j is denoted by $t_{i,j}$. Similarly, the energy consumption of task T_i running at speed level j is denoted by $e_{i,j}$.

Associated with each task T_i there is a task value v_i (also called task reward or utility). The value of the system is defined as the sum of task values for all tasks that are selected for execution. It is the ultimate goal to find a subset of tasks $S \subseteq \{1, 2, \dots, N\}$ that maximizes the system value $\sum_{i \in S} v_i$. For all tasks $i \in S$ the speed level $s_i \in \{1, 2, \dots, M\}$ must also be determined. There are two major constraints on the system:

- The *timing constraint* imposed by the global deadline, D . Each task selected for execution must finish before this deadline, D .
- The *energy constraint* imposed by the amount of energy available in the system, E_{max} . The total energy consumed by the selected tasks cannot exceed E_{max} .

Thus, the problem is to find the subset S and the speeds $s_i, \forall i \in S$ so that to

$$\text{maximize} \quad \sum_{i \in S} v_i \quad (1)$$

$$\text{subject to} \quad \sum_{i \in S} t_{i,s_i} \leq D \quad (2)$$

$$\sum_{i \in S} e_{i,s_i} \leq E_{max} \quad (3)$$

$$S \subseteq \{1, 2, \dots, N\} \quad (4)$$

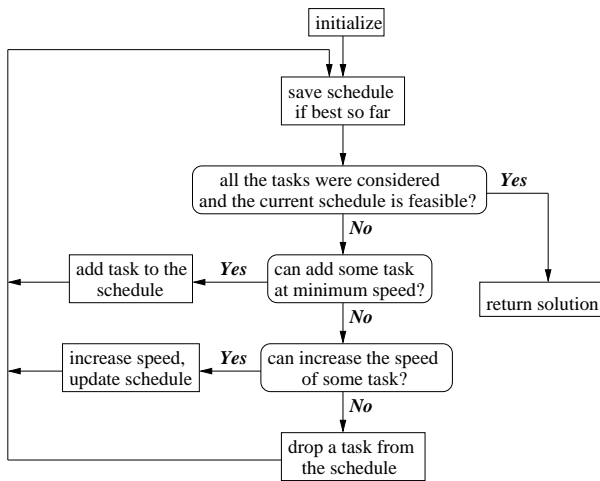
$$s_i \in \{1, 2, \dots, M\} \quad (5)$$

Inequality (2) guarantees that the timing constraint is satisfied, and inequality (3) guarantees that the energy budget is not exceeded. As shown in the Appendix, the problem defined by (1)-(5) is NP-hard. Therefore, we relax the maximization objective in (1) and look for solutions that approximate the optimal solution.

3 Algorithms *REW-Pack* and *REW-Unpack*

We have tried many algorithms to solve equations (1)-(5). Some of these algorithms were based on sorting all tasks at all speed levels according to some metric that combines the three constraints (energy, deadline and reward). Tasks were then added to the schedule in one traversal of the sorted list of tasks until the timing or energy constraint could no longer be satisfied. This approach was too conservative and almost invariably lead to poor utilization of one of the resources (energy or time) and poor system values. Algorithms that dynamically modify the schedule based on the resource usage (while still considering task values) turned out much more rewarding in terms of resource utilization and system value. Several heuristics for task selection were considered, ignoring or including the task values, favoring tasks with low energy consumption or low time requirements, or considering all the three constraints at once. Two algorithms were found to closely approximate the optimal solution. We describe the two algorithms in this section, followed by a quantitative evaluation in the next section.

We assume that tables exist that store the task values $v(i)$, running times $t(i, j)$ and energy requirements $e(i, j)$ for all tasks $i \in \{1, 2, \dots, N\}$ and all speed levels $j \in \{1, 2, \dots, M\}$. The algorithms are based on adapting the schedule by adding and dropping tasks until all the tasks are considered. We also use two boolean arrays, $selected(i)$ and $considered(i)$ of size N , to store information about the status of all tasks. Initially, we start with an empty schedule ($selected(i) = false$) and no task is considered yet ($considered(i) = false$). The set of selected tasks (initially empty) is defined as $S = \{i | selected(i) = true\}$.

Figure 1: Flowchart of *REW-Pack*

Two variables, *time* and *energy*, store the total running time of the schedule ($time = \sum_{i \in S} t(i, s_i)$) and the total energy consumed ($energy = \sum_{i \in S} e(i, s_i)$) and are initialized to zero. V stores the system value for the current schedule ($V = \sum_{i \in S} v(i)$) and SV stores the *system value*, that is, the largest value of V encountered thus far. Finally, an array, $speed(i)$, of size N , stores the speeds of all tasks.

3.1 The *REW-Pack* Algorithm

The flowchart of the *REW-Pack* algorithm is presented in Figure 1. The three major components (add task, drop task and increase speed) are described next in detail.

Add a task A new task is added (always at the minimum speed) to the current schedule if all of the following criteria are met:

- It was not considered before ($considered(i) = false$).
- The current schedule is feasible ($time \leq D$).
- By adding the task to the current schedule at the minimum speed the energy budget is not exceeded ($energy + e(i, 1) \leq E_{max}$).
- Among all the tasks T_i that satisfy the above criteria, select the one that has the largest ratio $\frac{v(i)}{t(i, 1) \cdot e(i, 1)}$.

A new task is always added if possible. The task added must have a good (large) value, a reasonable (small) running time and a reasonable (small) energy consumption. Hence the metric used to decide which task is best to add is proportional to the reward and inversely proportional to the time

and the energy required by the task. The task with the highest metric is considered the best. In our experiments, metrics that do not consider all parameters (i.e., task value, task energy and task time) failed to give good approximations of the optimal solution.

Observe that for each task, the smaller the speed, the larger the value of the metric (since energy increases more than linearly with the speed while time decreases approximately linearly and the task value remains the same regardless of the running speed). Thus, it is reasonable to start with the smallest speed (level 1) and later increase the task's speed. Also observe that exceeding the deadline is allowed. We noticed during experiments that without this enhancement premature task drops occur, hurting the accuracy of the solution. However, we do not allow exceeding the energy budget, because our experiments have shown that allowing the energy budget to be exceeded typically leads to poor results.

Increase speed of a task If no task can be added to the schedule, the algorithm *packs* tasks to make room for other not yet selected tasks, where *packing* means to increase the speed of one of the selected tasks, always to the next higher speed level. The task chosen for a speed increase must satisfy the following:

- It must be selected in the current schedule ($selected(i) = true$).
- It is not running at the maximum speed ($s_i \neq M$).
- By increasing its speed to the next higher speed level the energy budget is not exceeded ($energy + e(i, s_i + 1) - e(i, s_i) \leq E_{max}$).
- Among all selected tasks T_i it has the highest ratio $\frac{\Delta t}{\Delta E}$, where $\Delta t = t(i, s_i) - t(i, s_i + 1)$ and $\Delta E = e(i, s_i + 1) - e(i, s_i)$.

Packing reduces the total execution time and increases the energy consumption. The best candidates are considered the tasks that create a lot of room (time or slack) for the remaining tasks while not significantly increasing the energy consumption. Task values do not play any role here as the total reward is not changed by the packing operation. Interestingly, when we used the same metric for packing as we did for task selection (i.e., increasing the speed of the task with the smallest ratio $\frac{v(i)}{t(i, s_i) \cdot e(i, s_i)}$) we obtained poor results.

Drop a task If the previous two steps fail, a task is eliminated from the current schedule. The task that is dropped satisfies:

- It is selected in the current schedule ($selected(i) = true$).

```

1 Initialize:  $selected(i) = false; considered(i) = false \forall i \in \{1, 2, \dots, N\}; energy = 0; time = 0; SV = 0; V = 0$ 
2 If  $time \leq D$  and  $SV < V$ 
  2.1  $sol\_selected(i) = selected(i); sol\_speed(i) = speed(i), \forall i \in \{1, 2, \dots, N\}$ 
  2.2  $SV = V$ 
3 If  $(\exists i, considered(i) == false)$  or  $(time > D)$  do
  3.1  $i = add\_task()$ 
  3.2 If  $i \neq -1$ 
    3.2.1  $selected(i) = true; considered(i) = true; energy = energy + e(i, 1); speed(i) = 1; time = time + t(i, 1); V = V + v(i)$ 
    3.2.2 Go to step 2
  3.3  $i = increase\_speed()$ 
  3.4 If  $i \neq -1$ 
    3.4.1  $energy = energy + e(i, speed(i) + 1) - e(i, speed(i)); time = time + t(i, speed(i) + 1) - t(i, speed(i)); speed(i) = speed(i) + 1$ 
    3.4.2 Go to step 2
  3.5  $i = drop\_task()$ 
  3.6  $energy = energy - e(i, speed(i)); time = time - t(i, speed(i)); V = V - v(i); selected(i) = false$ 
  3.7 Go to step 2
4 Return solution  $(sol\_selected, sol\_speed, SV)$ 

```

Figure 2: The *REW-Pack* algorithm

- Among all selected tasks, T_i , it has the smallest ratio $\frac{v(i)}{t(i, s_i) \cdot e(i, s_i)}$.

When dropping a task is necessary, the task with the worst metric (i.e., smallest $\frac{v(i)}{t(i, s_i) \cdot e(i, s_i)}$) is dropped. Task values need to be considered here since it is generally better to keep tasks with high values and drop the less important ones. Once a task is dropped, it is never added again. We also experimented with allowing tasks to be added or dropped k times in the schedule; there was an increase in the running time of the algorithm by a factor of k but no significant improvement in the accuracy of the solution.

The algorithm is shown in Figure 2. $add_task()$, $drop_task()$ and $increase_speed()$ all return the task number or -1 if no task can be chosen. Additional vectors are used to store the solution tasks ($sol_selected$) and speeds (sol_speed).

The complexity of the *REW-Pack* algorithm can be analyzed as follows. Each task is added at most once and dropped at most once. For each task we can increase its speed at most $M - 1$ times. Determining what task to pick takes $\log N$ time for all functions (add, increase and drop). Thus, the complexity of the algorithm is $O(MN \log N)$.

```

1 Initialize:  $selected(i) = false; considered(i) = false \forall i \in \{1, 2, \dots, N\}; energy = 0; time = 0; SV = 0; V = 0$ 
2 If  $energy \leq E_{max}$  and  $SV < V$ 
  2.1  $sol\_selected(i) = selected(i); sol\_speed(i) = speed(i), \forall i \in \{1, 2, \dots, N\}$ 
  2.2  $SV = V$ 
3 If  $(\exists i, considered(i) == false)$  or  $(energy > E_{max})$  do
  3.1  $i = add\_task()$ 
  3.2 If  $i \neq -1$ 
    3.2.1  $selected(i) = true; considered(i) = true; energy = energy + e(i, M); speed(i) = M; time = time + t(i, M); V = V + v(i)$ 
    3.2.2 Go to step 2
  3.3  $i = decrease\_speed()$ 
  3.4 If  $i \neq -1$ 
    3.4.1  $energy = energy + e(i, speed(i) - 1) - e(i, speed(i)); time = time + t(i, speed(i) - 1) - t(i, speed(i)); speed(i) = speed(i) - 1$ 
    3.4.2 Go to step 2
  3.5  $i = drop\_task()$ 
  3.6  $energy = energy - e(i, speed(i)); time = time - t(i, speed(i)); V = V - v(i); selected(i) = false$ 
  3.7 Go to step 2
4 Return solution  $(sol\_selected, sol\_speed, SV)$ 

```

Figure 3: The *REW-Unpack* algorithm

3.2 The *REW-Unpack* Algorithm

The idea behind the *REW-Unpack* algorithm is basically the same as *REW-Pack*. The difference is that instead of adding tasks at the minimum speed and then packing to create time for tasks still to be selected, the search goes in quite the opposite direction: tasks are added at the maximum speed and the schedule is unpacked (i.e., a task is selected and its speed decreased) to create energy for the remaining tasks.

The function $increase_speed()$ is replaced with $decrease_speed()$. The same metrics are used for adding and dropping tasks and the opposite metric is used to decide which task's speed to decrease (the task that saves the most energy while increasing the execution time the least is considered the best, that is, the task with the highest $\frac{\Delta E}{\Delta t}$ is selected). Analogously to *REW-Pack*, exceeding the energy budget is allowed while exceeding the deadline is not.

The algorithm is shown in Figure 3.

4 Experimental Results

We simulated both algorithms on the same task sets and, for relatively small task sets, compared our solution with the optimal solution, obtained through an exhaustive search.

Table 1: Intel XScale speed settings and voltages

Speed Level	Speed (MHz)	Voltage (V)
1	150	0.75
2	400	1.0
3	600	1.3
4	800	1.6
5	1000	1.8

We define the absolute error for any of the two algorithms to be $\frac{SV_{OPT} - SV}{SV_{OPT}}$, where SV represents the system value (reward) resulting from the algorithm and SV_{OPT} is the optimal system value. The average error for several experiments is defined as the arithmetic mean of the absolute errors for each experiment.

The simulations are described by the following parameters:

- N - number of tasks
- M - number of speed levels
- $t_{i,j}, e_{i,j}$ - time and energy requirements
- D - deadline
- E_{max} - available energy
- v_i - task values

The maximum deadline, Max_D , is defined as $Max_D = \sum_{i=1}^N t_{i,1}$, that is the total execution time of the tasks at minimum speed. The maximum energy, Max_E , is defined as $Max_E = \sum_{i=1}^N e_{i,M}$, that is the total energy requirement for all tasks if running at the maximum speed. Clearly, if $D \geq Max_D$ the timing constraint cannot be violated. Similarly, if $E_{max} \geq Max_E$ the available energy cannot be exceeded. Two parameters: α and β describe the available time and energy in the system. The deadline was generated using the formula $D = \alpha \cdot Max_D$ and the energy was generated by $E_{max} = \beta \cdot Max_E$, where $\alpha \in [0, 1]$ and $\beta \in [0, 1]$.

We simulated the Intel XScale architecture, with 5 speed levels. The running speeds and their corresponding voltages are presented in Table 1. For each task, its execution time at minimum speed $t_{i,1}$ was randomly generated in the range $[1, 100]$. The running time of task T_i at speed level j was then computed as $t_{i,j} = t_{i,1} \frac{f_1}{f_j}$, thus the running time is inversely proportional with the speed. For the power consumption of task T_i at speed level j , we use the formula $P_{ij} = a_i \cdot Voltage(j)^2 \frac{f_1}{f_j}$. Thus, the power is proportional with the normalized speed and the square of the voltage. a_i is an activity factor different for each task, proportional with the dynamic switching caused by the task and randomly generated in the range $[0.8, 1.2]$. The energy requirement $e_{i,j}$ is then computed as $e_{i,j} = P_{i,j} \cdot t_{i,j}$, that is the power multiplied with the time. Task values were generated randomly in the range $[1, 100]$.

First we compared the two algorithms with a simplified version of *REW-Pack* that does not take task val-

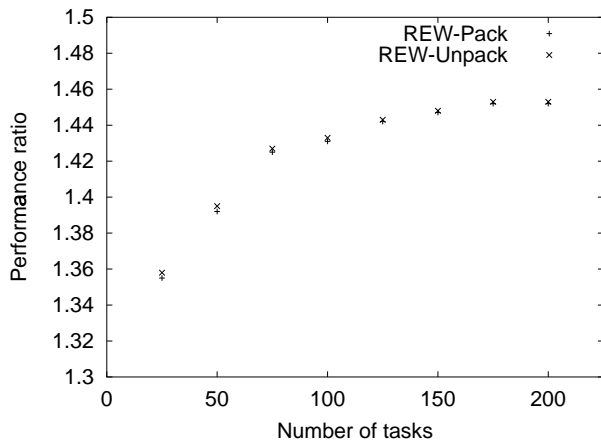
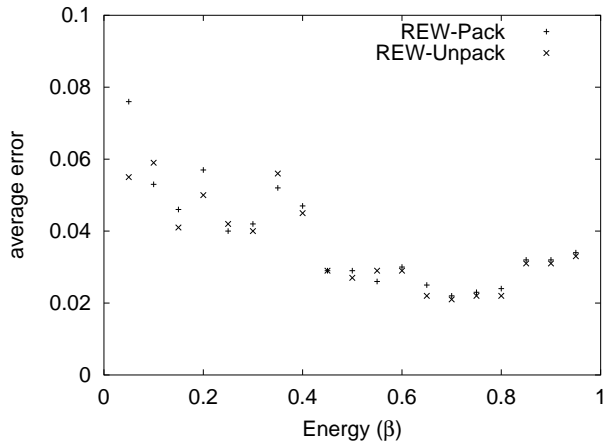


Figure 4: Comparison of the two algorithms with a simplified version of *REW-Pack* ($\alpha, \beta \in [0.1, 0.3]$)

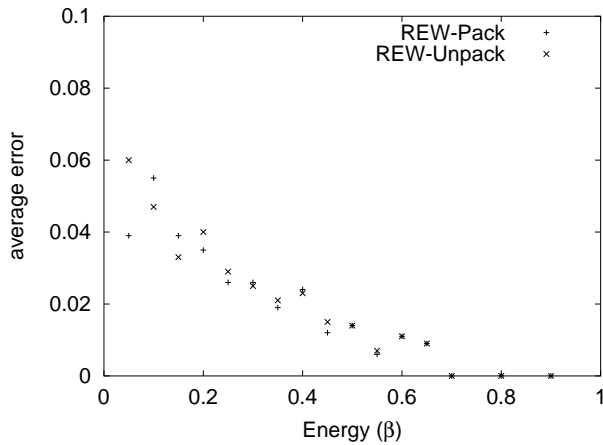
ues into consideration and randomly selects which tasks to add/drop/pack from the subset of tasks satisfying the add/drop/pack criteria. α and β were randomly generated in the range $[0.1, 0.3]$ for each simulation. Task sets with 25 to 200 tasks were simulated and 1000 experiments were averaged for each point in the graphs. The performance ratio shown in Figure 4 is defined as the system value returned by the algorithm (*REW-Pack* or *REW-Unpack*) divided by the system value of the simplified *REW-Pack*. It is clear that the two algorithms have almost identical performance. As expected, on average and on each particular simulation, they consistently outperformed the simplified *REW-Pack*.

Figure 5 shows the average absolute error of the algorithms as a function of the available energy. Task sets with $N = 10$ tasks were simulated for very tight deadlines ($\alpha = 0.1$) and more relaxed deadlines ($\alpha = 0.2$ and $\alpha = 0.3$). The average for 100 simulations was computed for each point. The same averages hold for higher number of simulations. The maximum error for each point is typically 10% – 20%. Experiments show that as α increases beyond 0.4, both algorithms find the optimal solution most of the time and the average error becomes zero. Also, as the amount of energy available increases, the average error of both algorithms tends to decrease. No algorithm is a clear winner, as the previous experiment suggested. The worst performance is when there is little slack in the system (i.e., small α values) combined with a reduced amount of energy (i.e., small β values). In this case even the optimal can select only two or three tasks; if the algorithms do not pick exactly the same tasks as the optimal, the error is likely to increase.

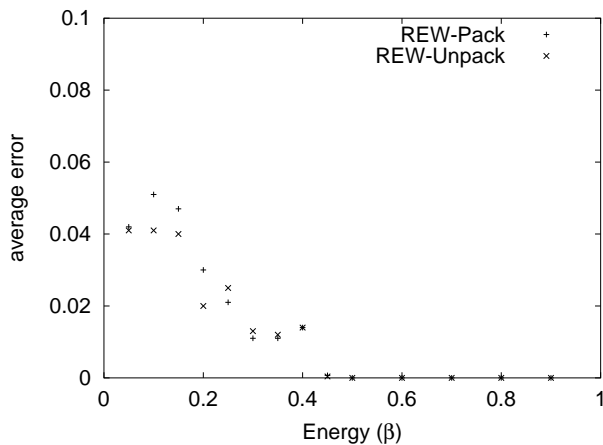
We noticed that although the two *REW* algorithms search for a solution from quite opposite directions, they



(a) $\alpha = 0.1$



(b) $\alpha = 0.2$



(c) $\alpha = 0.3$

Figure 5: Average error as a function of β (available energy) for 10 tasks

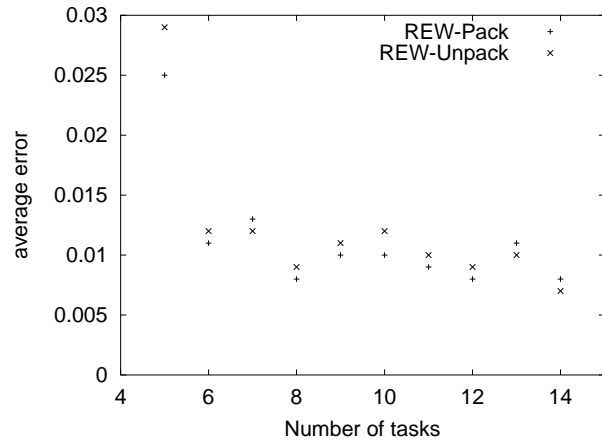


Figure 6: Average error as a function of N ($\alpha = 0.3, \beta = 0.3$)

usually select the same tasks in the end. Also, the tasks selected by the algorithms are usually the same that the optimal chooses. In fact, for each point in the graphs the algorithms were equal to the optimal at least 35% of the time (35% was obtained for *REW-Pack* at $\alpha = 0.1$ and $\beta = 0.35$). We hoped that *REW-Pack* would perform better on time-constrained task sets and *REW-Unpack* would have better results on energy-constrained task sets. It turns out that the time and energy are equally important (except for cases when D or E_{max} are too large to be used entirely given the other constraint) and both algorithms return schedules that use on average more than 90% of both the available time and energy.

When the optimal algorithm outperforms our *REW* algorithms, it usually manages to pick one more task or it selects the same number of tasks but one or two tasks are different. The higher the number of tasks in the optimal solution, the higher the number of tasks selected by our heuristics algorithms and thus the smaller the absolute error.

Unfortunately, the exponential nature of the optimal makes it impossible to compute the absolute error for high values of N . There is experimental evidence, however, that the absolute errors do not increase (rather, they actually decrease) as the number of tasks increases. For example, in Figure 6, where we simulated task sets with 5 to 14 tasks and $\alpha = 0.3$ and $\beta = 0.3$, we can see this trend. In the figure, each point is the average error of 100 runs.

In order to avoid the complexity of finding the optimal solution to evaluate our algorithms, we designed an experiment in which we constructed sets of tasks with known optimal solutions, and ran our algorithms against those task sets. The task sets were constructed as follows: the deadline D was set to $D = \sum_{i=1}^N t_{i,k_i}$ and the maximum energy E_{max}

was set to $E_{max} = \sum_{i=1}^N e_{i,k_i}$, where $k_i \in \{1, 2, \dots, M\}$ was randomly generated for each task. Thus, if each task T_i runs at speed level k_i , all tasks are schedulable and the optimal reward is simply $SV_{OPT} = \sum_{i=1}^N v_i$. We ran 1000 simulations on task sets with 50, 100 and 200 tasks. We do not show a graph for the results, because both our heuristic algorithms returned the optimal solution in all 1000 simulation runs.

5 Conclusions

We presented two algorithms for the problem of maximizing the system value given time and energy constraints. The goal is to determine which tasks to execute and the speeds to execute the selected tasks on a variable voltage processor so that the total value of the system (defined as the sum of task values for all tasks selected for execution) is maximized without violating the timing and energy constraints. While real-time researchers have dedicated much effort to reward-based scheduling and power-aware scheduling, the problems of maximizing the reward (system value) and minimizing the energy consumption are usually treated separately. Further, continuous speeds and/or continuous reward functions (increased reward with increased service) are usually assumed. In this work we departed from such assumptions to address the case of discrete speeds and discrete task values, with no reward for partial execution.

The problem is NP-hard and an optimal solution requires an exponential time solution. However, we show by simulation that the proposed algorithms closely approximate the optimal. The worst-case time complexity of the algorithms is just $O(MN \log N)$, where N is the number of tasks in the system and M is the number of available speeds. A small running time allows a scheduler to quickly adapt to changes in the system such as tasks becoming unavailable, new tasks being added to the system or new timing and energy constraints. In most current variable voltage processors, the number of speed levels is typically a small constant (5-10). A graphical demonstration of our heuristics is available at <http://www.cs.pitt.edu/PARTS/demos>. We thank Patrick Lanigan for writing the Java applet and for his contributions to the algorithms.

References

- [1] H. Aydin, R. Melhem, D. Mossé, 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)*, Delft, Netherlands, June 2001
- [2] H. Aydin, R. Melhem, D. Mossé and P. M. Alvarez: Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems, *Proceedings of Real-Time Systems Symposium, 2001*
- [3] H. Aydin, R. Melhem, D. Mossé, P. M. Alvarez: Optimal Reward-Based Scheduling for Periodic Real-Time Tasks, *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, December 1999
- [4] E. Chang, A. Zakhor: Scalable Video Coding using 3-D Subband Velocity Coding and Multi-Rate Quantization, *IEEE Int. Conf. On Acoustics, Speech and Signal Processing, July 1993*
- [5] R. K. Clark, E. D. Jensen and F. D. Reynolds: An Architectural Overview of the Alpha Real-Time Distributed Kernel, *USENIX Workshop on MicroKernels and Other Kernel Architectures*, April 1992
- [6] J. K. Dey, J. Kurose, D. Towsley, C. M. Krishna and M. Girkar: Efficient On-Line Processor Scheduling for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks, *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 217-228, May 1993
- [7] J. K. Dey, J. Kurose and D. Towsley: On-Line Scheduling Policies for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks, *IEEE Transactions on Computers* 45(7):802-813, July 1996
- [8] D. Kang, S. P. Crago and J. Suh: A Fast Resource Synthesis Technique for Energy-Efficient Real-Time Systems, *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Austin, December 2002
- [9] R. Ernst and W. Ye: Embedded Program Timing Analysis based on Path Clustering and Architecture Classification, *Computer-Aided Design (ICCAD'97)*, pp. 598-604, 1997
- [10] F. Gruian: Hard Real-Time Scheduling Using Stochastic Data and DVS Processors, *Proceedings of International Symposium on Low Power Electronics and Design*, pp. 46-51, 2001
- [11] W. Feng, J. W.-S. Liu: An extended imprecise computation model for time-constrained speech processing and generation, *Proceedings of the IEEE Workshop on Real-Time Applications*, May 1993
- [12] I. Hong, D. Kirovski, G. Qu, M. Potkonjak and M. Srivastava: Power Optimization of Variable Voltage Core-based Systems, *Proceedings of the 35th Design Automation Conference (DAC'98)*, 1998
- [13] I. Hong, M. Potkonjak and M. Srivastava: On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processors, *Computer-Aided Design (ICCAD'98)*, pp. 653-656, 1998
- [14] I. Hong, G. Qu, M. Potkonjak and M. Srivastava: Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors, *Proceedings of the 19th IEEE*

Real-Time Systems Symposium (RTSS'98), Madrid, December 1998

- [15] C. M. Krishna and Y. H. Lee: Voltage Clock Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems, *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00), Washington D. C., May 2000*
- [16] C. M. Krishna and K. G. Shin: Real-time Systems, *Mc Graw-Hill, New-York 1997*
- [17] J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao, W. Zhao: Algorithms for scheduling imprecise computations, *IEEE Computer*, 24(5):58-68, May 1991
- [18] J. R. Lorch and A. J. Smith: Improving Dynamic Voltage Scaling Algorithms with PACE, *Proceedings of the ACM SIGMETRICS 2001 Conference, Cambridge, MA, June 2001*
- [19] S. Martello and P. Toth: Knapsack Problems: Algorithms and Computer Implementation. *Wiley and Sons, 1997.*
- [20] D. Mossé, H. Aydin, B. Childers, R. Melhem: Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications, *Workshop on Compilers and Operating Systems for Low Power (COLP'00), Philadelphia, PA, October 2000*
- [21] R. Rajkumar, C. Lee, J. P. Lehoczky, D. P. Siewiorek: A Resource Allocation Model for QoS Management, *Proceedings of 18th IEEE Real-Time Systems Symposium (RTSS'97), December 1997*
- [22] R. Rajkumar, C. Lee, J. P. Lehoczky, D. P. Siewiorek: Practical Solutions for QoS-based Resource Allocation Problems, *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98), December 1998*
- [23] W.-K. Shih, J. W.-S. Liu, J.-Y. Chung: Algorithms for scheduling imprecise computations with timing constraints, *SIAM Journal on Computing*, 20(3):537-552, July 1991
- [24] Y. Shin and K. Choi: Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems, *Proceedings of the 36th Design Automation Conference (DAC'99), 1999*
- [25] D. Shin, J. Kim and S. Lee: Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications, *IEEE Design and Test of Computers*, 18(23):20-30, March 2001
- [26] C. J. Turner, L. L. Peterson: Image Transfer: An end-to-end design, *SIGCOMM Symposium on Communications Architectures and Protocols, August 1992*
- [27] F. Yao, A. Demers and S. Shankar: A Scheduling Model for Reduced CPU Energy, *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995
- [28] H.M. Weingartner and D.N. Ness: Methods for the resolution of the Multi-dimensional 0/1 Knapsack Problem. *Operations Research*, 15:83-103, 1967.

A Maximizing rewards while guaranteeing time and energy constraints is NP-hard

In Section 2 we claimed that the problem of choosing a set of task such that they maximize the total system reward and still meet the time and energy constraints (deadlines and energy budgets) as described by equations (1)-(5) is NP-hard. First we show how the problem can be transformed to a special case of the *0-1 Multidimensional Knapsack problem* [19]. Then we show that the problem is harder than the *0-1 Bidimensional Knapsack problem*, which is known to be NP-hard.

The *0-1 Multidimensional Knapsack* has the following formulation:

$$\text{maximize} \quad c \cdot x \quad (6)$$

$$\text{subject to} \quad A \cdot x \leq b \quad (7)$$

$$x_i \in \{0, 1\} \quad (8)$$

where $x = [x_1, x_2, \dots, x_n]^t$ is a column vector of 0 – 1 variables, $c = [c_1, c_2, \dots, c_n]$ is a row vector of integers, A is a matrix with m rows (constraints) and n columns with integer values and $b = [b_1, b_2, \dots, b_m]^t$ is a column vector of size m with integer values. A , b and c are given and the solution is the 0 – 1 vector x containing the items for the knapsack.

Equations (1)-(5) can be rewritten as follows:

$$\text{maximize} \quad \sum_{i=1}^N \sum_{j=1}^M v_i \cdot x_{i,j} \quad (9)$$

$$\text{subject to} \quad \sum_{i=1}^N \sum_{j=1}^M e_{i,j} \cdot x_{i,j} \leq E_{max} \quad (10)$$

$$\sum_{i=1}^N \sum_{j=1}^M t_{i,j} \cdot x_{i,j} \leq D \quad (11)$$

$$\sum_{j=1}^M x_{i,j} \leq 1 \quad (12)$$

$$x_{i,j} \in \{0, 1\} \quad (13)$$

$$\forall i \in \{1, 2, \dots, N\}, \forall j \in \{1, 2, \dots, M\}$$

Thus, there are $N \cdot M$ variables (the vector $x_{i,j}$) and $N + 2$ constraints. The solution is the column vector x with $N \cdot M$ elements in which $x_{i,j} = 1$ means that task i is selected and runs at speed level j . (10) enforces the energy constraint, (11) is the timing constraint and (12) consists of N inequalities which ensure that each task is selected at most once in the solution.

While many algorithms exist for approximating the *0-1 Multidimensional Knapsack problem* (both for real and integer coefficients) [28], we take advantage in our approach of the fact that each of the last N rows of matrix A have exactly N coefficients equal to 1, while the other coefficients $((N-1) \cdot M)$ are zero. Similarly, in vector b the last N values (out of a total of $N+2$) are all equal to 1. This allows a running time which is faster than even comparison-based sorting on the same input size ($N \cdot M$), yet leading to a very good approximation of the optimal solution.

In the *0-1 Bidimensional Knapsack problem* matrix A has only 2 rows (constraints):

$$\text{maximize} \quad \sum_{i=1}^N c_i \cdot x_i \quad (14)$$

$$\text{subject to} \quad \sum_{i=1}^N a_{1i} \cdot x_i \leq b_1 \quad (15)$$

$$\sum_{i=1}^N a_{2i} \cdot x_i \leq b_2 \quad (16)$$

$$x_i \in \{0, 1\} \quad (17)$$

We show next that the problem described by equations (9)-(13) is harder than than the *0-1 Bidimensional Knapsack problem* by showing the transformation from the 0-1 bidimensional knapsack of size N to a problem instance for (9)-(13) of size $N \cdot M$.

For each variable x_i we add $M-1$ variables $y_{ij}, \forall j \in \{1, 2, \dots, M-1\}$. The maximizing part $\sum_{i=1}^N c_i \cdot x_i$ is transformed to $\sum_{i=1}^N (c_i \cdot x_i + \sum_{j=1}^{M-1} c_i \cdot y_{ij})$. The first constraint is transformed from $\sum_{i=1}^N a_{1i} \cdot x_i \leq b_1$ to $\sum_{i=1}^N (a_{1i} \cdot x_i + \sum_{j=1}^{M-1} k \cdot y_{ij}) \leq b_1$, where k is chosen to be higher than b_1 . The second constraint is left unchanged and the new N constraints are added: $x_i + \sum_{j=1}^{M-1} y_{ij} \leq 1, \forall i \in \{1, 2, \dots, N\}$.

Observe that it is never possible to choose an item y_{ij} in the knapsack as the first constraint would be violated. Thus, the solution of the transformed problem must be the same as the bidimensional knapsack solution. This way the bidimensional knapsack was transformed to an instance of (9)-(13). Knowing that the *0-1 Bidimensional Knapsack problem* is NP-hard (by a transformation from the simple Knapsack problem), we conclude that (9)-(13) is also NP-hard.