An Integrated Approach for Applying Dynamic Voltage Scaling to Hard Real-Time Systems

Yanbin Liu, Aloysius K. Mok
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{ybliu,mok}@cs.utexas.edu

Abstract

Wireless and portable devices depend on the limited power supplied by the battery. Dynamic Voltage Scaling (DVS) is an effective method to reduce CPU power consumption. For real-time systems, DVS algorithms must not only provide enough CPU cycles, but also guarantee that no job misses its deadline. In this paper, we propose an integrated approach for applying DVS to real-time systems. We define two functions, the available cycle function (ACF) and the required cycle function (RCF), to capture the CPU workload of the real-time tasks. We then formulate the DVS scheduling problem for real-time systems as a nonlinear optimization problem and propose an optimal off-line algorithm to solve this problem. We also propose a novel on-line algorithm with time complexity $O(1)$ to further reduce power consumption when a job uses fewer execution cycles than the worst-case budget. The algorithms in this paper are based solely on ACF and RCF, and may be applied to different scheduling policies. We illustrate the generality of our approach over previous research by applying our method to EDF and RM scheduling policies and deriving the optimal off-line DVS algorithms for them. Our simulation results show significant improvement over previous work.

1 Introduction

Many wireless and portable devices depend on the limited power supplied by the battery. Dynamic voltage scaling (DVS) is an effective method to increase battery life by reducing the power consumption of a processor. Specifically, DVS reduces the energy consumption of a CMOS processor by dynamically controlling its supply voltage. The power consumed per CPU cycle can be expressed as $P = kCV^2f$, where k is a constant, C the total capacitance of wires and transistor gates, V the supply voltage, and f the clock frequency. DVS technology has already been incorporated into many processors produced by companies including Transmeta, AMD, and Intel.

Changing a processor's supply voltage also changes its speed, since the relationship between clock frequency and supply voltage is $f \propto (V - V_t)^2/V$, where V_t is the threshold voltage. As a result, DVS may hurt the performance of some systems in a significant way, in particular, real-time systems where some deadlines are hard. In past work, many algorithms have been proposed to apply DVS to general purpose and soft real-time systems only, *e.g.*, [22, 4, 17, 7, 16, 19].

Applying DVS to hard real-time systems is more difficult. When a DVS algorithm reduces CPU speed to save energy, it must guarantee that all jobs still meet their deadlines. In the past few years, a large number of algorithms are proposed, *e.g.*, [23, 8, 9, 10, 21, 1, 11, 5, 6, 18, 19, 25, 12, 20, 24]. Although these algorithms have demonstrated the benefits of applying DVS to hard real-time systems, they generally only apply to some specific real-time scheduler. There has not been any algorithm designed to work with different real-time schedulers.

The objective of this paper is to propose an integrated approach for applying DVS to hard real-time systems. Using this approach, we derive unified algorithms that work with different real-time schedulers. We illustrate the generality of the approach by applying it to RM and EDF schedulers; we shall derive the optimal off-line DVS algorithms for these real-time scheduling policies.

Specifically, we shall propose the definition of two functions: the available cycle function (ACF) and the required cycle function (RCF). The values of these two functions are derived from the parameters of the task set and the scheduling policy used to dispatch the tasks of the system. These functions help us to capture the workload of a real-time system by characterizing an envelope: they constrain the range of a speed function so that if the CPU executes at the speed defined by the function, then there is no idle cycle and no job misses its deadline.

Using ACF and RCF, we formulate the energy minimiza-

tion problem as a constrained nonlinear optimization problem. We prove that for strictly convex power functions, which are the case for all known power models, we can simplify the formulation of the problem. We propose an algorithm to solve the problem and to derive the optimal speed function that minimizes energy consumption. The algorithm is shown to work with both EDF and RM.

Since the static algorithm is based on the worst-case budget, we also propose a novel, online, dynamic algorithm to reclaim slack cycles which are generated by early completion of jobs; by a job, we mean an instance of a recurring task (periodic or otherwise) in a schedule. The dynamic algorithm lowers CPU speed whenever slack cycles are available; the new speed will not cause any missed deadline or extra idle cycles. The time complexity of the algorithm is $O(1)$. Furthermore, the algorithm may trade runtime cost for energy savings. It can be improved, with added complexity so that an optimal CPU speed solution is always found at any time. Again, the dynamic algorithm works with both EDF and RM.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we introduce the task model and the processor model. In Section 4, we define the functions to be used. In Section 5, we formulate the energy minimization problem and present our solution. In Section 6, we present the online dynamic algorithm. We present evaluations of our approach in Section 7 and conclude in Section 8. According to the space limitation, we omit the proofs of most lemmas and theorems in the paper. Interested readers are referred to [15].

2 Related Work

In applying DVS to hard real-time systems, different scheduling policies and task models have been examined in past work. Both static and dynamic algorithms have been proposed to achieve different system goals.

Most previous research has considered the EDF scheduler. In [23], Yao et al use the aperiodic task model and propose an optimal static solution. Extending their result, Hong et al also consider the effect of speed switching overhead [10]. The authors of [18, 2] use the periodic task model. They define the CPU speed in terms of the utilization of the task set and show that the speed is optimal in minimizing energy consumption.

Other previous research has considered the RM scheduler. The authors of [5, 18] propose static algorithms for the RM scheduler. Pillai and Shin [18] use a constant speed for the whole task set. Gruian [5] uses a constant speed for each task of the task set. However, the proposed solutions are not optimal. More importantly, to our knowedge there has been no previous unified algorithm that can work with different scheduling policies or different models.

Since the ratio of the worst-case execution time to the

best-case execution time can be as high as 10 in some applications [3], many researchers have proposed dynamic algorithms to reclaim unused CPU cycles, ı.e., slack cycles. In [18, 2], the authors propose dynamic algorithms for the EDF scheduler. In [21, 5, 18], the authors propose dynamic algorithms for the RM scheduler. These algorithms collect the slack cycles at runtime whenever a job finishes early and reclaim the unused cycles by reducing the CPU speed. However, the reduction of the CPU speed is under different constraints. In [18], for EDF, the speed is reduced until the arrival of the next job belonging to the same task; for RM, the speed is reduced until the next deadline. In [21], the speed is reduced when only one job is ready to execute. In [5], the speed is reduced depending on the job's priority. In [2], the speed is reduced when the dynamic completion time of a job is no later than its completion time in the static schedule.

While most research focuses on energy optimization, the authors of [12, 20] consider the optimization of system performance or system value under energy consumption constraints. There has also been other research pertaining to different system configurations. For example, DVS for multi-processors is discussed in [11, 25]. DVS for non-preemptible tasks is discussed in [8]. DVS for sporadic tasks is discussed in [9]. DVS for tasks with different power consumption characteristics is discussed in [1]. DVS for tasks with precedence constraints is discussed in [6], and DVS for tasks with non-preemptible sections is discussed in [24].

3 System Model

3.1 Task Model

We shall use the well known Liu and Layland periodic task model [14] to demonstrate our approach. Let $T = \{T_1, T_2, ..., T_n\}$ denote a task set. The tasks are periodic, preemptible, and mutually independent. Each task T_i is defined by a pair, (p_i, c_i), where p_i is the period, and c_i the worst-case workload measured in CPU cycles. As an example, the first task in the task set $\{T_1(4, 2), T_2(5, 1), T_3(10, 1)\}$ has period 4, relative deadline 4 and workload 2. We assume that all tasks start at time 0. We use $J(i, k)$ to denote the k-th job of a task T_i; its available time is $p_i * (k - 1)$ and its deadline is $p_i * k$.

We define the hyper period H to be the smallest interval from time 0 in the periodic schedule after which the schedule repeats itself. For our task model, H is the least common multiplier (LCM) of all the task periods in the task set.

We want to point out that the results in this paper can be easily extended to the cases where tasks have preperiod deadlines, i.e., a task may have a relative deadline that is smaller than its period and where tasks may have start times other than the beginning of their periods. We choose the simpler task model for ease of discussion.

3.2 Processor Model

We assume the processor can change its voltage continuously. In other words, a processor can operate at any normalized speed from 0 to 1. When the speed is 0, the processor is in shutdown mode. We shall ignore the speed switching overhead which in general is small [18].

From the Introduction section, we note that P, the power consumed per CPU cycle is proportional to $V^2 f$, and f is proportional to $(V - V_t)^2 / V$, where V is the supply voltage, f the frequency, and V_t the voltage threshold. The exact definition of P depends on the specific hardware. We shall not assume any fixed form for P, except that it is a strictly convex function of the frequency. We use $P(S)$ to denote that P is a function of speed S.

In this paper, only real-time tasks are considered. CPU cycle not assigned to a task is considered to be wasted and should be avoided to save energy.

4 Functions Definition

In this section, we shall define three functions: Speed Function $S(t)$, Available Cycle Function $ACF(t)$, and Required Cycle Function $RCF(t)$. These functions are the building blocks of our approach.

4.1 Speed Function

Definition 1 *The speed function $S(t)$ is the CPU speed, in cycles per time unit, at time t.*

Using our processor model, we have $0 \le S(t) \le 1$.

The cycles supplied by a processor during a time period $(t_1, t_2]$ is $\int_{t_1}^{t_2} S(t) dt$. In Figure 1(a), we show several speed functions. The x-axis shows time; the y-axis shows CPU cycles. The curve shows the cycles supplied by the speed function up to time t, *i.e.*, $\int_0^t S(x) dx$. We can see that $S(t)$ during a period $(t_1, t_2]$ is a constant iff $\int_0^t S(x) dx$, where $t_1 \le t \le t_2$, is a straight line from coordinate $(t_1, \int_0^{t_1} S(x))$ to coordinate $(t_2, \int_0^{t_2} S(x) dx)$.

The energy consumed by a speed function $S(t)$ during a time period $(t_1, t_2]$ is $E(t_1, t_2, S) = \int_{t_1}^{t_2} P(S(t)) dt$.

The optimization objective of applying DVS to a real-time system is to derive a speed function that minimizes the energy consumption and guarantees no missed deadline. For our task model, since a schedule repeats itself every hyper period, we only need to derive a speed function $S(t)$ for the hyper period $(0, H]$ that minimizes $E(S) = E(0, H, S)$ and guarantees no missed deadline. We now specify the constraints on such a speed function.

4.2 ACF and RCF

Definition 2 *The available cycle function $ACF(t)$ is defined as the upper bound of cycles available for execution up to time t.*

Given a periodic task set $T = \{T_i : 1 \le i \le n\}$, $ACF(t) = \sum_{i=1}^{n} (\lceil \frac{t}{p_i} \rceil * c_i)$. The function depends only on the task set and is independent of the scheduler. Consider a task set $\{T_1(4, 2), T_2(5, 1), T_3(10, 1)\}$. The available cycle function $ACF(t)$ for this task set is shown in Figure 1(b).

In [14], the same function is used to analyze RM schedulability. We use this function here differently.

Lemma 1 *If a speed function S optimizes energy consumption, then $\int_0^t S(x) dx \le ACF(t)$ for all time t.*

Proof. Suppose $\int_0^t S(x) dx > ACF(t)$ at a time t. This implies that the CPU must idle at some time. We can set the speed to 0 when the CPU idles and get a new speed function that provides the same cycles to the tasks. This new speed function consumes less energy because when the CPU idles, it operates at a speed bigger than 0 but does nothing useful. Thus, if S optimizes energy consumption, we have $\int_0^t S(x) dx \le ACF(t)$ for all time t. □

Definition 3 *The basic required cycle function $BRCF(t)$ is the minimal number of cycles that must be executed up to time t.*

If less than $BRCF(t)$ cycles are executed up to time t, some job misses its deadline. Given a periodic task set $T = \{T_i : 1 \le i \le n\}$, $BRCF(t) = \sum_{t=1}^{n} (\lfloor \frac{t}{p_i} \rfloor * c_i)$. This function depends only on the task set and is independent of the scheduler. Consider the previous task set example $\{T_1(4, 2), T_2(5, 1), T_3(10, 1)\}$. Figure 1(b) also shows $BRCF(t)$ for it.

Definition 4 *Given a scheduler, the required cycle function $RCF(t)$ defines the CPU cycles that need to be supplied to the tasks up to time t so that no job misses its deadline.*

If a task set is schedulable, $ACF(t)$, or any function $f(t) \ge ACF(t)$, can be a $RCF(t)$ for any scheduler. However, if $RCF(t') > ACF(t')$ at some time t', we can always set $RCF(t') = ACF(t')$ without affecting the schedulability. Thus, we assume $RCF(t) \le ACF(t)$. We also assume, without loss of generality, that $RCF(t)$ is a non-decreasing step function.

Different from $BRCF(t)$, $RCF(t)$ depends on both the task set and the scheduler. Since a scheduler specifies the execution order of the jobs, it is obvious that $RCF(t) \ge BRCF(t)$. We want to define $RCF(t)$ as small as possible while keeping $BRCF(t)$ as the lower bound.

Consider the EDF scheduler specifically. We have the following lemma.

Lemma 2 ([15]) *$BRCF(t)$ is a $RCF(t)$ for EDF.*

In general, it is not obvious how to define a good $RCF(t)$ for general scheduling policies. However, if a task set is schedulable with the full speed $S = 1$, we can always define a $RCF(t)$ so that at least one speed function constrained by it can be found. For example, we can run

Figure 1. (a) Speed Functions; (b) ACF, BRCF and RCF_{RM} for the task set example.

the task set using any scheduling policy over the hyper period at full speed. Let us call this schedule C. We can define $RCF(t)$ to be a step function that can increase at each deadline. The value at each deadline is the total cycles scheduled in C before the point. We can prove [15] that $RCF(t)$ defined in this way is a valid required cycle function. We also have a speed function S_1. At time t, if the CPU is busy in C, $S_1(t) = 1$; otherwise $S_1(t) = 0$. This speed function is constrained by $ACF(t)$ and $RCF(t)$; the task set is schedulable with it.

Consider RM specifically. We shall discuss our method of defining RCF_{RM} later. The method applies to any fixed-priority scheduler in a preemptive system. The RCF_{RM} for the task set example is shown in Figure 1(b).

Definition 5 *Given a scheduler, a valid speed function is a speed function $0 \leq S(t) \leq 1$ that guarantees no missed deadline.*

Lemma 3 *A speed function $0 \leq S(t) \leq 1$ that satisfies $ACF(t) \geq \int_0^t S(x)dx \geq RCF(t)$ at all time t is a valid speed function.*

The lemma follows directly from the definitions of ACF and RCF. In the following, we shall use the term RCF constraint to refer to the constraint that is $\int_0^t S(x)dx \geq RCF(t)$. We use the term ACF constraint in a similar way.

By defining ACF and RCF, we capture the workload requirement of a task set under a scheduling policy. A speed function satisfying both the ACF and RCF constraints guarantees that no job misses its deadline and there is no idle cycle. Such a speed function is a good candidate for minimizing energy consumption in a real-time system.

4.3 Properties

Below is a summary of the properties of the defined functions.

1. $0 \leq S(t) \leq 1$ for $0 \leq t \leq H$.
2. $ACF(t)$, $BRCF(t)$, and $RCF(t)$ are non-decreasing step functions of time t. A step point of ACF is at the available time of each job; a step point of $BRCF$ is at the deadline of each job.
3. $ACF(t) \geq RCF(t) \geq BRCF(t)$ for $0 \leq t \leq H$.
4. $ACF(H) = RCF(H) = BRCF(H)$.

5. At most $ACF(t)$ cycles can be executed up to time t. If $\int_0^t S(x)dx > ACF(t)$, there are $\int_0^t S(x)dx - ACF(t)$ idle cycles.
6. At least $BRCF(t)$ cycles should be executed up to time t. If $\int_0^t S(x)dx < BRCF(t)$, there are missed deadlines.
7. If $ACF(t) \geq \int_0^t S(x)dx \geq RCF(t)$ at all time t, no job misses its deadline.
8. If $ACF(t) = RCF(t)$ during a time period $(t_1, t_2]$, then $S(t) = 0$ during the period.

In the next section, we shall propose an algorithm to find the static speed function that minimizes energy consumption of a hyper period. Since the definition of ACF and RCF separates the real-time scheduling issue from the derivation of a DVS speed function, we can apply the algorithm to any scheduling policy. Better energy savings than previous research or optimal energy savings can be achieved with appropriate RCFs.

5 An Optimal Static Algorithm

In this section, we formulate the energy minimization problem as a nonlinear optimization problem and propose an algorithm to derive the optimal speed function.

5.1 The Energy Optimization Problem

Suppose ACF and RCF are given. The energy optimization problem can be formulated as follows, where $0 \leq t \leq H$:

$$minimize: \quad E(S) = \int_0^H P(S(x))dx$$
$$subject\,to: \quad 0 \leq S(t) \leq 1,$$
$$RCF(t) \leq \int_0^t S(x)dx \leq ACF(t).$$

This is a nonlinear optimization problem. The solution of the problem is an optimal speed function.

Definition 6 *We call a valid speed function that minimizes energy consumption an optimal speed function.*

Since $P(S)$ is a strictly convex function of S, we can prove the following lemmas.

Lemma 4 ([15]) *The energy consumption $E(S)$ is a strictly convex function of S.*

Lemma 5 ([15]) *The optimal speed function is unique.*

Lemma 6 ([15]) *The optimal speed function is a piecewise linear function that changes speed only at the time when ACF or RCF changes.*

By Lemma 6, we know that the optimal speed function consists of a sequence of constant speed segments that change at the time points where ACF or RCF increases. Thus, we simplify our optimization problem as following.

Let $a_0, ..., a_m$ be a sorted sequence of time when ACF or RCF increases. Let $a_0 = 0$. Let S_j be the constant speed segment in the period $(a_{j-1}, a_j]$.

$$minimize: \quad E(S) = \sum_{j=1}^{m} P(S_j) * (a_j - a_{j-1})$$
$$subject\ to: \quad 0 \leq S_j \leq 1\ for\ 1 \leq j \leq m$$
$$RCF(a_k) \leq \sum_{j=1}^{k} S_j * (a_j - a_{j-1})\ at\ deadline\ a_k,$$
$$ACF(a_l) \geq \sum_{j=1}^{l} S_j * (a_j - a_{j-1})\ at\ avail.\ time\ a_l.$$

Next, we consider the algorithm listed in Figure 2 for solving this optimization problem with time complexity polynomial in m. Consider the 2-dimensional plane whose axes are time and CPU cycles. The algorithm tries to extend a straight line as long as possible from coordinate $(0, 0)$ to $(H, AFC(H))$ that lies within the region delineated by ACF and RCF. This line can only touch either the pre-step point of ACF or the post-step point of RCF. (By pre-step or post-step point, we refer to the points immediately before or after a step function changes its value.) Let the point $P = (t, c)$ be the last RCF point, or ACF point if no RCF point is on the line. A straight line denoted a constant speed. This speed is assigned to the time period $(0, t]$. The algorithm then tries to find the next such straight line starting at point P. This procedure repeats until it reaches $(H, AFC(H))$ and produces the speed function from 0 to H. The main function of the algorithm is called $speedFunctionForPeriod$ which tries to find the next straight line, i.e., the next constant speed.

The optimality of the algorithm can be proven by the following Lemma.

Lemma 7 ([15]) *A valid speed function is an optimal speed function iff for any time value $t_1 < t_2$, the following is true: if the speed during the period $(t_1, t_2]$ is not a constant, then the constant speed is not constrained by ACF or RCF.*

Theorem 1 ([15]) *Given ACF and RCF, the algorithm listed in Figure 2 is an optimal algorithm; the speed function derived from the algorithm is the optimal speed function.*

Next, we use the algorithm to derive the optimal speed function for two of the most popular schedulers used in a real-time system, EDF and RM.

5.2 EDF Scheduler

Using our approach, we can verify the result in [18, 2].

```
 1 DVS_SpeedSchedule () {
 2    startTime = 0; startValue = 0;
 3    do{
 4       speedItem =speedFunctionForPeriod
 5                (startTime, startValue, H);
 6       if speedItem.speed > 1
 7          exit (TASK_SET_NOT_SCHEDULABLE);
 8       startValue+ = speedItem.speed*
 9             (speedItem.endTime − startTime);
10       startTime = speedItem.endTime;
11    } until (speedItem.endTime == H);
12 }
   ▷Let d be any RCF step point in (startT, endT] .
   ▷Let a be any ACF step point in (startT, endT].
   ▷Let b be any ACF step point in (startT, d*].
13 speedFunctionForPeriod (startT, startV, endT) {
14    for each d, RCSlope(d) = (RCF(d)−startV)/(d−startT);
15    for each a, ACSlope(a) = (ACF(a)−startV)/(a−startT);
16    speed_req = max RCSlope(d);
17    speed_avail = min ACSlope(a);
18    if speed_avail ≥ speed_req
19       return (speed_req, endT);
20    else { // speed_req cause idle cycles in (startT, endT]
21       d* = max{d : speed_req = RCSlope(d)};
22       speed_avail = min ACSlope(b);
23       if speed_avail ≥ speed_req
24          return (speed_req, d*);
25       else { // speed_req cause idle cycles in (startT, d*]
26          a* = max{b : speed_avail = ACSlope(b)};
27          RCF(a*) = ACF(a*);
28          s=speedFunctionForPeriod (startT, startV, a*);
29          reset RCF(a*);  return s; }}
30 }
```

Figure 2. Optimal Algorithm

Theorem 2 ([15]) *For EDF, the optimal speed function is* $S(t) = \frac{BRCF(H)}{H} = \sum_{i=1}^{n} \frac{c_i}{p_i}$.

Figure 3 (a) shows the EDF optimal speed function for the given task set example.

Since $BRCF$ is the lower bound of any RCF, we prove that EDF is an optimal real-time scheduler with any speed function, which means that if a task set can be scheduled with a speed function by any scheduler, it can be scheduled with the same speed function by EDF.

Theorem 3 ([15]) *EDF is an optimal real-time scheduler with any speed function.*

5.3 RM Scheduler

Consider the RM scheduler. We define RCF_{RM} by borrowing the idea from scheduling soft aperiodic tasks in fixed-priority preemptive systems [13].
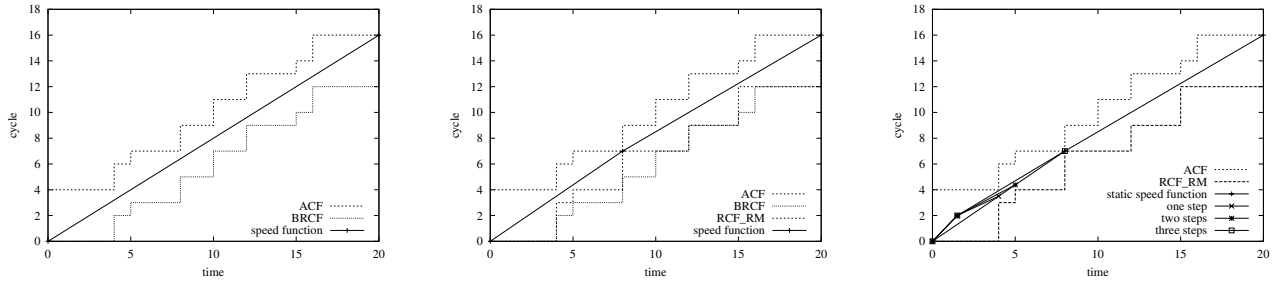
Figure 3. Speed function: (a)EDF static; (b)RM static (c) RM dynamic

First, we follow the method developed by Lehoczky and Ramos-Thuel [13] to generate a special job schedule. Instead of scheduling the soft aperiodic tasks as early as possible as is in [13], we let the CPU idle as early and as much as possible and schedule the jobs of the periodic tasks as late as possible. The details can be found in [15]. Given the task set $\{T_1(4,2), T_2(5,1), T_3(10,1)\}$, Figure 4 shows the special job schedule generated by using the method.
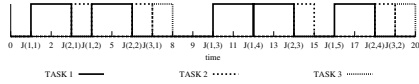

Figure 4. The special job schedule

Based on this special job schedule, we define RCF_{RM} to be a step function that can increase at each deadline. The value at each deadline is the total cycles scheduled in the job schedule before the deadline. We can see that RCF_{RM} is a valid required cycle function for the RM scheduler. We can also prove that RCF_{RM} defined in this way is the lower bound of any RCF for RM and the speed function found by our algorithm is optimal. [15].

Theorem 4 ([15]) *For RM, using the RCF_{RM} defined above, our algorithm derives the optimal speed function.*

Consider our task set example. The derived RCF_{RM} and the optimal speed function based on it are shown in Figure 3 (b).

6 Online Reclaim Algorithm

The off-line algorithm is based on the worse-case budget. At runtime, slack cycles are generated if a job uses less than the worst-case budget. We can easily adopt our technique to reclaim slack cycles.

The idea is that, when a job finishes with δ less cycles compared with the static schedule at time t, this can be interpreted as if $\int_0^t S(x)dx$ is increased by δ. Beginning at time t, we have the same speed optimization problem with ACF and RCF constraints. In other words, a new optimal solution can be found at any time during run time. Based on the off-line speed function, faster algorithms can be used online. We propose such an algorithm which has time complexity $O(1)$.

First, we need to save some additional information: the current speed which may be different from the off-line speed, the accumulated cycles FC up to the curren time and SC which is the accumulated cycles calculated from the off-line speed function. The algorithm follows the off-line speed function and updates the information at each scheduling point; we use scheduling point to refer to any time when a job becomes available, is completed, or is preempted.

Normally, FC is increased by the CPU cycles supplied at the current speed. However, when a job finishes earlier, we increase FC further with the slack cycles δ defined above. Thus, $FC - SC$ represents the slack cycles at current time t_0. At this point $ACF(t_0) \geq FC$ still holds. If $FC > SC$, we reduce CPU speed so that FC will approach SC. Once $FC = SC$ again, we can follow the off-line speed function from then on. Since we keep $FC \geq SC$ all the time and $SC \geq RFC$, no job will miss its deadline as long as we keep $ACF \geq FC$. Note that the online algorithm tries to find a new speed in the range constrained by ACF and SC.

When defining the new speed, we can look ahead at any number of future ACF step points to find a new speed. The more ACF points we check, the smoother the new speed is and the more energy is saved. If we check all ACF points up to the end of a hyper period, we have the optimal solution. This is a trade-off between runtime cost and energy savings. To keep the runtime cost constant, we look at a fixed number of ACF points and assign a constant speed to the new speed.

Let t be a decision time. Consider the first ACF step point $(r, ACF(r))$ after t. Let $speed_s$ be the off-line speed during $(t, r]$; $speed_s$ is constant. If $FC = ACF(r)$, the new speed is 0 during (t,r]. If $FC < ACF(r)$, the new speed during $(t, r]$ can be any speed between $\max\{0, \frac{SC(r)-FC}{r-t}\}$ and $\min\{speed_s, \frac{ACF(r)-FC}{r-t}\}$. One option is to keep the current speed if possible. Another option is $\frac{SC(r)-FC}{r-t}$ if it is bigger than 0. We can look ahead at more points in a similar way.

Consider our task set $\{T_1(4,2), T_2(5,1), T_3(10,1)\}$. Suppose $J(1,1)$ is finished earlier at time 1.5. We have $FC - SC = 2 - 0.875 * 1.5 = \frac{11}{16}$ slack cycles at the time. Figure 3(c) shows the new speed derived by checking one, two or three future ACF step points. We use the second option above to define a new speed. We are unable to look at

more than three points in the example because we cannot find a constant new speed that causes no extra idle cycles.

To our best knowledge, we note that in previous research, a reclaim algorithm can look ahead only one task period. Our method has no such limit, hence more energy reduction can be achieved by our approach. Also note that our algorithm reclaims slack cycles from multiple jobs of different task.

Our algorithm depends on ACF and a off-line speed function. Besides the optimal speed function, it applies to any valid speed function defined off-line, including what we call the speculative speed function [15].

7 Evaluations

We use simulations to evaluate our algorithms. The periodic task model is used in the simulations. The period of each task is uniformly distributed between 20 and 100 units. The worst-case computation time (WCET) of each task is uniformly distributed between 1 and 20 units. The best-case execution time (BCET) is defined to be from 10% to 100% of WCET. The actual execution time of each job is generated using a normal distribution. The mean of the distribution is $\frac{BCET+WCET}{2}$; the standard deviation is $\frac{WCET-BCET}{6}$. We assume that the processor has continuous available speed, and $P = S^3$. For each experiment, we repeat 100 times and report the average.

We first compare our RM static algorithm with the algorithm of Pillai and Shin [18] which uses a constant speed for the whole task set. This constant speed is derived by analyzing critical instants of the tasks and cannot be smaller. We vary the task set utilization and experiment with task sets consisting of 5, 10 or 15 tasks. We measure our performance improvement by the ratio of the energy consumption of our algorithm to that of the compared algorithm. Figure 5 shows the result. We observe that our static algorithm can save up to 40% of energy better than the algorithm in [18].
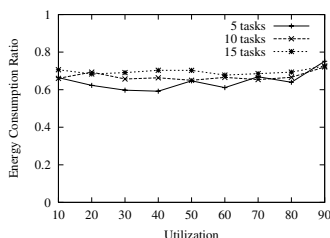

Figure 5. Evaluation of RM static algorithm

We next demonstrate the energy savings of the dynamic reclaim algorithm. We report the ratio of the energy cost of the dynamic algorithm to that of our static algorithm. In the first set of experiments, we fix $\frac{BCET}{WCET} = 0.1$ and vary the task set utilization. We only look ahead one ACF step point in the simulations. Figure 6(a) and 6(b) shows the result for RM and EDF, respectively. We observe that the energy savings of the dynamic algorithm is almost independent of task set utilization. This is because our optimal static algorithm explores the energy savings when the utilization is less than 1 and is already very effective. We also observe that the energy savings is similar between EDF and RM. This is because our algorithm reclaims slack cycles based on the optimal static speed and ACF; it does not depend on any specific scheduler directly.

In the second set of experiments, we fix task set utilization at 50% and vary the ratio $\frac{BCET}{WCET}$. Figure 6(c) shows the result. We observe that our algorithm saves quite a significant amount of energy, up to 75%. We also observe that the energy savings decreases as $\frac{BCET}{WCET}$ increases. Since fewer slack cycles are available when the ratio increases, it is expected that the energy savings by reclaiming slack cycles will decrease.

We do not investigate the effects when the processor is not ideal, such as, the shutdown energy cost is not 0, only discrete speeds are available, or speed switching overhead can not be ignored. We leave the topics to our next paper.

8 Conclusion

In this paper, we have proposed an intergrated apporach for applying DVS to real-time systems. We define two functions, ACF and RCF, to capture the workload characteristics of a real-time system. Once ACF and RCF are defined, we formulate DVS scheduling of a real-time system as a constrained nonlinear optimization problem. For strictly convex power functions, which are the case for all known power models, we simplify the formulation of the problem and propose an algorithm to solve it. We demonstrate the approach by using the periodic task model under both EDF and RM scheduling policies. We define the appropriate RCF for EDF and RM; we derive the optimal off-line speed function that minimizes energy consumption for them. We showed by simulation that the RM algorithm can save up to 40% energy compared with another algorithm [18].

Since a job's actual execution demand is sometimes much shorter than the worst-case budget, we propose a dynamic algorithm to reclaim slack cycles. The dynamic algorithm works with different schedulers. It can reclaim cycles from multiple jobs of different tasks, and it can look ahead at more than one task period to reduce the speed. This algorithm has time complexity O(1) and it can trade runtime cost for energy savings. The optimal solution can be found at any time during a computation. By using simulation, we show that our dynamic algorithm can save up to 75% of energy when $\frac{BCET}{WECT} = 0.1$ compared with the static optimal algorithm.

For future work, we would like to extend our approach to other task and processor models. We shall also look into
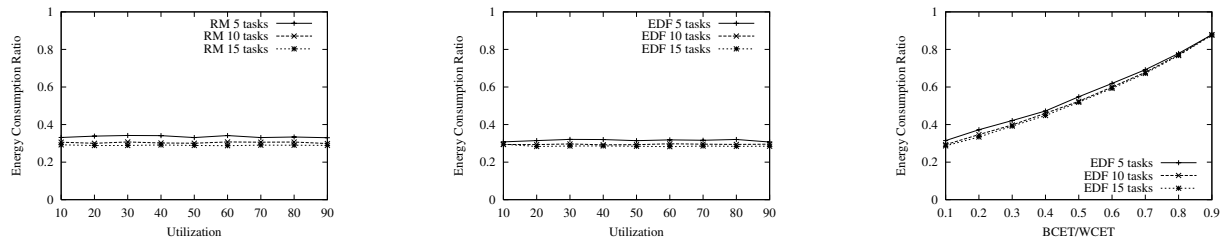
Figure 6. Evaluation of dynamic algorithm: (a)(b)$\frac{BCET}{WCET} = 0.1$; (c) Utilization=50%.

the effect of discrete CPU speeds and speed switching overhead.

References

[1] H. Aydin, R. Melhem, and D. Mosse. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[2] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Real-time System Symposium*, London, UK, Dec. 2001.

[3] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Internatioal Conference on Computer Aided Design*, San Jose, CA, Nov. 1997.

[4] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *International Conference on Mobile Computing and Networking*, Berkeley, CA, Nov. 1995.

[5] F. Gruian. Hard real-time scheduling using stochastic data and DVS processors. In *International Symposium on Low Power Electronic and Design*, Huntington Beach, CA, Aug. 2001.

[6] F. Gruian and K. Kuchcinski. Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In *International Symposium on Low Power Electronic and Design*, Huntington Beach, CA, Aug. 2001.

[7] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In *Symposium on Operating Systems' Design and Implementation*, San Diego, CA, Oct. 2000.

[8] I. Hong, D. Kirovshi, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *Design Automation Conference*, San Francico, CA, June 1998.

[9] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processors. In *International Conference on Computer-Aided Design*, San Jose, CA, Nov. 1998.

[10] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Real-time System Symposium*, Madrid, Spain, Dec. 1998.

[11] D.-I. Kang, S. Crago, and J. Suh. Power-aware design synthesis techniques for distributed real-time systems. In *ACM Workshop on Languages, Compilers, and Tools for Embedded Systems*, Snowbird, UT, June 2001.

[12] D.-I. Kang, S. Crago, and J. Suh. Technique for energy-efficient real-time systems. In *Real-time System Symposium*, Austin, TX, Dec. 2002.

[13] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-time System Symposium*, Phoenix, AR, Dec. 1992.

[14] J. W. S. Liu. *Real-time Systems*. Prentice Hall, 2000.

[15] Y. Liu and A. Mok. An integrated approach for applying dynamic voltage scaling to real-time systems. Technical Report TR-03-YBL-01, University of Texas at Austin, Mar. 2003.

[16] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Joint International Conference on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.

[17] T. Pering, T. Burd, and R. W. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithm. In *International Symposium on Low Power Electronic and Design*, Monterey, CA, Aug. 1998.

[18] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM symposium on operating systems principles*, Banff, Canada, Oct. 2001.

[19] V. Raghunathan, P. Spanos, and M. Srivastava. Adaptive power-fidelity in energy-aware wireless embedded systems. In *Real-time System Symposium*, London, UK, Dec. 2001.

[20] C. Rusu, R. Melhem, and D. Mosse. Maximizing the system value while satisfying time and energy consumption. In *Real-time System Symposium*, Austin, TX, Dec. 2002.

[21] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference*, New Orleans, LA, June 1999.

[22] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Symposium on Operating Systems' Design and Implementation*, Monterey, CA, Nov. 1994.

[23] F. Yao, A. Demers, and S.Shenker. A scheduling model for reduced CPU energy. In *Symposium on Foundations of Computer Science*, Milwaukee, WI, Oct. 1995.

[24] F. Zhang and S. Chanson. Processor voltage scheduling for real-time tasks with non-preemptible sections. In *Real-time System Symposium*, Austin, TX, Dec. 2002.

[25] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in mutli-processor real-time systems. In *Real-time System Symposium*, London, UK, Dec. 2001.