Processor Voltage Scheduling for Real-Time Tasks with Non-Preemptible Sections

Fan Zhang & Samuel T. Chanson
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{zhangfan,chanson}@cs.ust.hk

Abstract

As mobile computing is getting popular, there is an increasing interest in techniques that can minimize energy consumption and prolong the battery life on mobile devices. Processor voltage scheduling is an effective way to reduce energy dissipation by reducing the processor speed. In this paper, we study voltage scheduling for real-time periodic tasks with non-preemptible sections. Three schemes are proposed to address this problem. The static speed algorithm derives a static feasible speed based on the Stack Resource Policy (SRP). As worst-case blocking does not always occur, the novel dual speed algorithm switches the processor speed to a lower value whenever possible. The dynamic reclaiming algorithm deploys a reservation-based approach to reclaim unused run time for redistribution. It effectively decreases the processor idle time and further reduces the processor speed. The feasibility conditions are given and proved. Simulation results show that the two dynamic algorithms can reduce processor energy consumption by up to 80 percent over the static speed scheme.

1 Introduction

For convenience and ease of access, more and more personal computing and communication devices are becoming portable and mobile. These include laptop computers, pocket PCs and PDAs. Most of them are powered by batteries with limited power capacity. Some recent portable systems are equipped with powerful processors that were available on desktop computers only a few years ago. The performance boost comes at the cost of higher energy consumption. On the other hand, as battery technology has not been keeping up with the speed of the processors, the limited battery power has become a major concern. How to conserve power and prolong battery life is of critical importance and has received much attention recently [1, 2, 3].

Work exists on evaluating the power consumption of portable computing systems [4, 5]. It was found that the display (including backlight), processor, hard disk and wireless LAN card attribute to most of the power consumption [4, 5]. In particular, the processor may consume up to 25% of the system power for laptop computers [4]. This percentage is even higher for PDAs since hard disks are not used. If the power[1] consumed by the processor is reduced, the power consumption of the whole system is also substantially reduced.

As the processor may not be fully utilized all the time, the variation in system load can be exploited to reduce power dissipation. The processor can be turned off or made to operate at lower a speed when the system has no or little work to do. Some processors (e.g., the Crusoe processor of Transmeta [6] and the Intel StrongARM processor [7]) allow the processor voltage to be dynamically adjusted. This is called dynamic voltage scaling (DVS). The relationship among power consumption rate (P), supply voltage (V_s) and clock frequency (f) can be described by the following formula [8]:

$$P = C \cdot f \cdot V_s^2, \tag{1}$$

where C is the switched capacitance. Furthermore, as the operating frequency f (and therefore the processor speed) is approximately proportional to the supply voltage [9], the power consumption rate is roughly proportional to the cube of the supply voltage. It follows that if we decrease the processor speed by lowering the supply voltage, the power consumption will be reduced. Voltage scaling is particularly useful for real-time applications because a longer computation time is acceptable as long as the deadlines are not violated. With voltage scaling, the job scheduler must make two types of decisions: which job to run and at what speed to run it. In this paper, we shall refer to job scheduling with voltage scaling simply as "voltage scheduling" for short.

[1] In this paper, the word "power" is used with the same meaning as "energy"

1052-8725/02 $17.00 © 2002 IEEE

Much work has been done on voltage scheduling with the objective of minimizing processor energy consumption [1, 2, 3, 9, 10, 11, 12]. Most of them assume the tasks are preemptible. However, in reality, tasks may have regions that are not preemptible. We call these regions *blocking sections*. An example is when the task is holding some critical resources or is in the middle of an atomic transaction. Assuming fully preemptible tasks in this case may cause deadline misses or result in incorrect computation.

In this work, we consider voltage scheduling of real-time periodic tasks with blocking sections. We show how to calculate a static speed by which all admitted tasks can be feasibly scheduled with minimal energy. A dynamic dual speed algorithm is also proposed to run the processor at an even lower speed in some intervals. Furthermore, as the actual execution time usually differs from the declared worst-case execution time (WCET), a reservation-based scheme which dynamically reclaims processor time for redistribution is proposed to further reduce energy consumption.

The rest of the paper is organized as follows: The system models are introduced in Section 2. Section 3 gives the feasibility conditions and formulas for the static speed algorithm. Section 4 presents two dynamic voltage scheduling algorithms. Performance evaluation results are presented in Section 5. Section 6 reviews related research on voltage scheduling. Finally, Section 7 concludes this paper.

2 System Model

2.1 Task Model

Real-time periodic tasks are considered in this paper. A periodic task is a sequence of jobs released at constant intervals (called the period). We denote the set of tasks by T. Each task $T_i \in T$ is characterized by four parameters:

- A_i: time the task is first released.
- D_i: relative deadline of the task.
- P_i: period of the task.
- E_i: worst-case execution time (WCET) of any job in the task.

In this paper, we assume the relative deadline of a task is equal to its period. Each job in a task can be considered as a processing request. It is associated with an absolute deadline by which the job should be completed. We say a job *meets* the deadline if the job is completed at or before before its deadline, and it *misses* the deadline otherwise. In this paper, we assume hard real-time tasks, i.e., there should be no deadline miss. A task $T_i = (J_{i,1}, J_{i,2}, J_{i,3}, \ldots, J_{i,n})$ consists of n jobs, where job $J_{i,j}$ is characterized by its release time $r_{i,j}$, the execution time $e_{i,j} (\leq E_i)$ and the absolute deadline $d_{i,j}$. The execution time is defined as the time required to process the job at the processor's maximum speed. Furthermore, jobs are preemptible except when they are running in their blocking sections. A job can have zero, one or more blocking sections, and G_i denotes the length of the longest blocking section of any job in task T_i[2]. The positions of the blocking sections are randomly distributed within a job except that they are non-overlapping.

2.2 Processor Model

The processor is capable of dynamic voltage scaling and its speed is proportional to the supply voltage. The maximum and minimum possible supply voltages are denoted as $V_{\max}$ and $V_{\min}$, respectively, while the corresponding processor speeds are $S_{\max}$ and $S_{\min}$, respectively. The processor voltage can be adjusted at discrete steps within the range. Throughout this paper, we assume the processor's maximum speed is 1 and all other speeds are normalized with respect to the maximum speed. As observed in [2], the voltage transition delay is very short. We therefore assume the voltage transition cost is negligible and the voltage can be adjusted at any time (whether inside or outside a blocking section). We also assume the processor power follows formula (1), which in our case can be simplified to $P = K \cdot V_s^3$ where K is a constant.

3 Static Blocking-aware Voltage Scheduling

In this section we show how to find a static voltage/speed setting to minimize energy consumption in a system consisting of periodic tasks with blocking sections. In the static scheme, the processor voltage is changed only when a new task arrives or when an existing task terminates.

The Stack Resource Policy (SRP) was proposed by Baker to schedule tasks with shared resources [13]. The core idea is that a job is allowed to preempt a lower priority job only if all the resources it needs are available. The feasibility condition of the SRP was also derived and is listed in Theorem 1:

Theorem 1 *[13] Suppose n periodic tasks are sorted by their periods. They are schedulable by the earliest deadline first (EDF) algorithm with the SRP if*

$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{E_i}{D_i} + \frac{B_k}{D_k} \leq 1,$$

where B_i is the maximum length that a job in T_i can be blocked.

[2]Note that if there are multiple blocking sections in the same job, the total length of these sections can exceed G_i. The maximum size of G_i is the size of the job.

In voltage scheduling, if the SRP is used with EDF [14], the processor speed can be reduced according to Theorem 2. Note that we have replaced D with P in the formula since they have the same value in our task model.

Theorem 2 *Suppose n periodic tasks are sorted by their periods. They can be feasibly scheduled by EDF with the SRP at processor speed $H(0 < H \leq 1)$ if*

$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{E_i}{P_i} + \frac{B_k}{P_k} \leq H,$$

where B_i is the maximum length that a job in T_i can be blocked.

Proof: Note that scheduling a task set T at processor speed H is equivalent to scheduling a task set T^* at the maximum processor speed where the execution times and resource holding times of T^* are $1/H$ times the corresponding values in T. Hence, the above inequality can be rewritten as

$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{E_i \cdot \frac{1}{H}}{P_i} + \frac{B_k \cdot \frac{1}{H}}{P_k} \leq 1.$$

Moreover, the maximum blocking time a job in T_i^* may encounter would also be scaled up by $1/H$ times. According to Theorem 1, T^* is schedulable by the SRP at full processor speed (=1), so the original task set T is schedulable at speed H. □

Our specific task model can be regarded as a special case of the SRP, where only one resource is shared and all tasks need to request the resource to complete. If a job does not actually need the resource, it is still required to request the resource for zero time unit at the start of its execution. Such a job corresponds to a job without a blocking section in our task model. The feasible static processor speed can be calculated according to Corollary 1.

Corollary 1 *Suppose n periodic tasks are sorted by their periods. They can be feasibly scheduled by EDF at processor speed $H(0 < H \leq 1)$ if*

$$\forall k, 1 \leq k \leq n, \sum_{i=1}^{k} \frac{E_i}{P_i} + \frac{\max\{G_j | P_k < P_j\}}{P_k} \leq H, \quad (2)$$

where $1 < j \leq n$.

Proof: Note that in EDF scheduling, a job can only be blocked by jobs in tasks with larger periods, so $\max\{G_j | P_k < P_j\}$ is the maximum time that any job in T_k can be blocked. □

The static processor speed needs to be re-computed only when a task completes or when a new task arrives. The objective is to find a minimum H such that the inequalities are satisfied. If the re-calculated H exceeds 1, the newly arrived task is not admitted to the system and the original H value is restored. Otherwise the system will run at speed H until the value is changed again. For discrete processor speed levels, the lowest speed level that is greater than or equal to H is used.

4 Dynamic Blocking-aware Voltage Scheduling

By applying a feasible static speed as in Section 3, the task set is guaranteed to be schedulable. However, there may exist idle intervals which could be exploited to further reduce the processor speed. The idle intervals come from two sources. First, the feasibility test is based on the WCETs and the maximum blocking lengths of the tasks. The actual execution times and blocking sections of individual jobs are usually shorter. Second and more importantly, given the feasibility condition, the total processor utilization of the admitted tasks is usually lower than the normalized static feasible speed H. This difference becomes substantial as the length of blocking sections increases. In this case, the processor is still under-utilized even if all jobs use their WCETs. However, if a single static speed is used, it is not possible to further reduce it below H without incurring deadline misses. In this section, we propose new algorithms that dynamically slow down the processor at strategic intervals that will preserve the feasibility of the task set.

4.1 A Dual-Speed Switching Algorithm

If the tasks are fully preemptible, they can be feasibly scheduled by EDF with a minimum static speed L [1] such that

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \leq L.$$

Comparing with (2), it is easy to see $L \leq H$. We refer to L as the *utilization speed*, or simply the "low speed". In contrast, the *static speed* H (or "high speed") calculated in Section 3 ensures a job will not miss its deadline even if worst-case blocking occurs. We propose a dual speed algorithm that allows the processor to operate at speed L whenever possible. In the dual speed algorithm, if a job blocks a higher priority job, then during the lifetime of the low priority job (i.e., until its deadline), the processor must run at speed H. In all other situations the processor may run at the low speed L. The Dual Speed (DS) algorithm is formally presented in Figure 1. If the system is executing in a high speed interval (i.e., the system is running at speed H), then End_H in the algorithm indicates the time point at the end of that interval. Otherwise $End\_H = -1$. If there is no work to do, the processor will enter the idle state. Note that

/* H and L are recomputed by the static speed algorithm as a task joins or leaves the system. Initially the processor speed is L. End_H indicates the end of the high-speed interval. If the system is not in a high speed interval, $End\_H = -1$. Initially $End\_H = -1$. */
When job $J_{i,j}$ arrives:
  if Priority($J_{i,j}$) > Priority(current job)
    if Preempt_Current_Job() is successful
      Execute $J_{i,j}$;
    else /*$J_{i,j}$ is blocked*/
      Set_Speed(H); /*Set the processor speed at H*/
      $End\_H = \max(End\_H, d_{current\_job})$;
      /* d is the deadline of the job */
    end if
  end if

When the end of high speed interval is reached:
  $End\_H = -1$;
  Set_Speed(L); /*Set the processor speed at L*/

Figure 1. The dual speed (DS) algorithm.

with discrete speed levels, the Set_Speed function in the algorithm sets the processor speed to the lowest speed level that is greater than or equal to the speed specified in the parameter.

In the dual speed scheme, the high speed and the low speed need to be re-calculated only when a task joins or leaves the system. By maintaining two task lists respectively sorted by the tasks' periods and maximum blocking lengths, the calculation of H and L can be done in $O(n)$ time where n is the number of tasks. After H and L have been calculated, both procedures in Figure 1 only take $O(1)$ time. As H and L are not frequently changed, the overhead of the algorithm is minimal.

An example is given in Figure 2 to illustrate the difference between the static speed algorithm and the dual speed algorithm. The up and down arrows denote the arrival times and deadlines of the jobs, respectively. The white boxes indicate job execution intervals, while the shaded boxes indicate blocking sections. The two jobs in T_1 both need 1 time unit to finish at the high speed H while the job in T_2 needs 4 time units. Suppose the low speed L is half the value of the high speed. Under the static speed algorithm, the processor runs at speed H throughout interval $[0, 6]$ (Figure 2a), while under the dual speed algorithm, the processor runs at speed L before time point 4, at which blocking occurs (Figure 2b). Based on (1), the dual speed algorithm would save 25 percent of energy compared to the case of static speed.

Although the dual speed algorithm reduces the processor speed, the feasibility of the task set is still maintained. The following theorem guarantees that if a task set is schedulable by the static speed algorithm, it is also schedulable by the dual-speed algorithm.

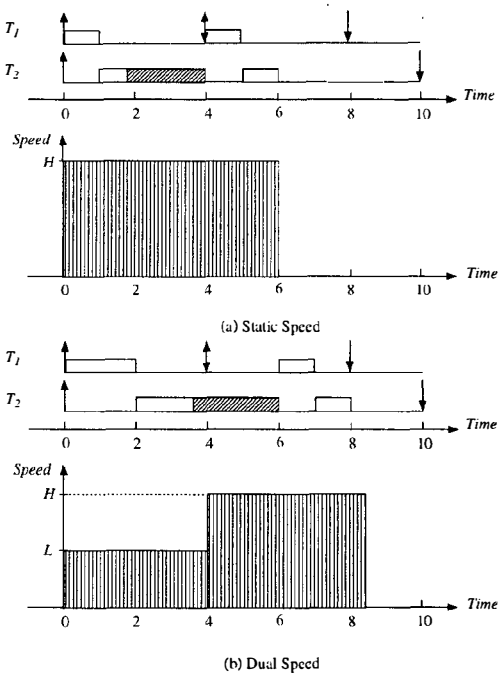

Figure 2. Comparison of static speed and dual speed algorithms.

Theorem 3 *Suppose n periodic tasks with blocking sections are sorted by their periods. They can be feasibly scheduled by the dual speed EDF algorithm with high speed H and low speed L if*

$$\forall k, 1 \le k \le n, \sum_{i=1}^{k} \frac{E_i}{P_i} + \frac{\max\{G_j | P_k < P_j\}}{P_k} \le H \quad (3)$$

and

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \le L. \quad (4)$$

Proof: We prove the theorem by contradiction.

Suppose the claim is false and t is the earliest time that a job misses its deadline. We find another time t' before t which is the latest time point such that no active job arrived before t' has a deadline at or before t. If t' does not exist, we let $t' = 0$. Constructed in this way, the processor is never idle during $(t', t]$ and only two categories of jobs can execute in the interval. The jobs in the first category, denoted by M, are released after t' and have deadlines at or before t. The second category, if exists, consists of a single job, denoted by J_k, which has a deadline after t and is executing in its blocking section at time t'.

We consider two cases: only jobs in M are executed during $(t', t]$, and both J_k and jobs in M are executed during the interval.

Let $X = t - t'$. In the first case, because all tasks are periodic, the processor demand generated by the jobs in M is bounded by $\sum_{i=1}^{n} \lfloor X/P_i \rfloor \cdot E_i$. On the other hand, as the processor is never idle and its speed is greater than or equal to L in the entire interval, the processor demand the processor can handle is at least $L \cdot X$. Since a job misses its deadline at time t by assumption, the total processor demand in the interval must exceed what the processor can handle in the same interval. Therefore

$$\sum_{i=1}^{n} \left\lfloor \frac{X}{P_i} \right\rfloor \cdot E_i > L \cdot X.$$

Since $X/P_i \geq \lfloor X/P_i \rfloor$, we have

$$\sum_{i=1}^{n} \frac{X}{P_i} \cdot E_i \geq \sum_{i=1}^{n} \left\lfloor \frac{X}{P_i} \right\rfloor \cdot E_i > L \cdot X \Rightarrow \sum_{i=1}^{n} \frac{E_i}{P_i} > L,$$

which contradicts with (4).

In the second case, the processor demand during $(t', t]$ is larger than that in the first case due to the execution of J_k. However, since J_k will no longer be executed once it leaves its blocking section, the total time J_k can execute during $(t', t]$ cannot exceed its longest blocking section. Suppose J_k belongs to task T_m. Then the time it can execute is bounded by G_m. Therefore the total processor demand is bounded by $G_m + \sum_{i=1}^{l} \lfloor X/P_i \rfloor \cdot E_i$, where P_l is the longest period that is smaller than or equal to X. As the deadline of J_k is later than t, the processor must have been operating at speed H throughout the whole interval $(t', t]$, therefore the amount of work processed is $X \cdot H$. If there is a deadline miss at time t, the processor demand must be greater than the work processed, i.e.,

$$G_m + \sum_{i=1}^{l} \left\lfloor \frac{X}{P_i} \right\rfloor \cdot E_i > X \cdot H.$$

Since $P_l < P_m \Rightarrow G_m \leq \max\{G_j | P_l < P_j\}$, we have

$$\max\{G_j | P_l < P_j\} + \sum_{i=1}^{l} \frac{X}{P_i} \cdot E_i > X \cdot H.$$

Because $P_l \leq X$, we get

$$\sum_{i=1}^{l} \frac{E_i}{P_i} + \frac{\max\{G_j | P_l < P_j\}}{P_l} > H,$$

which contradicts with (3). □

The basic dual speed algorithm can be extended to further reduce energy consumption. This is achieved by shortening the lengths of high speed intervals. In the extension, a high speed interval can be terminated at once if one of the following conditions occurs: i) A job whose deadline is later than or equal to *End_H* starts execution, ii) The processor becomes idle. Because these two situations can never occur in the middle of the interval $(t', t]$ as shown in the proof of Theorem 3, the feasibility of the task set is not impaired.

4.2 A Dynamic Reclaiming Algorithm

In the dual speed algorithm, the processor speed is always higher than or equal to the utilization speed, which suggests room for further speed reduction. Moreover, the utilization speed is calculated based on the tasks' WCETs, but the actual processing demand is often lower. When a job completes early, the processor would have some idle time. If this portion of time can be redistributed to the other pending jobs, the processor speed can be further reduced. In this section, we present a reservation-based scheme that dynamically collects the residue time from early completions. As the algorithm is an extension to the dual speed algorithm, we call this algorithm the Dual Speed Dynamic Reclaiming (DSDR) algorithm. Aydin et al. [1] proposed a similar approach for a fully preemptible environment, but their algorithm is not applicable when blocking sections are present.

First, we need to define the *run time* of a job and distinguish it from the job's execution time. The run time (denoted by R) can be viewed as a budget assigned to a job. It specifies the wall clock time[3] that can be used to process a job and is consumed as the job executes. The run time of a job has a deadline which is set equal to the job's deadline for reclaiming purpose. The execution time (denoted by E) describes the time needed to complete the job under the maximum processor speed. Given the run time and the execution time of a job, the speed at which the processor should operate is $S = E \cdot S_{\max}/R$, where $S_{\max}$ is the maximum processor speed.

We first introduce a reservation-based extension to the dual speed algorithm in which run time is reserved for each job. In order to maintain the same feasibility conditions as in the dual speed algorithm, we need to allocate each job the same amount of run time as it will get under the dual speed algorithm. We take the processor speed determined by the dual speed algorithm (H or L) as the base speed for a job and allocate run time to each job so it can complete its WCET at the base speed before the run time is depleted. When a job arrives, it is assigned an initial run time assuming speed L, which is the optimal allocation if no task is blocked. Note that the processor is never idle if all jobs execute at their WCETs and at speed L. However, since the base speed cannot be determined until the first time the job is selected for execution, the actual run time of the job may be adjusted at that time. As the job executes, it consumes run time and it must complete before its run time is depleted.

The system maintains a Free Run Time list called the FRT-list to collect the run time not consumed. Similar to the CASH queue in [15], each item in the FRT-list contains two values: the amount of run time and its deadline. The

[3]Note that since the processor speed may change, the run time may not be the same as the processor time.

list is sorted in increasing order of the deadlines. The free run time comes from two sources. The first source is the residue run time when a job completes. The deadline of the reclaimed run time is set equal to the deadline of the completed job. A job may also contribute run time to the FRT-list if it starts to execute (for the first time) in a high speed interval. In this case, because the job's base speed is H, its actual run time assigned could be less than its initial run time allocated when it arrived. The difference in run time is taken from the job and is inserted to the FRT-list. The deadline of this free run time is set to the end of the high speed interval. In this way, DSDR effectively reclaims unused run time for redistribution, which in turn reduces the processor idle time and leads to decreased processor speed.

When a job is scheduled to run, it is eligible to use its own run time as well as the run time in the FRT-list with deadline earlier than or equal to the job's deadline. With the additional free run time, the job can be processed at a lower speed. If the run time in an item of the FRT-list is depleted, the item is removed.

Before we formally present the algorithm, we first introduce the following notations used in the algorithm:

- J_i: the current job of task T_i. (Since at any time each task can only have one job present, no ambiguity is introduced. Correspondingly, d_i indicates the deadline of job J_i.)
- $R_i^r(t)$: the available run time of job J_i at time t.
- $R_i^F(t)$: the run time in the FRT-list that can be used by job J_i at time t.
- $E_i^r(t)$: the worst-case residue execution time of job J_i under the maximum speed $S_{\max}$ at time t.

The core of the DSDR algorithm is given in Figure 3. H, L and End_H in the figure have the same meanings as in the dual speed algorithm. Note that the processor speed is calculated according to the usable run time $R_i^F(t) + R_i^r(t)$ and the worst-case residue execution time $E_i^r(t)$ except when the current job is blocking another job. In the latter case, the processor speed is always set at H to reduce the blocking time. To complete the DSDR algorithm, the following rules are used to update the run time and the worst-case residue execution time of a job, and the run time in the FRT-list:

1. As job J_i executes, it consumes run time at the same speed as wall clock time starting from the front of the FRT-list if $R_i^F(t) > 0$. Otherwise, $R_i^r(t)$ is used. $E_i^r(t)$ is reduced by the processor speed per unit time.
2. When the processor is idle, if $R_i^F(t) > 0$, the run time in the FRT-list is reduced at the same speed as wall clock time.

```
When a new job (J_i) arrives:
    E_i^r(t) = E_i;
    R_i^r(t) = E_i/L;
    if Priority(J_i) > Priority(current job)
        if Preempt_Current_Job() is successful
            Select J_i to run;
        else /* the job is blocked */
            base_speed = H;
            End_H = max(End_H, d_current_job);
            Set_Speed(H);
        end if
    end if

When job J_i is selected to run:
    if J_i is executed for the first time && base_speed == H
    /* first run reclamation */
        Insert_To_FRT(R_i^r(t) - E_i/H, End_H );
        R_i^r(t) = E_i/H;
    end if
    Set_Speed(E_i^r(t) / (R_i^F(t)+R_i^r(t)));
    Execute J_i;

When job J_i completes:
    if R_i^r(t) > 0
        Insert_To_FRT(R_i^r(t) , d_i );
        /*early completion reclamation*/
    end if

When the end of high speed interval is reached:
    End_H = -1;
    base_speed = L;
```

Figure 3. The core DSDR algorithm.

3. If a new task arrives at time t, then the run time of all jobs is set to $E_i^r(t)/H'$, where H' is the newly calculated high speed due to the task arrival. The FRT-list is cleared.

Note that these rules do not need to be carried out at every time unit. Instead, Rule 1 is applied only when the current job (J_i) completes, blocks another job, or is preempted, and Rule 2 is used only if a new job arrives when the processor is idle.

Based on the above discussions, the following lemmas can be proved.

Lemma 1 *When a task set is scheduled by DSDR, no job will deplete its run time before it completes.*

The proof should be clear from the DSDR algorithm (Figure 3) and the three updating rules above. Note that a job may use the run time in the FRT-list before consuming its own allocated run time.

Lemma 2 *Suppose n periodic tasks with blocking sections*

are sorted by their periods. If

$$\forall k, 1 \le k \le n, \sum_{i=1}^{k} \frac{E_i}{P_i} + \frac{\max\{G_j | P_k < P_j\}}{P_k} \le H, \quad (5)$$

where $k < j \le n$, and

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \le L, \quad (6)$$

then when DSDR is used with high speed H and low speed L, the run time of a job is always depleted at or before the job's deadline. Furthermore, the blocks of run time in the FRT-list are also depleted before their deadlines.

Proof: We prove the lemma by contradiction.

Suppose the claim is not true. Let t be the earliest instant that the run time of a job or in the FRT-list is not depleted at its deadline. We choose another time t_1 before t, which is the latest time point such that no pending jobs arrived before t_1 has a deadline at or before t, and the FRT-list does not contain any item whose deadline is at or before t. If such a time point does not exist, let $t_1 = 0$. Constructed in this way, the processor never stops consuming run time throughout the interval $(t_1, t]$. Furthermore, the run time consumed during the interval is either generated after t_1 and has a deadline before t (denoted as run time A), or has a deadline after t (denoted as run time B).

We consider two cases. In the first case, only the run time in A is consumed. Note that run time is only generated on job releases. Let $Y = t - t_1$, the total amount of run time in A is bounded by $\sum_{i=1}^{n} \lfloor Y/P_i \rfloor \cdot E_i/L$. As there is still run time left at time t, the amount of run time in A must be greater than the run time consumed in the interval, which is Y. Therefore

$$\sum_{i=1}^{n} \left\lfloor \frac{Y}{P_i} \right\rfloor \cdot \frac{E_i}{L} > Y,$$

which contradicts with (6).

In the second case, the run time in both A and B is consumed in the interval. We choose a third time t_2 which is the latest time point before t such that the deadline of the run time being consumed at t_2 is after t. Since run time in B is consumed in the interval, t_2 must exist and $t_1 < t_2 < t$. The scenario is illustrated in Figure 4.

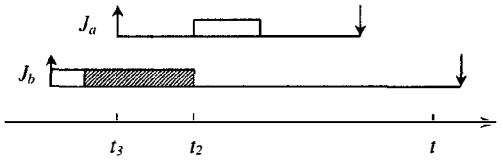

Figure 4. An illustration of when the run time in B is consumed.

Before time t_2, the executing job J_b must be blocking at least one job whose deadline is at or before t. Otherwise J_b would have been preempted before t_2 or the assumption that $t_1 < t_2$ would be violated. Furthermore, there is no run time in the FRT-list at t_2 whose deadline is at or before t, otherwise run time in B would not have been used at time t_2. Let J_a be the first job that is blocked by J_b. We use $t_3 (t_1 \le t_3)$ to denote the arrival time of J_a. In the interval $(t_3, t_2]$, the processor does not consume the run time generated during $(t_3, t]$, but in the interval $(t_2, t]$ it only consumes the run time generated in the period $(t_3, t]$. Suppose $Z = t - t_3$. Since the base speed throughout $(t_3, t]$ is H, the amount of run time generated after t_3 which has a deadline at or before t is bounded by $\sum_{i=1}^{l} \lfloor Z/P_i \rfloor \cdot E_i/H$. In the above expression, P_l is the maximum period that is smaller than or equal to Z. Moreover, as the processor is operating at speed H throughout $(t_3, t_2]$, the total amount of run time that can be consumed during $(t_3, t]$ is therefore bounded by $G_b/H + \sum_{i=1}^{l} \lfloor Z/P_i \rfloor \cdot E_i/H$, where G_b is the length of the longest blocking section in J_b. As there is still residue run time at time t whose deadline is at t and the processor keeps consuming run time throughout the period, we have

$$\frac{G_b}{H} + \sum_{i=1}^{l} \left\lfloor \frac{Z}{P_i} \right\rfloor \cdot \frac{E_i}{H} > Z. \quad (7)$$

Because $Z \ge P_l$ and $Z/P_i \ge \lfloor Z/P_i \rfloor$, (7) can be rewritten as

$$\frac{G_b}{P_l} + \sum_{i=1}^{l} \frac{E_i}{P_i} > H,$$

which contradicts with (5). □

Theorem 4 *Suppose n periodic tasks with blocking sections are sorted by their periods. They can be feasibly scheduled by DSDR with high speed H and low speed L if*

$$\forall k, 1 \le k \le n, \sum_{i=1}^{k} \frac{E_i}{P_i} + \frac{\max\{G_j | P_k < P_j\}}{P_k} \le H,$$

where $k < j \le n$, and

$$\sum_{i=1}^{n} \frac{E_i}{P_i} \le L.$$

Proof: As a job's deadline is the same as the deadline of its run time, the claim directly follows from Lemma 1 and Lemma 2. □

Similar to the extension of the dual speed algorithm, the high speed intervals in DSDR can also be shortened. If the deadline of the run time being consumed is later than or

equal to the end of the current high speed interval, the high speed interval is terminated at once. Note that this extension does not affect the proof of Lemma 2, so the conclusions in Lemma 2 and Theorem 4 are still valid.

5 Performance Evaluation

Simulation experiments were carried out to evaluate the effectiveness of the proposed algorithms in saving energy. In this section, we first describe the assumptions used and the characteristics of the task sets. The simulation results are then presented and analyzed.

5.1 Simulation Setup

The event-driven simulator imitates the processing of periodic tasks on a single processor that is capable of voltage/speed scheduling. We assume a maximum processor speed (=1) and a minimum processor speed (=0.1). Speed levels between the two bounds are discrete and spaced by 0.1. The supply voltage is proportional to the processor speed and the power consumption follows formula (1). We also assume the processor does not consume power when it is in the idle state. The simulation experiments consisted of three phases: task generation, admission control, job scheduling and execution.

To simulate the mixed workload in real systems, we generated tasks whose periods belonged to one of three ranges: long period (1000~5000ms), middle period (100~1000ms) and short period (20~100ms). The WCETs of the tasks in the three categories were (1~1000ms), (1~100ms) and (1~20ms), respectively. The tasks were uniformly distributed in these categories. Within each category, the tasks' periods and WCETs were randomly selected from the corresponding ranges. After the task set was generated, the WCETs of the tasks were scaled such that the total processor utilization would not exceed the desired value, which was specified by the utilization factor. The blocking factor specified the maximum percentage the blocking section could occupy in a job's execution time, so the maximum length of any blocking section of a task was $WCET \times blocking\_factor$.

All tasks generated must undergo an admission control procedure in the order of their released time. We used the formula in Section 3 to calculate a static processor speed H. If the required speed was smaller than the maximum processor speed, the new task was admitted; otherwise the task was dropped without processing.

Finally, the admitted tasks periodically generated jobs. The slack factor specified the difference between the actual job execution time and the task's WCET. Specifically, the execution times of the jobs in a task were uniformly distributed between the task's WCET and $(1 - slack\_factor) \times WCET$. Each job could at most have one blocking section, and we used $blocking\_prob$ to represent the probability that a job would be assigned a blocking section. The position of the blocking section was randomly chosen. Released jobs were stored in the job queue and processed in the EDF order. The voltage scheduling algorithms presented in this paper were used to determine the processor speed for the current executing job.

5.2 Experimental Results

Extensive simulations were conducted with the three proposed algorithms[4]. In each experiment, we generated a task set of 30 tasks. All tasks were released at time 0. The experiment was carried out for 100,000ms and the energy consumption was recorded. In order to improve accuracy, we performed simulations on 10 distinct task sets for each set of parameters and took the average. The static speed algorithm was used as the baseline and the power consumptions of the other two algorithms were normalized with the baseline.

Blocking Parameters

In the first set of simulations, we evaluated the effect of the blocking sections on power consumption. The blocking parameters $blocking\_prob$ and $blocking\_factor$ were varied to produce task sets with different numbers of blocking sections and different maximum blocking lengths. In these experiments, we let all jobs execute at their WCETs. The performance would improve if the actual execution time is less.

We first fixed $blocking\_prob$ to 0.5 and varied $blocking\_factor$ between 0 and 0.3. The range of blocking factor was chosen based on two reasons: i) The blocking sections are usually short in practical applications. ii) As the blocking factor exceeded 0.3, the number of tasks not admitted increased quickly, which affected the accuracy of the results. Figure 5 shows the normalized energy consumption of the dual speed algorithm and the DSDR algorithm under utilization factors 0.4 and 0.6, respectively. As the maximum blocking length increased with blocking factor, the value of high speed H was also increased. The energy consumed by all three algorithms grew with H, but the energy consumption of the two dynamic algorithms grew slower because these two algorithms switched to low speed modes in some intervals. As a result, the dual speed algorithm saved up to 70 percent of the energy consumed under the static algorithm. Since DSDR utilized the run time reclaimed in high-speed intervals, it saved more energy. However, the difference is not significant. As observed in the experiments, blocking only occurred rarely and the durations

[4]Only the results of the extended rather than the basic dual speed algorithm and the basic DSDR were presented since the extended algorithms always outperformed the basic algorithms in all cases.

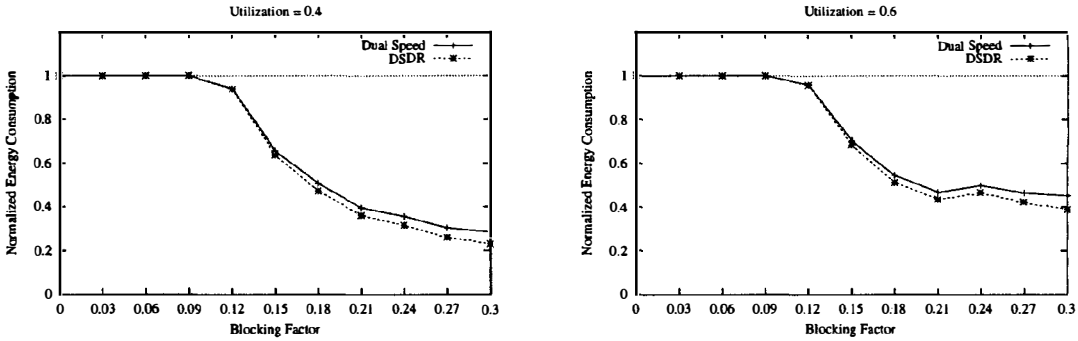

Figure 5. Normalized energy consumption with varying blocking factor.

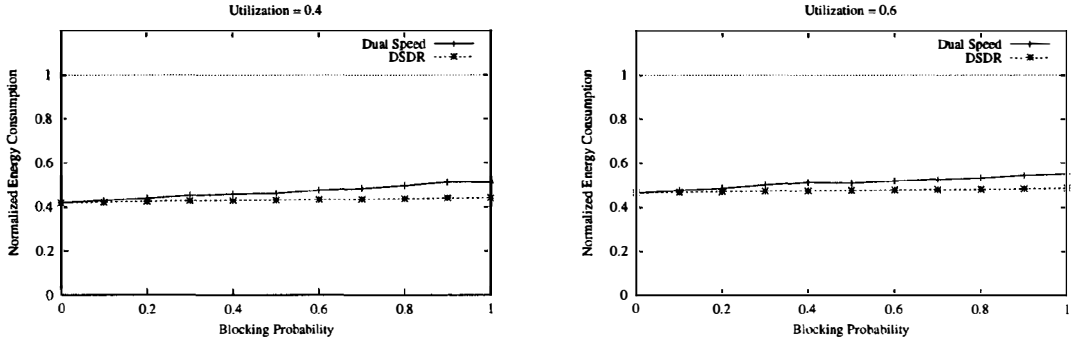

Figure 6. Normalized energy consumption with varying blocking probability.

of high speed intervals were very short, which left very little room for DSDR to reclaim. Another observation is that when the blocking factor was smaller than 0.09, the discrete values of high and low speeds coincided, consequently all three algorithms consumed the same amount of energy.

In the similar way, we have varied *blocking_prob* with *blocking_factor* fixed at 0.2. As shown in Figure 6, the energy consumed by the dual speed algorithm increased with the blocking probability. However, the observed actual blocking rate was very low, so the normalized energy consumption was only increased by less than 10 percent. In DSDR, as a job executing in a high speed interval can use the run time in the FRT-list to compensate for its run time loss due to run time reclamation (see Section 4.2), the impact on energy consumption was even less (less than 4 percent).

Slack Factor

In this set of simulations, we fixed *blocking_factor* and *blocking_prob* to 0.2 and 0.5, respectively. Figure 7 shows the simulation results for slack factor between 0 and 0.9. The dual speed algorithm was insensitive to the variation of the slack factor. Again, DSDR far outperformed the dual speed algorithm due to its ability to reclaim unused run time from early completions. For example, at utilization 0.4 and slack factor 0.5, DSDR saved 70 percent while the dual speed algorithm saved only about 50 percent of the power consumed under the static speed algorithm.

Processor Utilization

Simulations experiments were also performed at different system load levels. Again we fixed *blocking_factor* and *blocking_prob* at 0.2 and 0.5, respectively. Two sets of experiments were carried out with slack factor set at 0.5 and 0.8, respectively. The utilization factor was varied between 0.1 and 1.0 (see Figure 8). The normalized energy consumptions remained relatively constant when the utilization was low but grew as the utilization exceeded 0.6. It turned out the growth was due to task dropping in the admission control phase. Tasks with large blocking sections or small periods were more likely to be dropped since they would significantly increase H. As a result, $H - L$ began to shrink, which reduced the energy saving of the dynamic algorithms. In all cases, DSDR consistently outperformed the dual speed algorithm.

In summary, both dynamic voltage scheduling algorithms outperformed the static speed algorithm under all parameter settings. Taking advantage of the dynamic reclaiming mechanism, DSDR saved even more energy than the dual speed algorithm, especially when the actual execution times were less than their WCETs.

6 Related Work

Since the seminal paper by Weiser et al. [12], much work on voltage scheduling has been published. The work in

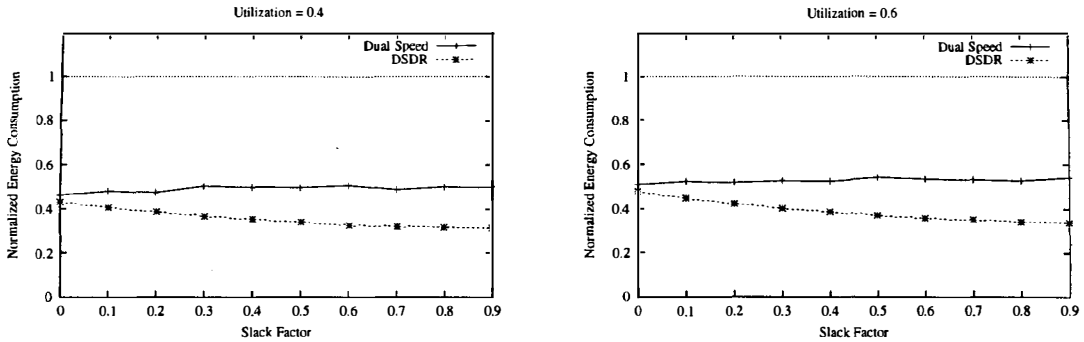

Figure 7. Normalized energy consumption with varying slack factor.

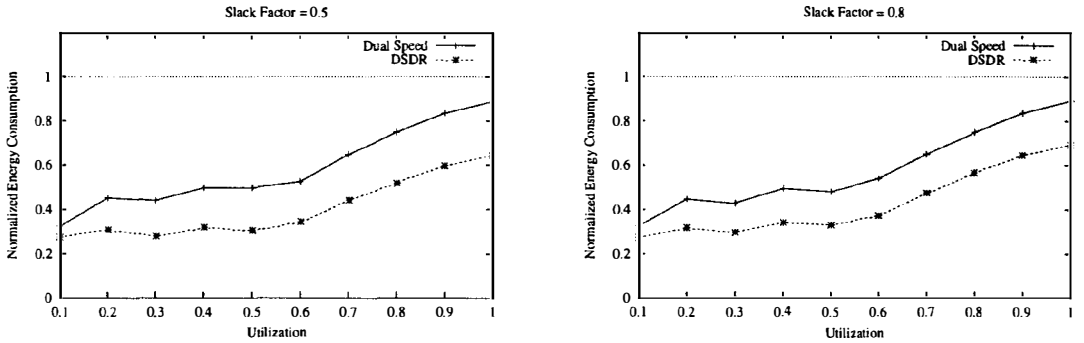

Figure 8. Normalized energy consumption with varying processor utilization.

this area can be classified into two categories: interval-based scheduling [3, 10, 12] and profile-based scheduling [1, 2, 9, 11].

Interval-based scheduling estimates the processor utilization based on observations in the past interval(s). For example, the PAST algorithm records the processor utilization in the previous interval. If the utilization exceeded the upper threshold, the supply voltage is incremented; if the utilization was below the lower threshold, the supply voltage is decremented [12]. The weighted-AVG algorithm [10] makes use of the processor utilization in the previous interval as well as the average utilization in all past intervals. Thus prediction is made based on both short-term and long-term system behaviors. Simulations using real-life traces were also carried out to evaluate the performance of these algorithms. More recently, Lorch et al. [3] proposed to use distributions to estimate the processing requirements of future tasks from the recent behaviors of similar tasks. They suggested using the gamma model for its low complexity and good predicating capability.

Interval-based algorithms assume the workload is more or less stable (or follows some distributions) so that the behaviors of future tasks can be accurately estimated based on past observations. The accuracy in predicting future tasks significantly affects possible energy reduction. This kind of strategies is not suitable for real-time tasks. As time constraints are not considered, the algorithms may improperly lower the processor speed and cause deadline misses, especially when the execution demands vary greatly from job to job.

Profile-based scheduling is usually used to schedule real-time tasks. Some knowledge of the released jobs, and to some extent of the future jobs in the current task set is supposed to be known. Yao et al. [11] derived an optimal offline scheduling algorithm for aperiodic real-time tasks. An online approximation algorithm was also presented in the same paper. Hong et al. [9] used a similar planning-based approach to handle non-preemptible tasks. The traditional real-time periodic task model is used in some recent work. Pillai and Shin [2] studied both fixed priority and dynamic priority scheduling and proposed schemes under the rate-monotonic (RM) algorithm and the earliest-deadline-first (EDF) algorithm individually. Their look-ahead algorithm takes the processing requirements of future jobs into consideration and delay processing of these jobs as much as possible. Aydin et al. [1] proved there is an optimal static voltage that can feasibly schedule periodic tasks with minimum power consumption. They also proposed a dynamic algorithm which uses the slacks of the jobs that complete early.

All of the above work assumed that tasks were either fully preemptible or completely non-preemptible. Scheduling tasks with blocking sections usually require a higher processor speed than the speed needed for fully preemptible

tasks (i.e., the utilization speed). Our approaches utilize this fact and allow the processor to operate at the utilization speed or lower at strategic intervals without impairing the feasibility of the task set.

7 Conclusion

In this paper, we have investigated voltage scheduling of real-time periodic tasks with non-preemptible blocking sections. Three voltage scheduling schemes have been proposed to minimize energy consumption while satisfying the tasks' time constraints. The static speed scheme is based on the stack resource policy (SRP) [13] and calculates a minimal feasibly static speed. Instead of always operating at one static speed, the dual speed algorithm lowers the processor speed to the utilization speed in some intervals. We have also presented a reservation-based scheme which reserves run time for each job. A reclaiming mechanism is used to collect the unused run time and redistribute it to jobs that are able to make use of it. As the jobs that receive extra run time are eligible to run for a longer period of time, the processor speed can be further reduced to save energy.

Feasibility conditions have been derived for the three algorithms and proved mathematically. Simulation experiments were carried out to evaluate the performance in energy saving. The results show that both dynamic algorithms can significantly reduce energy consumption compared with the static speed algorithm in all scenarios.

8 Acknowledgements

The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China.

References

[1] H. Aydin, R. Melhem, D. Mosse and P. Mejia-Alvarez. "Dynamic and aggressive scheduling techniques for power-aware real-time systems". In *Proceedings of IEEE Real-Time Systems Symposium*, pages 95–105, 2001.

[2] P. Pillai and K. G. Shin. "Real-time dynamic voltage scaling for low-power embedded operating systems". In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102, 2001.

[3] J. R. Lorch and A. J. Smith. "Improving dynamic voltage scaling algorithms with PACE". In *Proceedings of SIGMETRICS*, pages 50–61, 2001.

[4] J.R. Lorch and A.J. Smith. "Software strategies for portable computer energy management". *IEEE Personal Communications*, 5(3):60–73, 1998.

[5] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. "Quantifying the energy consumption of a pocket computer and a java virtual machine". In *Proceedings of SIGMETRICS*, pages 252–263, 2000.

[6] http://www.transmeta.com.

[7] http://www.intel.com.

[8] T. Burd and R. W. Brodersen. *Energy Efficient Microprocessor Design*. Kluwer Academic Publishers, 2002.

[9] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M.B. Srivastava. "Power optimization of variable-voltage core-based systems". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(12):1702–1714, 1999.

[10] T. Pering, T. Burd, and R. W. Brodersen. "The simulation and evaluation of dynamic voltage scaling algorithms". In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 76–81, 1998.

[11] F. Yao, A. Demers and S. Shenker. "A scheduling model for reduced cpu energy". In *Proceedings of IEEE Annual Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[12] M. Weiser, B. Welch, A. Demers and S. Shenker. "Scheduling for reduced cpu energy". In *Proceedings of Symposium on Operating system Design and Implementation (OSDI)*, pages 13–23, 1994.

[13] T.P. Baker. "Stack-based scheduling of real-time processes". In *Advances in Real-Time Systems*. IEEE Computer Society Press, 1993.

[14] C. L. Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". *Journal of the ACM*, 20(1):46–61, 1973.

[15] M. Caccamo, G. Buttazzo, and Lui Sha. "Capacity sharing for overrun control". In *Proceedings of IEEE Real-Time Systems Symposium*, pages 295 –304, 2000.