Maximizing the system value while satisfying time and energy constraints

C. A. Rusu
R. Melhem
D. Mossé

Embedded devices designed for various real-time applications typically have three constraints that must be addressed: ***energy*****,** ***deadlines*****,** *and* ***reward****. These constraints play important roles in the next generation of embedded systems, since they provide users with a variety of quality-of-service (QoS) tradeoffs. We propose a QoS model in which applications may have several versions, each with different time and energy requirements, while providing different levels of accuracy (reward). An optimal scheme would allow the device to run the most critical and valuable versions of applications without depleting the energy source, while still meeting all deadlines. A solution is presented for frame-based and periodic task sets. Three algorithms are devised 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. Overloaded systems naturally lend themselves to scenarios in which only the most critical applications can be executed. If some applications cannot be executed in a timely fashion, the system must select the applications that will maximize the overall reward (a value/reward is assigned to each application), such that all selected applications will execute within their respective deadlines and without exceeding the energy budget.

A solution to this problem for frame-based task sets was presented in [1]. Two algorithms were devised that select the most valuable applications, given the timing and energy constraints. This work includes a review of these algorithms, with new experimental results obtained for a different system. A third algorithm is then proposed that extends the preliminary work in [1] in three directions: periodic tasks, optional/mandatory tasks, and task versions.

Version programming enhances the opportunity for QoS tradeoffs. An example of version programming comes from satellite-based signal processing [2]. Four different algorithms (least mean square, maximum likelihood, software trigger, matched filter) with running times ranging from microseconds to hundreds of milliseconds and energy consumptions from microjoules to joules provide different levels of accuracy. Another example is automatic target recognition (ATR), in which task values, running times, and energy requirements are roughly proportional to the number of targets detected [3]. Task versions can result from different algorithms, as well as from the same application with different input arguments, such as encoding/decoding at different rates, low/high-quality compression schemes, and low/high-resolution image processing.

Note: A preliminary version of this work appeared in the Proceedings of the 23rd IEEE Real-time Systems Symposium (RTSS'02), Austin, TX, December 2002 [1].

0018-8646/03/$5.00 © 2003 IBM

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

Note that this problem differs from simply minimizing power consumption due to the extra constraints considered, namely deadlines and task values. Clearly, minimizing the energy consumption of applications is useful, but it does not consider the value/reward characteristics of different applications. Simultaneous consideration of *reward*, *energy*, and *deadlines* is important because it allows system designers to determine the most important components of their system, or allows them to emphasize one subset of the system over another in a dynamic fashion. An example of such flexibility is when one decides to maximize mission lifetime (the time the system is functional) 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 single-version and multiple-version task models and defines the problem. In Section 3 we review the two algorithms in [1] for the single-version task model, followed by new experimental results obtained through simulation. Section 4 describes and evaluates a new algorithm for the multiple-version task model. Section 5 concludes the paper.

Related work

For decades, the issue of assignment of CPU cycles to different tasks has been studied through scheduling and operations research. In the mid-1980s, researchers began considering the tradeoff between time and other metrics, such as value/reward [4]. In the late 1990s, researchers began to study a case that was similar, but focused on the tradeoff between energy and time [5]. Below we describe representative work in these two fields. However, none of this work addressed the general framework with the three types of constraints we consider here: energy, deadline, and reward/value.

Rewards and real time

The Imprecise Computation (IC) [6, 7] and Increased Reward with Increased Service (IRIS) [8, 9] models were proposed to enhance utilization of resources 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 more time the CPU is allocated to the process, the better the quality of the result. Liu et al. proposed several efficient algorithms to solve the scheduling problem of aperiodic tasks [6, 7]. A common assumption in these studies is that the quality of the results produced is a linear function of the precision; more general precision functions are not usually addressed.

An alternative 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 nondecreasing concave reward function is associated with the execution time of each task. Dey et al. addressed the problem of maximizing the total reward in a system of aperiodic tasks and presented an optimal solution for static task sets, as well as two extensions that include mandatory parts and policies for dynamic task arrivals[1] [8, 10]. Aydin et al. presented an optimal algorithm assuming concave reward functions and periodic real-time applications [11]. Both IC and IRIS focus on linear and concave (for example, logarithmic) functions representing applications such as image and speech processing [12–14] or multimedia applications [15]. The case of real applications with no reward for partial executions or step functions has been shown in [6] to be NP-complete.[2] Furthermore, the reward-based scheduling problem for convex reward functions is NP-hard[3] [11].

Rajkumar et al. proposed a QoS-based resource allocation model (QRAM) for periodic applications [15]. The reward functions were given 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 could be either dependent or independent. In [16], the QRAM work was 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) were 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 [5, 17] can be achieved at the expense of only 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. provided a static off-line scheduling algorithm [5], assuming aperiodic tasks and worst-case

[1] Dynamic task arrivals — Tasks that are presented to the system dynamically, without *a priori* knowledge of task arrival times.
[2] Nondeterministic Polynomial-time complete — A set or property of computational decision problems which is a subset of NP (i.e., can be solved by a nondeterministic Turing machine in polynomial time), with the additional property that it is also NP-hard.
[3] A problem is NP-hard if solving it in polynomial time would make it possible to solve all problems in class NP in polynomial time.

execution times (WCET). Hong et al. proposed heuristics for on-line scheduling of aperiodic tasks while not affecting the feasibility of periodic requests [18]. The same authors investigated nonpreemptive power-aware scheduling [19] and examined the effects of upper bounds on the voltage change rate [17]. Shin and Choi explored the CPU slowdown that occurs when there is a single task eligible for execution [20]. Lorch and Smith investigated VVS in the context of soft deadlines [21]. Cyclic and EDF scheduling of periodic hard real-time tasks on systems with two (discrete) voltage levels were investigated by Krishna and Lee [22]. Aydin et al. provided the static solution for the general periodic model in which tasks have potentially different power characteristics [23]. Since real-time applications exhibit a large variation in actual execution times (Ernst and Ye [24]) and WCET is too pessimistic, much research was directed at dynamic slack-management techniques [25–28].

Aydin et al. proved 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 that all of the tasks run at the maximum speed [25].

In this work we address the problem of maximizing the rewards, assuming the VVS framework and a limited energy budget for real-time tasks. 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 at which these selected tasks should run so that the total reward of the system is maximized and the timing and energy constraints are met.

Recently, similar research combined the time, energy, and reward constraints for the case of IRIS tasks (Kang et al. [29]). An algorithm was developed to maximize the system value through an energy-aware allocation of resources. However, the task model in [29] does not include voltage or frequency scaling.

2. Task model

Two task models with their corresponding problem definitions are presented. To simplify the problem, we first assume that tasks are frame-based, meaning that all task periods are identical and all task deadlines are equal to their period. At the end of the section we show how the periodic-task case, with individual deadlines for each task, results in an equivalent problem formulation.

The common deadline/period (also known as frame length) is denoted by D. There are N available periodic tasks in the system, all ready at time zero. A frame consists of a subset of tasks that are selected for execution. The execution of the frame is to be repeated.

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), $\{f_1, f_2, \cdots, f_M\}$. 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 that the overhead of $\min(M, N)$ speed changes is negligible compared to the frame length D or that it has already been subtracted from D.

Optional single-version task model

The task set is denoted by $\mathbf{T} = \{T_1, T_2, \cdots, T_N\}$. It is not required that all tasks be scheduled; however, a task cannot be selected more than once during a frame. We 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 r_i (also called the 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. The ultimate goal is to find a subset of tasks $S \subseteq \{1, 2, \cdots, N\}$ that maximizes the system value (reward) $\Sigma_{i \in S} r_i$. For all tasks $i \in S$, the speed level $s_i \in \{1, 2, \cdots, M\}$ 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.
- 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, $\forall i \in S$ in order to

$$\text{maximize} \quad \sum_{i \in S} r_i \tag{1}$$

$$\text{subject to} \quad \sum_{i \in s} t_{i,s_i} \le D, \tag{2}$$

$$\sum_{i \in s} e_{i,s_i} \le E_{\max}, \tag{3}$$

$$S \subseteq \{1, 2, \cdots, N\}, \tag{4}$$

$$s_i \in \{1, 2, \cdots, M\}. \tag{5}$$

Inequality (2) guarantees that the timing constraint is satisfied, and inequality (3) guarantees that the energy budget is not exceeded.

Multiple-version task model

In this model each task has several versions, each with different rewards, time, and energy requirements. For example, one version can execute faster and requires less energy, at the expense of producing less accurate/complete/valuable results. For simplicity we assume the same number of versions, V, for each task, although the algorithms proposed can handle different numbers of versions as well as different numbers of speed levels for each task. Within each frame, *exactly one* version of each task must be scheduled.

The version k of task i is denoted by T_i^k. The execution time and energy requirement of version k of task i running at speed level j are denoted by $t_{i,j}^k$ and $e_{i,j}^k$, respectively. Associated with version k of task i there is a version value or reward, r_i^k. The goal is to determine for each task which version to execute and the speed level at which to do so in order to maximize the system value (i.e., the sum of rewards for all task versions selected for execution). The same two major constraints apply: the timing constraint in the form of the deadline, D, and the energy constraint imposed by the amount of energy available in the system, $E_{\max}$.

Thus, the problem is to determine for each task i its version v_i and speed level s_i in order to

$$\text{maximize} \quad \sum_{i \in S} r_i^{v_i} \tag{6}$$

$$\text{subject to} \quad \sum_{i \in s} t_{i,s_i}^{v_i} \le D, \tag{7}$$

$$\sum_{i \in s} e_{i,s_i}^{v_i} \le E_{\max}, \tag{8}$$

$$v_i \in \{1, 2, \cdots, V\}, \tag{9}$$

$$s_i \in \{1, 2, \cdots, M\}. \tag{10}$$

If power is also a constraint, task versions at high speed levels exceeding the power budget are simply removed from further consideration. The algorithms proposed in Sections 3 and 4 can handle a different number of speed levels and versions for each task. Thus, the search is limited to those versions and speed levels that are feasible from the point of view of peak power.

As shown in the Appendix, the problems defined by (1)–(5) and (6)–(10) are NP-hard. Therefore, we relax the maximization objectives in (1) and (6) and look for solutions that approximate the optimal solution.

Periodic tasks

We next present the problem definition for multiple-version periodic tasks. We denote the deadline of task T_i by D_i and the least common multiple of all task deadlines (hyperperiod) by T_{LCM}. Assuming that the maximum energy is associated with T_{LCM}, the formulation of the multiple-version periodic task problem is

$$\text{maximize} \quad \sum_{i \in S} r_i^{v_i} \frac{T_{\text{LCM}}}{D_i} \tag{11}$$

$$\text{subject to} \quad \sum_{i \in s} \frac{t_{i,s_i}^{v_i}}{D_i} \le 1, \tag{12}$$

$$\sum_{i \in s} e_{i,s_i}^{v_i} \frac{T_{\text{LCM}}}{D_i} \le E_{\max}, \tag{13}$$

$$v_i \in \{1, 2, \cdots, V\}, \tag{14}$$

$$s_i \in \{1, 2, \cdots, M\}. \tag{15}$$

The total reward for the hyperperiod is the sum of rewards for all task instances (11). Similarly, the energy consumption of all task instances is accounted for in (13). The timing constraint in (12) assumes Earliest Deadline First (EDF)[4] scheduling. A different utilization formula can be used with different schedulers, such as Rate Monotonic Scheduling (RMS).[5] Observe that problems (6)–(10) and (11)–(15) are equivalent. The periodic task set corresponds to a frame of length T_{LCM}, in which the time, energy, and value of each task T_i are multiplied by (T_{LCM}/D_i).

The algorithms presented in the next two sections assume frame-based task sets, as described by Equations (1)–(5) and (6)–(10). Clearly, the same algorithms can be used for periodic tasks as well.

3. Optional single version

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 energy, deadline, and reward constraints. 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 led to poor utilization of one of the resources (energy or time) and poor system values. Algorithms that dynamically modify the schedule on the basis of resource usage (while still considering task values) turned out to be 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 three constraints at once. Two algorithms were proposed in [1] that closely approximate the optimal solution; these

[4] Earliest Deadline First — Preemptive real-time scheduling algorithm in which tasks with the earliest deadline have priority.
[5] Rate Monotonic Scheduling — Preemptive real-time scheduler in which tasks have fixed priorities according to their period. Tasks are executed in the order of their priority.

algorithms are reviewed in the next sections, followed by a quantitative reevaluation for a different system.

We assume that tables exist that store the task values $r(i)$, running times $t(i,j)$ and energy requirements $e(i,j)$ for all tasks $i \in \{1, 2, \cdots, N\}$ and all speed levels $j \in \{1, 2, \cdots, M\}$. The algorithms are based on adapting the schedule by adding and dropping tasks until all of 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*] with no task considered yet [*considered(i)=false*]. The set of selected tasks (initially empty) is defined as $S=\{i \mid selected(i)=true\}$. Two variables, *time* and *energy*, store the total running time of the schedule [$time = \Sigma_{i \in S}\, t(i, s_i)$] and the total energy consumed [$energy = \Sigma_{i \in S}\, e(i, s_i)$] and are initialized to zero. R stores the system value for the current schedule [$R = \Sigma_{i \in S}\, r(i)$] and SR stores the system value, that is, the largest value of R encountered so far. Finally, an array, *speed(i)*, of size N stores the speeds of all tasks.

The REW-Pack algorithm

The flowchart of the REW-Pack algorithm is presented in **Figure 1**. The three major components (add task, increase speed, and drop task) 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 has not been considered before [*considered(i)=false*].
- The current schedule is feasible ($time \le D$).
- Adding the task to the current schedule at the minimum speed does not cause the energy budget to be exceeded [$energy + e(i, 1) \le E_{max}$].
- Among all tasks T_i that satisfy the above criteria, the one that has the largest ratio,

$$\frac{r(i)}{t(i, 1)e(i, 1)},$$

is selected.

A new task is always added if possible. The task added must have a good (large) value, a reasonable (small) running time, and 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.

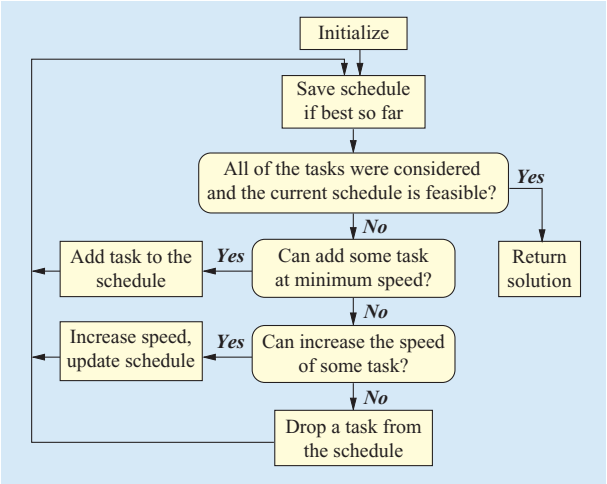

Figure 1

Flowchart of the REW-Pack algorithm. Reprinted with permission from [1]; © 2002 IEEE.

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 speed of the task. Also observe that exceeding the deadline is allowed. We noticed during experiments that without this enhancement, some tasks were prematurely removed from consideration by the scheduler, affecting the accuracy of the solution. However, for similar reasons, we do not allow the energy budget to be exceeded.

Increase task speed

If no task can be added to the schedule, the algorithm *packs* tasks to make room for other, not-yet-selected tasks (the term *packing* means increasing 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 conditions:

- It is selected in the current schedule [*selected(i)=true*].
- It is not running at the maximum speed ($s_i \ne M$).
- Increasing the speed of the task to the next higher level does not cause the energy budget to be exceeded [$energy + e(i, s_i + 1) - e(i, s_i) \le E_{max}$].
- Among all selected tasks T_i, it has the highest ratio $(\Delta t/\Delta E)$, where $\Delta t = t(i, s_i) - t(i, s_i + 1)$ and $\Delta E = e(i, s_i + 1) - e(i, s_i)$.

```
1. Initialize: selected(i)=false;
   considered(i)=false ∀i∈{1, 2, ···, N};
   energy=0; time=0; SR=0; R=0
2. If time ≤ D and SR<R
   a. sol_selected(i)=selected(i);
      sol_speed(i)=speed(i),
      ∀i ∈ {1, 2, ···, N}
   b. SR=R
3. If (∃i, considered(i)==false) or
   (time>D) do
   a. i=add_task()
   b. If i ≠ −1
      i. selected(i)=true;
         considered(i)=true;
         energy=energy+e(i,1)
         speed(i)=1; time=time+t(i,1);
         R=R+r(i)
     ii. Go to step 2
   c. i=increase_speed()
   d. If i ≠ −1
      i. 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
     ii. Go to step 2
   e. i=drop_task()
   f. energy=energy-e(i,speed(i));
      time=time-t(i,speed(i); R=R-
      r(i); selected(i)=false
   g. Go to step 2
4. Return solution (sol_selected,
   sol_speed, SR)
```

(a)

```
1. Initialize: selected(i)=false;
   considered(i)=false ∀i∈{1, 2, ···, N};
   energy=0; time=0; SR=0; R=0
2. If energy ≤ E_max and SR<R
   a. sol_selected(i)=selected(i);
      sol_speed(i)=speed(i),
      ∀i ∈ {1, 2, ···, N}
   b. SR=R
3. If (∃i, considered(i)==false) or
   (energy>E_max) do
   a. i=add_task()
   b. If i ≠ −1
      i. selected(i)=true;
         considered(i)=true;
         energy=energy+e(i,M);
         speed(i)=M;
         time=time+t(i,M); R=R+r(i)
     ii. Go to step 2
   c. i=decrease_speed()
   d. If i ≠ −1
      i. 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
     ii. Go to step 2
   e. i=drop_task()
   f. energy=energy-e(i,speed(i));
      time=time-t(i,speed(i); R=R-
      r(i); selected(i)=false
   g. Go to step 2
4. Return solution (sol_selected,
   sol_speed, SR)
```

(b)

Figure 2

(a) REW-Pack algorithm. (b) REW-Unpack algorithm. Both parts reprinted with permission from [1]; © 2002 IEEE.

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 because the total reward is not changed by the packing operation. Interestingly, we obtained poor results when we used the same metric for packing as we did for task selection—that is, increasing the speed of the task with the smallest ratio,

$$\frac{r(i)}{t(i, s_i)e(i, s_i)}.$$

Drop a task

If the previous two steps fail, a task is eliminated from the current schedule. The task that is dropped must satisfy the following conditions:

- It is selected in the current schedule [*selected(i)=true*].
- Among all selected tasks T_i, it has the smallest ratio,

$$\frac{r(i)}{t(i, s_i)e(i, s_i)}.$$

When it is necessary to drop a task, the task with the worst (i.e., smallest) metric $[r(i)/t(i, s_i)e(i, s_i)]$ is dropped. Task values must 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 REW-Pack algorithm is shown in **Figure 2(a)**; *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 $O(MN \log N)$.

The REW-Unpack algorithm

The idea behind the REW-Unpack algorithm is basically the same as that for 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 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 $(\Delta E/\Delta t)$ is selected). As in REW-Pack, exceeding the energy budget is allowed, while exceeding the deadline is not. The algorithm is shown in **Figure 2(b)**.

Another interesting problem is that of minimizing the energy given a desired system value. The REW-Pack algorithm can solve this problem by pruning the search for a solution when the total value exceeds the desired system value.

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. We define the absolute error for each of the two algorithms to be

$$\frac{SR_{\mathrm{OPT}} - SR}{SR_{\mathrm{OPT}}},$$

where SR represents the system value (reward) resulting from the algorithm and SR_{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 following parameters are used in our simulations:

- N = number of tasks.
- M = number of speed levels.
- $t_{i,j}$, $e_{i,j}$ = time and energy requirements.

Table 1 PowerPC 405LP speeds, voltages, and power ranges.

Speed level	*Speed* (MHz)	*Voltage* (V)	*Power range* (mW)
1	100	1.0	46–82
2	200	1.4	154–300
3	266	1.7	307–630
4	333	1.9	429–881

- D = deadline.
- $E_{\max}$ = available energy.
- r_i = task values (rewards).
- N_{r} = number of runs from which we obtain averages.

The maximum deadline, Max_{D}, is defined as $Max_{\mathrm{D}} = \sum_{i=1}^{N} t_{i,1}$, that is, the total execution time of the tasks at minimum speed. The maximum energy, Max_{E}, is defined as $Max_{\mathrm{E}} = \sum_{i=1}^{N} e_{i,M}$, that is, the total energy requirement for all tasks if they are running at maximum speed. Clearly, if $D \geq Max_{\mathrm{D}}$, the timing constraint cannot be violated. Similarly, if $E_{\max} \geq Max_{\mathrm{E}}$, 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 $D = \alpha Max_{\mathrm{D}}$, and the energy was generated by $E_{\max} = \beta Max_{\mathrm{E}}$, where $\alpha, \beta \in [0, 1]$.

We simulated the IBM PowerPC* 405LP processor as presented in **Table 1**. The 405LP power ranges were obtained by experiments we performed at the IBM Austin Research Laboratory on a 405LP evaluation board (designed for power measurements). We measured the 405LP core power for a set of artificial benchmarks at four frequency/voltage settings: 33 MHz/1.0 V, 100 MHz/1.0 V, 266 MHz/1.8 V, and 333 MHz/1.9 V. The combinations 33 MHz/1.0 V and 266 MHz/1.8 V were identified to be energy-inefficient operating points, and we removed them from the model (for example, our benchmarks consume a little more energy at 33 MHz/1.0 V than they do at 100 MHz/1.0 V). The 200 MHz/1.4 V and 266 MHz/1.7 V settings in Table 1 are known to be feasible but were not actually measured. The power range for these two frequency/voltage settings was extrapolated on the basis of voltage, frequency, and data from the measured operating points. Minimum and maximum power consumption values at each operating point are indicated in the last column of the table. Similar results are obtained for other power models, such as XScale [1].

For each task, the 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 $t_{i,j} = t_{i,1}(f_1/f_j)$; thus, the running time is inversely

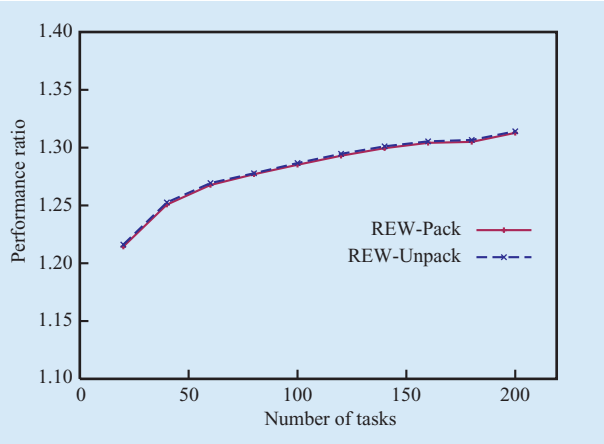

Figure 3

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

proportional to the speed. The power was computed as $P_{i,j} = P_j^{\min} + a_i(P_j^{\max} - P_j^{\min})$, where $[P_j^{\min}, P_j^{\max}]$ is the power range of speed level j and $a_i \in [0, 1]$. The energy requirement $e_{i,j}$ is then computed as $e_{i,j} = P_{i,j}t_{i,j}$, that is, the power multiplied by 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 values into consideration and randomly selects which tasks to add/drop/pack from the subset of tasks satisfying the add/drop/pack criteria. For each simulation, α and β were randomly generated in the range [0.1, 0.3]. Task sets with 20 to 200 tasks were simulated, and 1000 experiments were averaged for each point in the graphs. The performance ratio shown in **Figure 3** 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 for each particular simulation, they consistently outperformed the simplified REW-Pack, which, in turn, outperforms the other heuristics we tried for solving (1)–(5).

Figure 4 shows the average absolute error of the algorithms as a function of the available energy. Task sets with $N = 10$ tasks were simulated for tight deadlines ($\alpha = 0.2$) and for more relaxed deadlines ($\alpha = 0.3$ and $\alpha = 0.4$). The average for 100 simulations was computed for each point. The same averages hold for higher numbers of simulations. The maximum error for each point is typically 10–30%. Experiments show that as α increases beyond 0.5, 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 for 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

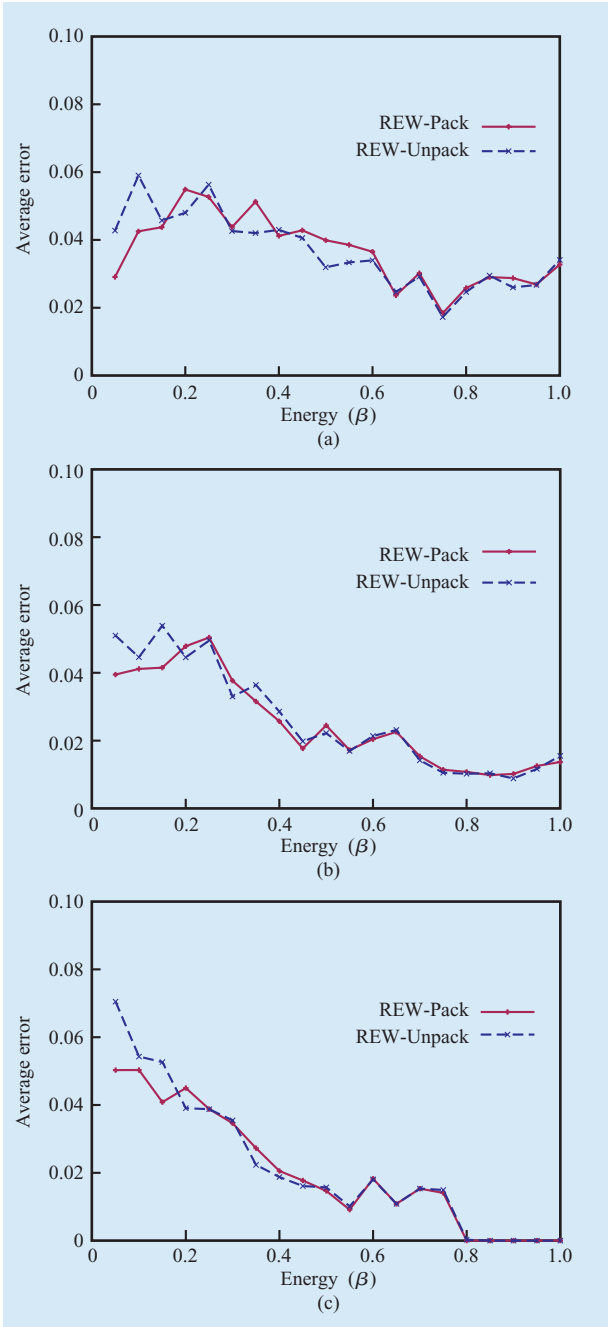

Figure 4

Average absolute error of the algorithms as a function of β (available energy) for ten tasks: (a) $\alpha = 0.2$; (b) $\alpha = 0.3$; (c) $\alpha = 0.4$.

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 usually select the same tasks in the end. Also, the tasks selected by the algorithms are usually the same as those selected by the optimal. In fact, for each point in the graphs, the algorithms were equal to the optimal at least 31% of the time (31% was obtained for REW-Pack at $\alpha = 0.2$ and $\beta = 0.25$).

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 in which D or E_{max} are too large to be used entirely given the other constraint), and both algorithms return schedules that tend to use to the maximum 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 5**, which shows results for simulated task sets with five to 14 tasks and $\alpha = 0.4$ and $\beta = 0.4$, we can see this trend. In the figure, each point is the average error of 100 runs.

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 $k_i \in \{1, 2, \cdots, M\}$ 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 $SR_{OPT} = \sum_{i=1}^{N} r_i$. We ran 1000 simulations on task sets with 50, 100, and 200 tasks. We do not show a graph for the results, because both of our heuristic algorithms returned the optimal solution in all 1000 simulation runs.

4. Multiple versions

For multiple versions, the MV-Pack algorithm is proposed to solve Equations (6)–(10). The algorithm, similar in many respects to REW-Pack, is described and evaluated in this section.

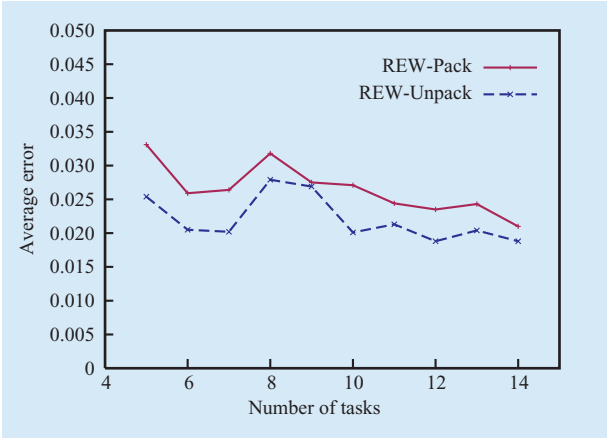

Figure 5

Average absolute error of the algorithms as a function of N ($\alpha = 0.4$, $\beta = 0.4$).

The MV-Pack algorithm

The flowchart of MV-Pack is shown in **Figure 6(a)**. The algorithm has three major components: add task, increase speed, and increase version. The first two components are identical to those of REW-Pack. However, since the multiple-version task model requires that each task in the schedule be selected, tasks are never dropped.

The algorithm begins with an empty schedule. A new task is added if possible, always at the first (smallest) speed level and version (we assume that task versions are sorted by their reward, with the first version having the smallest reward). If the deadline is exceeded, tasks are packed to make room for other tasks. When all of the tasks in the schedule have been selected, a minimum-reward solution is found; otherwise, failure is returned.

Next, while the remaining energy allows it, a better schedule (higher reward) is searched by increasing the version of some task. The third component of the algorithm (increase version) selects the task to move to its next higher version. The old version is removed from the schedule, while the new version is added at the minimum speed. Tasks are then packed, if necessary, until either a solution with the new version is found or the energy is exceeded, in which case the current solution is returned.

The process of adding and packing tasks was described in detail in the previous section. The last component of MV-Pack is described next. The task i that is selected to move to the next higher reward version satisfies the following criteria:

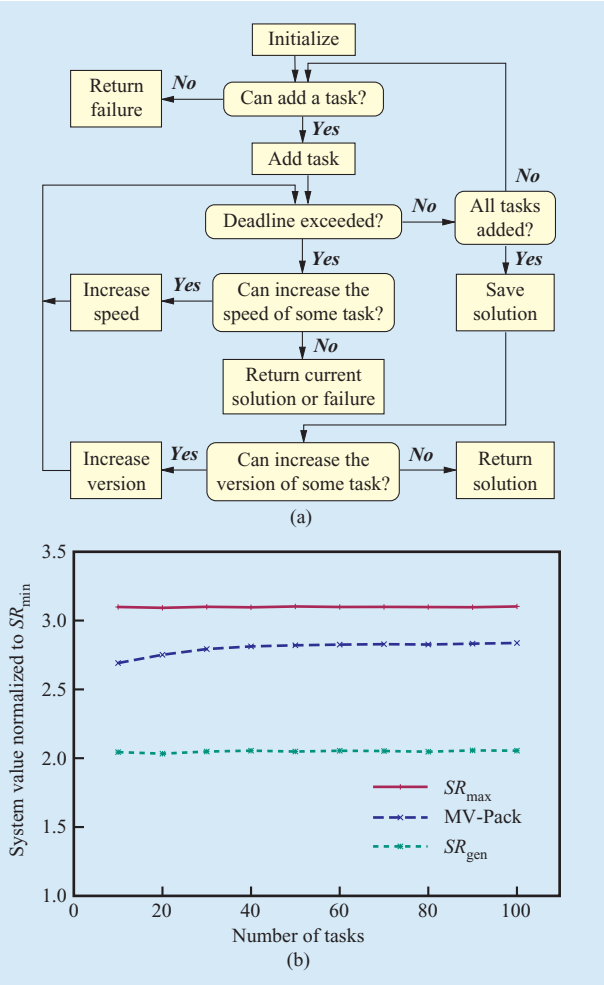

Figure 6

(a) MV-Pack flowchart; (b) evaluation of MV-Pack.

- It is not running at the highest version ($v_i < V$).
- Replacing the current version of the task with the next higher version at the first speed level does not cause the energy budget to be exceeded ($energy + e_{i,1}^{v_i+1} - e_{i,s_i}^{v_i} \le E_{max}$).
- Among all the tasks that are not running at their highest version, the next version at minimum speed has the largest reward per unit time and energy. That is, we select task i that maximizes

$$\frac{r_i^{v_i+1}}{t_{i,1}^{v_i+1} e_{i,1}^{v_i+1}}.$$

The complexity of MV-Pack can be analyzed as follows. Each task is added at most once and its version can be increased at most $V - 1$ times. We can increase the speed of each task at most $(M - 1)V$ times. With appropriate data structures, determining which task to choose takes $\log N$ time for all functions (add task, increase speed, and increase version). Thus, the complexity of the algorithm is $O(MVN \log N)$.

Experimental results

We simulated the 405LP processor as presented in Table 1. For each task, the execution time of the first version at minimum speed $t_{i,1}^1$ was randomly generated in the range [10, 100]. For the remaining versions, the running time at the first speed level was generated by the formula $t_{i,1}^k = t_{i,1}^{k-1} + \Delta_i^k$, where $\Delta_i^k \in [0.2t_{i,1}^1, 1.2t_{i,1}^1]$ was randomly generated for each task version. Next, $t_{i,j}^k$ was computed for all versions and all speed levels, inversely proportional to the speed $[t_{i,j}^k = t_{i,1}^k(f_1/f_j)]$.

The energy requirements $e_{i,j}^k$ were generated as described in the single-version experiments. The activity coefficients a_i are different for each task and identical for all versions of the same task. Task values of the first versions r_i^1 were generated randomly in the range [10, 100]. For the higher versions (a number of $V = 4$ versions were used for each task), task rewards were generated according to the formula $r_i^k = r_i^{k-1} + \delta_i^k$, where $\delta_i^k \in [0.2r_i^1, 1.2r_i^1]$ was randomly generated for each task version. Thus, observe that each version requires more time and more energy than the previous versions, but gives a higher reward. The deadline D and maximum energy E_{max} are respectively generated by the formulas $D = \sum_{i=1}^N t_{i,s_i}^{v_i}$ and $E_{max} = \sum_{i=1}^N e_{i,s_i}^{v_i}$, where $s_i \in \{1, 2, \cdots, M\}$ and $v_i \in \{1, 2, \cdots, V\}$ are randomly generated for each task $i \in \{1, 2, \cdots, N\}$. We denote by SR_{min} the minimum reward that can be achieved for a given task set, $SR_{min} = \sum_{i=1}^N r_i^1$. Similarly, SR_{max} denotes the maximum reward that can be achieved, $SR_{max} = \sum_{i=1}^N r_i^V$. Observe that if each task i runs at the version v_i and the speed level s_i used to generate D and E_{max}, the energy and deadlines are not exceeded, and the system reward is $SR_{gen} = \sum_{i=1}^N r_i^{v_i}$.

Since it is impractical to compute the optimal solution, we compare the performance of MV-Pack with SR_{min}, SR_{max}, and SR_{gen}. **Figure 6(b)** shows the comparison for task sets of 10 to 100 tasks, where SR_{gen}, SR_{max}, and the reward returned by the algorithm are normalized to SR_{min}. Each point is the average of 1000 simulation runs. In all experiments, MV-Pack returned a system value higher than SR_{gen} and close to SR_{max}. Note that SR_{max} is an upper bound on the optimal solution, not the optimal solution itself. For most graph points, MV-Pack used more than 99% of the available energy; the smallest value is 96%. Similarly for the available time, the smallest usage was 98%.

The system value can be improved even more with the following enhancement: If there is not enough energy to pack tasks within the deadline, the task that caused the

energy problem when increasing its version number is removed from further consideration, and the schedule is restored to its state just prior to the attempt to increase the version number of the offending task. The algorithm continues then as usual by selecting a task to increase its version from the remaining set of tasks. The upper bound on the running time becomes $O(MVN^2 \log N)$, although in practice the running time does not increase significantly. The enhanced MV-Pack returns slightly higher rewards in 24% of the simulations with ten tasks and in 57% of the simulations with 100 tasks. However, the increase in the average normalized reward is almost unnoticeable.

The enhanced MV-Pack algorithm can also handle the optional single-version model. The original task set is modified in the following way: For each task we artificially add to the single version a second version with zero reward and zero energy and time requirements. We call this added version the zero version. A task selected in the final schedule at its zero version is equivalent to a task not selected for execution in the optional single-version model. **Figure 7** compares the REW-Pack algorithm applied to a single-version task set and the MV-Pack algorithm applied to the same task set enhanced with zero versions. The system value is normalized to SR_{gen}, as $SR_{min} = 0$ due to the zero versions. MV-Pack is slightly better for all points in the graph, at the cost of a higher execution time. In practice, MV-Pack (which has a higher theoretical upper bound) takes 25% to 70% longer than REW-Pack, both algorithms running in less than a millisecond even for 100 tasks. For 20 tasks, MV-Pack is better in 2.5% of the experiments and REW-Pack is better in 0.4% of the experiments, the algorithms returning equal system values in the rest of the simulations. As the number of tasks increases, the dominance of MV-Pack is more evident: For 100 tasks, MV-Pack is slightly better in 12% of the experiments, the algorithms being equal in the rest of the simulations.

5. Conclusions

We have presented two algorithms for the problem of maximizing the system value given time and energy constraints in a single-version task set environment. A third algorithm has been presented for the same problem and multiple-version task sets. The goal is to determine which tasks (or task versions) to execute and the speeds at which 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 (or task versions) 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,

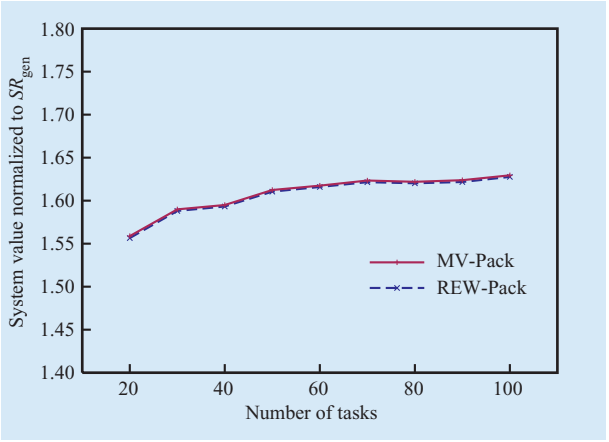

Figure 7

Comparison of the enhanced MV-Pack algorithm with REW-Pack.

continuous speeds and/or continuous reward functions (increased reward with increased service) are usually assumed. In this work we have 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. Where possible we show by simulation that the proposed algorithms closely approximate the optimal. The worst-case time complexity of the single-version 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. For multiple versions, the running time is $O(MVN \log N)$, where V is the number of versions. A small running time allows a scheduler to quickly adapt to changes in the system (e.g., 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). The number of versions in version programming is also typically very small.

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

In Section 2 we claimed that both problem formulations, as described respectively by Equations (1)–(5) and (6)–(10), are NP-hard. We next present the proof for the single-version problem, followed by a sketch of the proof for multiple versions.

First we show how the single-version problem can be transformed to a special case of the 0–1 multidimensional knapsack problem [30]. Then we show that the single-version problem formulation is harder than the 0–1

bidimensional knapsack problem, which is known to be NP-hard.

The 0–1 multidimensional knapsack problem has the following formulation:

$$\text{maximize} \quad cx \tag{16}$$

$$\text{subject to} \quad Ax \le b, \tag{17}$$

$$x_i \in \{0, 1\}, \tag{18}$$

where $x = [x_1, x_2, \cdots, x_n]^t$ is a column vector of 0–1 variables, $c = [c_1, c_2, \cdots, c_n]$ is a row vector of integers, A is a matrix with m rows (constraints) and n columns with integer values, and $b = [b_1, b_2, \cdots, b_m]^t$ 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:

$$\text{maximize} \quad \sum_{i=1}^{N} \sum_{j=1}^{M} r_i x_{i,j} \tag{19}$$

$$\text{subject to} \quad \sum_{i=1}^{N} \sum_{j=1}^{M} e_{i,j} x_{i,j} \le E_{\max}, \tag{20}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M} t_{i,j} x_{i,j} \le D, \tag{21}$$

$$\sum_{j=1}^{M} x_{i,j} \le 1, \tag{22}$$

$$x_{i,j} \in \{0, 1\}, \tag{23}$$

$$\forall\, i \in \{1, 2, \cdots, N\}, \qquad \forall\, j \in \{1, 2, \cdots, M\}.$$

Thus, there are NM variables (the vector $x_{i,j}$) and $N + 2$ constraints. The solution is the column vector x with NM elements in which $x_{i,j} = 1$ means that task i is selected and runs at speed level j. Equation (20) enforces the energy constraint, (21) is the timing constraint, and (22) 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 (for both real and integer coefficients) [31], we take advantage in our approach of the fact that each of the last N rows of matrix A has exactly N coefficients equal to 1, while the other coefficients $[(N - 1)M]$ are 0. 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 (NM), yet leading to a very good approximation of the optimal solution.

In the 0–1 bidimensional knapsack problem, matrix A has only two rows (constraints):

$$\text{maximize} \quad \sum_{i=1}^{N} c_i x_i \tag{24}$$

$$\text{subject to} \quad \sum_{i=1}^{N} a_{1i} x_i \le b_1, \tag{25}$$

$$\sum_{i=1}^{N} a_{2i} x_i \le b_2, \tag{26}$$

$$x_i \in \{0, 1\}. \tag{27}$$

We show next that the problem described by Equations (19)–(23) is harder than the 0–1 bidimensional knapsack problem by showing the transformation from the 0–1 bidimensional knapsack problem of size N to a problem instance for (19)–(23) of size NM.

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

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 problem was transformed to an instance of (19)–(23). Knowing that the 0–1 bidimensional knapsack problem is NP-hard (by a transformation from the simple knapsack problem), we conclude that (19)–(23) is also NP-hard.

For multiple versions, Equations (6)–(10) can be rewritten with a similar transformation into exactly the formulation of the 0–1 multiple-choice bidimensional knapsack problem, which is known to be NP-hard.

Acknowledgments

The power measurements were performed at the IBM Austin Research Laboratory, in an effort to build a power model for the PPC405LP and 405GP processors. We would like to thank IBM ARL members involved in this project: Hazim Shafi, Patrick Bohrer, James Phelan, and James Peterson. We also thank Rabi Mahapatra for his help with the power benchmarks and measurements, as well as Bishop Brock and Chandler McDowell for their assistance during all phases of the project. This work was supported by the Defense Advanced Research Projects Agency through the PARTS (Power-Aware Real-Time Systems) project under Contract No. F33615-00-C-1736.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. C. Rusu, R. Melhem, and D. Mossé, "Maximizing the System Value While Satisfying Time and Energy Constraints," *Proceedings of the 23rd IEEE Real-Time Systems Symposium* (*RTSS'02*), Austin, TX, December 2002, pp. 246–255.
2. P. M. Shriver, M. B. Gokhale, S. D. Briles, D. Kang, M. Cai, K. McCabe, S. P. Crago, and J. Suh, "A Power-Aware, Satellite-Based Parallel Signal Processing Scheme," *Power Aware Computing*, Kluwer Academic Press, New York, 2002, pp. 243–259.
3. B. D. Guenther, "Aided and Automatic Target Recognition Based upon Sensory Inputs from Image Forming Systems," *IEEE Trans. Pattern Anal. & Machine Intell.* **19,** No. 9, 1004–1019 (1997).
4. R. K. Clark, E. D. Jensen, and F. D. Reynolds, "An Architectural Overview of the Alpha Real-Time Distributed Kernel," *Proceedings of the USENIX Workshop on MicroKernels and Other Kernel Architectures*, April 1992, pp. 127–146.
5. F. Yao, A. Demers, and S. Shankar, "A Scheduling Model for Reduced CPU Energy," *Proceedings of IEEE Annual Foundations of Computer Science*, 1995, pp. 374–382.
6. J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer* **24,** No. 5, 58–68 (May 1991).
7. W.-K. Shih, J. W.-S. Liu, and J.-Y. Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints," *SIAM J. Computing* **20,** No. 3, 537–552 (July 1991).
8. 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 Trans. Computers* **45,** No. 7, 802–813 (July 1996).
9. C. M. Krishna and K. G. Shin, *Real-Time Systems*, McGraw-Hill Book Co., Inc., New York, 1997.
10. 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 the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 217–228.
11. H. Aydin, R. Melhem, D. Mossé, and 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, pp. 79–89.
12. E. Chang and A. Zakhor, "Scalable Video Coding Using 3-D Subband Velocity Coding and Multi-Rate Quantization," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, July 1993, Vol. 5, pp. 574–577.
13. W. Feng and 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, pp. 76–80.
14. C. J. Turner and L. L. Peterson, "Image Transfer: An End-to-End Design," *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, August 1992, pp. 258–268.
15. R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek, "A Resource Allocation Model for QoS Management," *Proceedings of the 18th IEEE Real-Time Systems Symposium* (*RTSS'97*), December 1997, pp. 298–307.
16. R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek, "Practical Solutions for QoS-based Resource Allocation Problems," *Proceedings of the 19th IEEE Real-Time Systems Symposium* (*RTSS'98*), Madrid, December 1998, pp. 296–306.
17. 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, pp. 178–187.
18. I. Hong, M. Potkonjak, and M. B. Srivastava, "On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processors," *Proceedings of the International Conference on Computer-Aided Design* (*ICCAD'98*), 1998, pp. 653–656.
19. I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power Optimization of Variable Voltage Core-Based Systems," *Proceedings of the 35th Annual Design Automation Conference* (*DAC'98*), 1998, pp. 176–181.
20. 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, pp. 134–139.
21. 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, pp. 50–61.
22. 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, pp. 156–165.
23. H. Aydin, R. Melhem, D. Mossé, 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*), Delft, Netherlands, June 2001, pp. 164–169.
24. R. Ernst and W. Ye, "Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification," *Proceedings of the International Conference on Computer-Aided Design* (*ICCAD'97*), 1997, pp. 598–604.
25. H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez, "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems," *Proceedings of the 22nd IEEE Symposium on Real-Time Systems*, 2001, pp. 213–222.
26. F. Gruian, "Hard Real-Time Scheduling Using Stochastic Data and DVS Processors," *Proceedings of the International Symposium on Low Power Electronics and Design*, 2001, pp. 46–51.
27. D. Mossé, H. Aydin, B. Childers, and R. Melhem, "Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications," *Proceedings of the Workshop on Compilers and Operating Systems for Low Power* (*COLP'00*), Philadelphia, October 2000, pp. 28–39.
28. D. Shin, J. Kim, and S. Lee, "Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications," *IEEE Design & Test of Computers* **18,** No. 23, 20–30 (March 2001).
29. D. I. 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, pp. 225–234.
30. S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementation*, Wiley & Sons, New York, 1997.

31. H. M. Weingartner and D. N. Ness, "Methods for the Resolution of the Multi-Dimensional 0/1 Knapsack Problem," *Oper. Res.* **15,** 83–103 (1967).

Received November 18, 2002; accepted for publication March 19, 2003

Cosmin A. Rusu *Computer Science Department, University of Pittsburgh, Sennott Square, Pittsburgh, Pennsylvania 15260 (rusu@cs.pitt.edu).* Mr. Rusu received a B.S. degree in computer science from the Technical University of Cluj-Napoca, Romania, in 2000. He joined the University of Pittsburgh that same year and is currently pursuing a Ph.D. degree in the Computer Science Department. His research interests include real-time systems and power-constrained systems.

Rami Melhem *Computer Science Department, University of Pittsburgh, Sennott Square, Pittsburgh, Pennsylvania 15260 (melhem@cs.pitt.edu).* Dr. Melhem received a B.E. degree in electrical engineering from Cairo University in 1976, an M.A. degree in mathematics and an M.S. degree in computer science from the University of Pittsburgh in 1981, and a Ph.D. degree in computer science from the University of Pittsburgh in 1983. He was an Assistant Professor at Purdue University prior to 1986, when he joined the faculty of the University of Pittsburgh, where he is currently a Professor of Computer Science and Electrical Engineering and the Chair of the Computer Science Department. His research interests include real-time and fault-tolerant systems, optical interconnection networks, high-performance computing, and parallel computer architectures. Dr. Melhem has served on program committees for numerous conferences and workshops; he was the general chair for the third International Conference on Massively Parallel Processing Using Optical Interconnections. Dr. Melhem was on the editorial board of the *IEEE Transactions on Computers* and served on the advisory boards of the IEEE technical committees on parallel processing and on computer architecture. He is the editor for the Plenum Book Series in Computer Science and is on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems* and the *Computer Architecture Letters*. Dr. Melhem is a Fellow of IEEE and a member of the ACM.

Daniel Mossé *Computer Science Department, University of Pittsburgh, Sennott Square, Pittsburgh, Pennsylvania 15260 (mosse@cs.pitt.edu).* Dr. Mossé received a B.S. degree in mathematics from the University of Brasilia in 1986, and M.S. and Ph.D. degrees in computer science from the University of Maryland in 1990 and 1993, respectively. In 1992 he joined the faculty of the University of Pittsburgh, where he is currently an Associate Professor. His research interests include fault-tolerant and real-time systems, as well as networking. The major thrust of his research in the new millennium is power-aware computing and security. Dr. Mossé has served on program committees for all major IEEE-sponsored real-time related conferences and as program and general chairs for the RTAS and RT Education Workshop. Typically funded by NSF and DARPA, his projects combine theoretical results and implementations. Dr. Mossé is a member of the editorial board of the *IEEE Transactions on Computers*, of the IEEE Computer Society, and of the Association for Computing Machinery.