Fixed-Priority Scheduling for Reducing Both the Dynamic and Leakage Energy on Variable Voltage Processors

Gang Quan Linwei Niu
Dept. of CSE
University of South Carolina
Columbia, SC 29208
{gquan, niul}@cse.sc.edu

Bren Mochocki Xiaobo Sharon Hu
Dept. of CSE
University of Notre Dame
Notre Dame, IN 46556
{bmochock,shu}@cse.nd.edu

Abstract

With ever-scaling VLSI technology, the leakage is increasingly becoming a serious concern when addressing the power consumption problem for next-generation real-time embedded systems. Dynamic Voltage Scaling (DVS) is efficient in reducing the dynamic energy consumption of a CMOS processor. However, methods that employ DVS without considering the leakage current are quickly becoming less effective to reduce the processor's overall energy consumption. To be overall energy efficient, the processor may have to run at a higher-than-necessary speed, which will cause a large number of idle intervals. While the processor can be shut down during these idle intervals to save energy, this process may incur significant timing and energy overhead. In this paper, we propose a DVS scheduling technique for fixed-priority hard real-time systems that can judiciously merge the short, scattered idle intervals into longer ones to reduce the shut-down overhead. The proposed technique has very low on-line computation complexity and can be readily incorporated with a variety of DVS scheduling techniques. Experimental results demonstrate that proposed technique can significantly reduce the number of idle intervals and the overall energy consumption than conventional scheduling techniques.

1 Introduction

Power consumption has become one of the primary design issues of next-generation portable, scalable and pervasive embedded systems. For CMOS circuits, power consumption includes dynamic power and leakage power. Dynamic power is due to the switching activities of the transistors, and leakage power is consumed when the sub-threshold current flows through the transistors. Current power saving techniques mainly focus on reducing dynamic power, because it has traditionally been the dominant component in the overall power consumption for most embedded systems today. However, as VLSI technology continues to evolve towards deep sub-micron and nanoscale circuits operating at multi-GHz frequencies, the rapidly elevated leakage power dissipation will soon become comparable to, if not exceed, the dynamic power consumption [13]. More advanced techniques are required for the development of future generations of low-power embedded systems.

Facing the increasing challenges presented by leakage power consumption, design efforts on all fronts must be pursued to form an integrated solution for this problem. Recently, many circuit and architecture techniques, such as those presented in [4, 8, 16, 27], have been proposed to control the leakage power. For a more comprehensive survey on the circuit and architecture level techniques for leakage reduction, readers can refer to the recent publications [5, 32]. It has been demonstrated [7, 25, 38] that reducing both the dynamic and leakage power consumption simultaneously is critical for an overall energy-efficient design. It is our belief that real-time scheduling plays a unique role in this integrated effort, not only because a large percentage of future embedded systems will be real-

time, but also because real-time scheduling has proven to be one of the most effective ways of reducing power consumption.

Dynamic Voltage Scaling (DVS) can effectively reduce dynamic power consumption in real-time systems, and extensive DVS-based real-time scheduling techniques, *e.g.* [3,11,29,37] have been proposed. DVS works by varying the processor's supply voltage and frequency during runtime to match workload and deadline requirements. However, the energy savings achievable via voltage reduction is becoming severely limited due to the dramatic increase in leakage power consumption [13]. Using DVS alone with no consideration of leakage power consumption may actually increase the total energy consumption. This is because DVS tends to make the processor speed as low as possible to minimize dynamic power. Unfortunately, as shown by Irani *et. al.* [10], to be overall energy efficient, the processor may have to run at a higher-than-necessary speed, since a low processor speed (supply voltage) increases the active period of the processor, which in turn increases the leakage energy consumption to a degree that can offset or even surpass the dynamic energy reduction.

In this paper, we present a leakage conscious scheduling approach that combines both the DVS and shut-down strategies [1] for hard real-time systems, scheduled by the fixed priority (FP) policy (such as the rate monotonic scheduling (RMS) policy [23].) Running processor at a higher-than-necessary speed can produce a large number of scattered idle intervals. While it is desirable to shut down the processor or put the processor in a low-leakage mode when idle, the significant energy overhead associated with a large number of processor shut-downs and wake-ups will make the system less energy-efficient. Moreover, considering the timing overhead, the processor simply cannot be put to the low-leakage mode if the idle interval is not long enough. In this regard, we present efficient techniques to delay the execution of tasks and merge the scattered idle intervals, thus greatly reducing the processor shut-down overhead. The proposed technique has a very low on-line computation cost. Using a processor model with projected 0.07μs technology [25], our experimental results show that the proposed method can significantly reduce the shut-down overhead by merging the idle intervals, and it is particularly effective in reducing overall energy when the workload of the system is relatively low. When the system workload is relatively high or when processor shut-down timing overhead is significant, however, our experiments did show that traditional DVS continues to be an effective way to reduce the total energy consumption.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 introduces preliminaries related to our problem. Section 4 discusses our general leakage conscious DVS approach. Section 5 introduces in details the techniques to delay the execution of real-time jobs such that idle intervals can be merged. Section 6 demonstrates the effectiveness of our approach based on simulations. Section 7 concludes the paper.

2 Related Work

Previously, there have been extensive research (e.g. [3, 11, 29, 37]) on applying real-time scheduling methods to reduce power/energy consumption. Most of these scheduling techniques are focused on reducing the dynamic power/energy consumption. Recently, there have been increasing research efforts [14, 15, 20, 36] that use real-time scheduling approach to control the leakage and reduce the overall power consumption for real-time embedded systems. Yan *et. al.* [36] proposed a scheduling algorithm to reduce both dynamic and leakage power based on a processor that has been furnished with both DVS and adaptive body biasing (ABB) features. An analytic power model is derived that can be applied to compute the optimal supply voltage and body bias voltage in terms of overall energy reduction for a given clock frequency. This approach is only feasible when the supply voltage and body biasing voltage are continuous variables. For processors with discrete levels of supply voltage and body biasing voltage, Andrei *et. al.* [2] proved it is an NP-hard problem and provided a non-linear mathematic programming technique to solve this problem. However, this approach cannot be applied for priority-based preemptive scheduling schemes employed in many real applications [24]. Under the assumption that the processor shut-down overhead can be infinitely small, Irani *et. al.* [10] proposed a theoretical "optimal" voltage schedule with leakage consideration which can be constructed from the traditional optimal DVS schedule. Their work forms the basis of our heuristic techniques.

[1] By *shut down*, we mean to either literally shut down the processor or put the processor in a low leakage mode.

For a processor with no complex ABB control mechanism, putting the processor into a low leakage mode when it is idle is one of the most effective strategies to reduce the leakage power consumption. However, care must be taken as entering and exiting the low-leakage state can incur significant timing and energy overhead [8]. For real-time tasks scheduled according to EDF policy, Lee *et. al.* [20] proposed a leakage reduction scheduling technique called **LC-EDF**, which delays the execution of the arriving task instances when the processor is idle to extend the idle intervals and reduce the number of power mode transitions. Specifically, according to **LC-EDF**, the extended idle time is treated as one part of the tasks' execution time. As long as the resultant total utilization is less than or equal to 1, the schedulability of the task set is guaranteed. However, they assume a non-DVS processor model, which cannot optimize the dynamic power consumption. So the overall energy consumed cannot be minimized. To save both the dynamic and leakage energy, Jejurikar *et. al.* [15] exploited a strategy that combines both DVS and procrastination for periodic tasks. Each task is associated with a processor speed and a procrastination value, computed based on the utilization-related feasibility condition. In the same way as **LC-EDF**, the idle interval is extended by procrastinating the execution of tasks when the processor is idle. This approach works very well for periodic task sets when the tasks' deadlines are equal to their periods. However, when the tasks are non-periodic or have deadlines less than their periods, this scheme becomes invalid or very pessimistic in determining the processor speeds and delay amount, and thus energy inefficient.

A number of papers extend the idea of delaying the execution of job executions for real-time systems that employ scheduling policies other than the EDF policy. Lee *et. al.* [20] proposed a leakage reduction scheduling technique called **LC-DP**, by extending the Dual-Priority (DP) scheduling model presented in [6]. The DP scheduling was initially proposed to improve the response time for soft real-time tasks by delaying the execution of hard deadline tasks. (The detailed algorithm and introduction of DP can be found in [6].) In **LC-DP**, idle time is treated as a soft real-time task in the DP model. The hard deadline tasks are originally put in the queue that has a lower priority than that of the soft real-time task if the processor is idle, and are promoted to the higher priority queue after certain *promotion times* have passed. If the processor is active, all the hard deadline tasks are promoted to the high priority queue immediately. The idle interval is extended since the hard real-time tasks are delayed at the lower priority queue. However, as explained in [14], the LC-DP algorithm cannot guarantee FP-tasks' deadlines because of its discrepancy with the original dual priority scheduling algorithm [6].

When real-time tasks are scheduled according to the FP scheme, Jejurikar *et. al.* [14] proposed delaying the execution of tasks by the minimal promotion time over all lower and equal priority tasks. However, as shown later in this paper(see Figure 2), this approach still *cannot* guarantee the schedulability of the tasks under the FP scheme. Lehoczky and Ramos-thuel [21] proposed a technique, called *slack stealer*, that can delay the FP jobs as late as possible and thus minimize the response times for aperiodic tasks. Specifically, they adopted the exact timing analysis strategy, which potentially can have a very high computational cost, to compute the maximal delay of the periodic tasks. In addition, note that it becomes extremely challenging if not totally impossible to employ the exact timing analysis strategy to deal with the scenario when different instances of the same task have different worst case execution times, i.e., when different jobs need to run at different processor speeds to maximize the energy savings as in our case. Kim et al. [17] proposed another strategy to delay the execution of FP jobs with the goal to reduce the preemption times in DVS scheduling. The rationale of this technique is quite similar to the Look-Ahead RT-DVS strategy [29, 33], that is, to delay the higher priority jobs until the absolute last moment, i.e., when the delayed tasks can only meet their deadlines by using the highest possible processor speed. The side effect of this technique is that it may increase the processor speed and therefore compromise the energy saving performance.

In this paper, we develop a novel technique to delay the FP job executions. Our goal is to merge the idle intervals by delaying the jobs while guaranteeing their deadlines. We achieve this goal by judiciously computing the latest starting time of a task without varying its speed which is pre-determined to optimize the dynamic and leakage energy consumption.

3 Preliminaries

In this section, we first introduce the real-time system and power model considered in this paper. We then present an example to motivate our approach.

3.1 System model

We consider a general system model that consists of N jobs, denoted by $\mathcal{J} = \{J_1, J_2, \cdots J_N\}$. Each individual job is denoted by $J_i = (r_i, c_i, d_i)$, where r_i, c_i, and d_i are the arrival time, worst case execution cycles, and absolute deadline for the job, respectively. The job set is scheduled using an FP scheme. Without loss of generality, we assume that J_i has a higher priority than J_k if $i < k$. When a real-time system is described by a set of periodic tasks, where each task instance represents one job, we assume that it is sufficient to schedule the set of jobs produced up until the Least Common Multiple (LCM) of the periods of all tasks.

3.2 Power model

In a CMOS circuit, the power consumption includes both dynamic and static components during its active operation [32]. The dynamic power consumption (P_{dyn}) mainly consists of the switching power for charging and discharging the load capacitance, and the short circuit power due to the non-zero rising and falling time of the input and output signals. The dynamic power (P_{dyn}) can be represented as

$$P_{dyn} = \alpha C_L V^2 f, \tag{1}$$

where α is the switching activity factor, C_L is the load capacitance, V is the supply voltage, and f is the system clock frequency. The static power (P_{leak}) can be expressed as

$$P_{leak} = I_{leak} V, \tag{2}$$

where I_{leak} is the leakage current, which consists of both the sub-threshold leakage current and the reverse bias junction current in the CMOS circuit. Leakage current increases rapidly with the scaling of devices and becomes particularly significant with the reduction of the threshold voltage [38]. Therefore, leakage power consumption is becoming a major part of the the active power consumption ($P_{act} = P_{dyn} + P_{leak}$) in future CMOS circuits with low supply voltage and high transistor density.

The processor consumes energy not only in its active mode but also when it is idle. When idle, the major portion of power consumption comes from leakage. With dramatically increasing leakage current as VLSI technology continue its evolution, it is imperative that this portion of energy be effectively reduced for the overall system-energy reduction. Processor shut-down, i.e., putting the processor into a "sleep mode", can greatly reduce the energy consumption when the processor is idle. For example, it has been reported in [9] that the power dissipation when the processor is idle can be three orders of magnitude higher than that when the processor is shut down.

While the processor consumes less power in the power down state, it costs extra energy and time to shut down and later wake up the processor in order to save/restore the context, as well as initiate the architectural components such as the cache, translation look aside buffers, and branch target buffers. One has to be careful when shutting down the processor since energy overhead may outweigh the benefit of energy savings if the idle interval is too short. Assume that the power consumption of a processor in its idle state and sleeping state are P_{idle} and P_{sleep}, respectively, the energy overhead of shutdown/wakeup is E_o and the timing overhead is t_o. The processor can be shut down with positive energy savings only when

$$P_{idle} \times t \geq E_o + P_{sleep} \times (t - t_0), \tag{3}$$

i.e., the length of the idle interval must be larger than $T_{min} = \max(\frac{E_o}{P_{idle} - P_{sleep}}, t_o)$. We call T_{min} the *minimal length of the idle interval*.

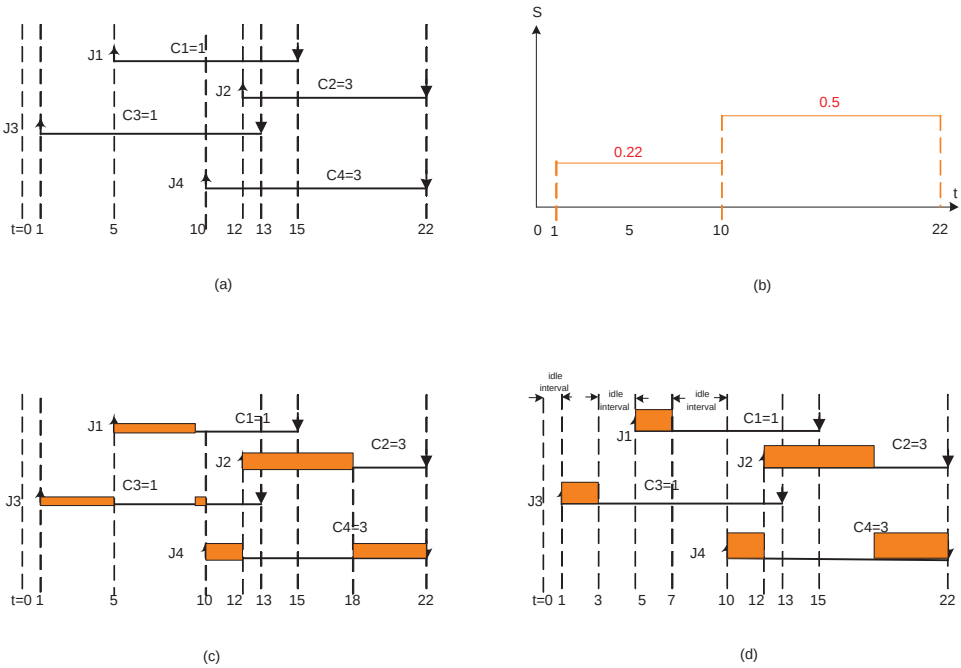


Figure 1. (a) A job set that consists of four jobs. (b) The voltage schedule that can reduce dynamic power consumption. (c) The actual executions of the jobs according to the voltage schedule shown in *(b)*. (d) Applying the threshold speed ($s_{th} = 0.5$) results in the scattered idle intervals.

3.3 A motivation example

As an illustrative example, Figure 1(a) shows a job set with four jobs (The upper and down arrows represent the arrival times and deadlines of jobs, respectively.) Figure 1(b) is the voltage schedule according to the DVS scheduling technique presented in [31], and Figure 1(c) shows the actual executions of the jobs based on the voltage schedule from Figure 1(b).

As indicated in equation (1), the dynamic energy consumption is quadratically related to the supplied voltage. Therefore, traditional DVS scheduling techniques (e.g. [31]) try to reduce the supply voltage to as low a level as possible (see the voltage schedule shown in Figure 1(b) and (c)). However, such a voltage schedule may not be always feasible and/or overall energy efficient due to the following reasons. First, practical processors have a minimal voltage supply limitation, which makes desirable processor speed not possible. Second, commercial processors usually provide only a discrete set of voltages. This means the processor will likely not be able to run at a speed selected by a particular DVS algorithm. Instead, the desired speed needs to be rounded up to the next discrete speed that is available[2]. Furthermore, even when a low processor speed is available, the rapidly increased leakage current may increase the static power consumption to the extent of over-weighing the dynamic power consumption. Therefore, to achieve the best energy efficiency, the processor speed must be determined in a cooperative manner with both dynamic and static energy consumption in mind.

Consider a job with workload w. Let the total power of a processor during its active mode be $P_{act}(s)$. Then the total energy, i.e., $E_{act}(s)$, consumed to finish this job with speed s, can be represented as

$$E_{act}(s) = P_{act}(s) \times \frac{w}{s}. \tag{4}$$

[2]While it is possible to use two discrete speeds immediate above and below the desired speed value to optimally schedule a single job [12, 19], this method can induce a significant transition overhead to the scheduling process, i.e., one extra transition per job.

Hence, to minimize the $E_{act}(s)$ in equation (4), let $\frac{dE_{act}(s)}{ds} = 0$ and thus

$$P_{act}(s) = P'_{act}(s)s. \tag{5}$$

Equation (5) computes the most energy efficient speed to finish one job. We call this speed the *threshold speed*[3], and denote it as s_{th}. To increase or decrease the processor speed from s_{th} will increase either the dynamic or static power, and thus the total active power consumption for executing the job.

Note that, while it is desirable to execute a job using the threshold speed to minimize the active power consumption, it is not always feasible to do so when considering the deadlines and the preemption effects among jobs. Given a voltage schedule, a job that is required to run at a speed higher than s_{th} must be executed with that speed to guarantee the schedulability of the job set. For jobs having required speeds lower than s_{th}, they can be executed at s_{th} to conserve energy. Figure 1(d) shows the scheduling results with $s_{th} = 0.5$.

Using s_{th} for jobs with speed requirements lower than s_{th} while maintaining the speeds of the rest certainly guarantees all deadlines. The problem is that, as shown in Figure 1(d), such a voltage schedule can result in a large number of scattered idle intervals. Though using a processor shut-down strategy is the most efficient method to reduce the energy consumption for these intervals, too many shut-downs will incur significant energy overhead. Moreover, using a processor power down strategy is not always feasible or necessarily energy efficient if the idle interval is not long enough. Unless we can effectively deal with the idle intervals in the schedule, we cannot achieve our ultimate goal of maximizing the overall energy-saving performance of the system. In what follows, we introduce our approach to save the idle energy when scheduling a real-time task set by effectively clustering the idle intervals.

4 The General Approach

The shut-down strategy favors longer idle intervals. To extend an idle interval, one can always increase the processor speed so that each job is executed faster. However, as shown in equation (5), increasing the speed over s_{th} will increase the dynamic power consumption. A better approach, as suggested in [10, 15, 20], would be one that extends the interval lengths by delaying the executions of the incoming jobs, *i.e.*, a job is executed as soon as possible when the processor is not idle, but delayed as much as possible when the processor is idle.

Delaying job executions helps to merge scattered, short idle intervals into longer ones. More energy can be saved because energy overhead incurred by frequently entering and leaving the power-down state is reduced. Moreover, intervals that were previously shorter than T_{min} can now be shut down. As mentioned before, the power dissipation when the processor is idle can be 10^3 times higher than that when the processor is shut down. Therefore, merging short idle intervals has the potential of significantly reduce the overall energy consumption.

To facilitate a clear explanation of our approach, we first introduce the following definition.

Definition 1 *Let each job in the job set ($\mathcal{J}$) be executed with its pre-determined, feasible execution speed. The* **latest starting time for a job set**, *e.g. $\mathcal{J}$, (denoted as $LST(\mathcal{J})$) is the latest time such that, if the execution of any job in $\mathcal{J}$ starts no later than $LST(\mathcal{J})$, all jobs can meet their deadlines.*

Algorithm 1 sketches the general framework of our approach. When the processor is not idle, it will run the jobs in the ready queue according to the FP DVS schedule. The DVS schedule is computed off-line, using an algorithm similar to the one presented in [31]. Other on-line DVS techniques that can dynamically reclaim system resources (such as [18, 34]) can be readily incorporated into this algorithm. The only variation when applying these techniques is to use the threshold speed (s_{th}) if a designated processor speed is less than s_{th}—since using speeds less than s_{th} will increase the total active energy—and dynamically compute the LST. The key to the success of Algorithm 1 is the computation of the LST for jobs arriving after the processor is idle, which is presented in the following section.

[3]The term used in [10, 15] is *the critical speed*. We use a different term to avoid the possible confusion with the speed for the *critical interval* when computing the unconstrained optimal DVS schedule [37].

Algorithm 1 Algorithm to reduce both dynamic and leakage power consumption for real-time systems scheduled according to the FP scheme

Input: $\mathcal{J}$, s_{th}, and T_{min}.
Compute the FP DVS schedule and s_n, $n = 1,2,...,N$;
// s_n is the minimal feasible speed for J_n based on the DVS schedule
Let $s_n = s_{th}$ if $s_n \leq s_{th}$, $n = 1,2,...,N$;
if processor is not idle **then**
  Run job J_i in the ready queue with s_i;
else
  Compute the latest starting time, i.e., $LST(\mathcal{J}_n)$, for future jobs;
  if $LST(\mathcal{J}_n) - t_{cur} > T_{min}$ **then**
    // t_{cur} is the current time
    Shut down the processor and set up the wake up timer to be $LST(\mathcal{J}_n) - t_{cur}$;
  end if
end if

5 Computing the LST

Delaying execution of jobs helps to extend the idle interval length. At the same time, however, it may also cause a job to miss its deadline. The major challenge when extending the length of idle intervals is to determine how long a job set can be delayed without causing any future job to miss its deadline.

5.1 Delaying job executions for FP job sets

Lee *et. al.* [20] first propose delaying the FP hard deadline tasks by their promotion times computed based on the dual priority scheduling scheme. However, this method has been shown to be infeasible [14]. Jejurikar *et. al.* [14] further proposed to delay the execution of a job by the *minimal* promotion time over all lower and equal priority tasks. However, this strategy still cannot guarantee the schedulability of FP-jobs as illustrated in Figure 2.

A task set with two periodic tasks, i.e, τ_1 and τ_2, scheduled with RMS is shown in Figure 2. According to DP scheme, task τ_i can meet its deadline if its promotion time (Y_i) satisfying

$$Y_i \leq D_i - R_i, \tag{6}$$

where D_i is the (relative) deadline of periodic task τ_i and R_i is its worst case response time. It is not difficult to verify that $Y_1 = 6$ and $Y_2 = 7$ in this example. However, assuming processor is idle before $t = 0$, delaying jobs to $t = \min(Y_1, Y_2) = 6$ will cause the first job of τ_2 to miss its deadline as shown in Figure 2. This is because that delaying the high priority jobs may increase the blocking time of lower-priority jobs. For example in Figure 2, without delaying, at most one job from τ_1 can preempt any job of τ_2. However, more than one job of τ_1 can preemp/block the execution of a job of τ_2 if the delayed execution is allowed, and thus cause τ_2 to miss its deadline. Only by strictly adhering to the DP rules, (*i.e.*, the second job of τ_1 will stay in the low priority queue until $t = 16$) can the deadlines be satisfied. This prevents "extra" higher priority jobs (the second job of τ_1 in this case) from preempting/blocking the execution of lower priority jobs (*i.e.*, the first job of τ_2). Therefore, the previous job procrastination strategies based on the promotion time are not feasible when jobs are scheduled according to the FP scheduling scheme.

In [26], Mochocki *et. al.* introduced a method to compute $LST(\mathcal{J})$ when $\mathcal{J}$ is scheduled according to EDF. Their method is based on the following lemma.

Lemma 1 *[26] Let job set ($\mathcal{J}$) be executed with a constant speed s^*, and*

$$lst(J_i) = d_i - \sum_{J_k \in hp(J_i)} \frac{c_k}{s^*}, \tag{7}$$

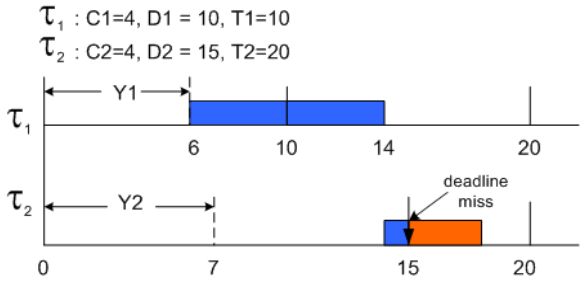


Figure 2. Using the minimal promotion time as the LST may cause FP task sets to miss their deadlines.

where $hp(J_i)$ is the jobs with the same or higher priorities than that of J_i. Then,

$$LST(\mathcal{J}) = \min_i \{lst(J_i)\}. \tag{8}$$

The rationale behind Lemma 1 is that if the accumulated workload from a job J_i and *all* the higher priority jobs can be finished before d_i, the deadline of J_i will be satisfied. It is worthy mentioning that in Lemma 1, equation (7) is pessimistic in evaluating the latest starting time for a *job*, but equation (8) tightly defines the *latest starting time* for the entire job set scheduled with EDF policy. While equation (8) can indeed guarantee the feasibility of an FP job set, the *feasible* starting time for the job set can be far from *the latest*. For example, in Figure 3(a), according to equation (7) and (8), assuming $s_{th} = 0.5$, we have $lst(J_1) = 13$, $lst(J_2) = 14$, $lst(J_3) = 3$, $lst(J_4) = 6$, and therefore, $LST(\mathcal{J}) = 3$. However, even though all the jobs can meet their deadlines, not all the short idle intervals can be effectively merged (Figure 3(a)). For example, if the LST of the task set is delayed to 6 as shown in Figure 3(b), all jobs can meet their deadlines and the short idle intervals are merged to one single idle interval. The reason is that, as opposed to the EDF case, a job with a higher fixed priority can have a deadline much later than that of the current job. Therefore, it would be too pessimistic to assume that all higher priority jobs have to finish before the deadline of the current job.

5.1.1 Identifying the LST of a FP job set

In what follows, we present a more effective method to compute the LST of a FP job set. Our method identifies the LST by judiciously expunging the high priority jobs during the LST computation. We theoretically prove that *all* jobs can meet their deadlines under the FP scheduling policy with the LST computed from our algorithm. Before we explain our strategy in detail, we first introduce some terminology used in this paper.

Definition 2 ***(Scheduling point)***[4] *Time t is called a J_n-**scheduling point** if $t = d_n$ or $t = r_i$, $i < n$ and $r_n < r_i < d_n$.*

Definition 3 ***(Reduced job set)*** *A job set is called a J_n-**reduced job set**, denoted by $\mathcal{R}(J_n)$ if every job J_i in the set satisfies $r_i \geq r_n$.*

We use Figure 3 to illustrate these definitions. Figure 3(c) shows the $\mathcal{R}(J_3)$ and all the J_3-scheduling points (as marked by "x"). Note that in Figure 3(c) if J_3 is to be finished at any one of the J_3-scheduling points (e.g., $t = 12$) all the higher priority jobs arriving before this scheduling point (e.g., J_1) must be completed before this scheduling point. Therefore, if J_n needs to finish at a J_n-scheduling point t, the execution of J_n or any higher priority jobs that may interfere with J_n must begin no later than $st_n(t)$, where

$$st_n(t) = t - \sum_{J_k \in hp(J_n)} \frac{c_k}{s_k}, r_k < t, \tag{9}$$

[4]This is a more general definition of the similar term defined in [22].

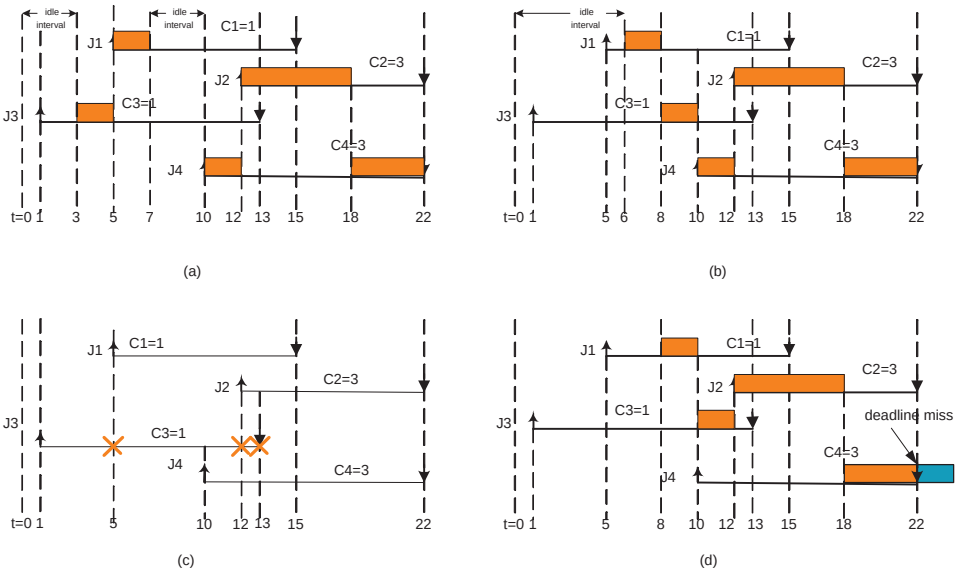


Figure 3. (a) A job set with four jobs scheduled with FP. The processor speed for each job is less than s_{th} (assuming $s_{th} = 0.5$) in the un-constrained DVS voltage schedule. The LST is computed to be 3 according to Lemma 1. (b) Delay the job set until $t = 6$ and every job can meet its deadline. (c) J_3-scheduling points (marked by "x"). (d) Delay execution of the job set untill $t = 8$ and J_4 misses its deadline.

where $hp(J_n)$ is the set of jobs with a priority greater than or equal to that of J_n that arrive before t.

It is not difficult to see that different J_n-scheduling points can lead to different values for $st_n(t)$. If we let $\mathcal{S}(J_n)$ be the set of all J_n-scheduling points, and let

$$lst(J_n) = max\{st_n(t), t \in \mathcal{S}(J_n)\}, \tag{10}$$

then $lst(J_n)$ is the latest time that J_n or any job in $hp(J_n)$ needs to start to ensure that J_n can meet its deadline. We denote the corresponding J_n-scheduling point by $P(lst(J_n))$. From Figure 3(c), we have $lst(J_3) = 8$ (and $P(lst(J_3)) = 12$). It can be readily verified that J_3 can meet its deadline with respect to $lst(J_3) = 8$.

Note that, while $lst(J_n)$ can guarantee the feasibility of job J_n, it cannot guarantee the schedulability for any other job in the J_n-reduced job set. This is shown in Figure 3(d). If J_3 and all the higher priority jobs are delayed to $t = 8$, J_4 will miss its deadline. The reason is that, with $lst(J_3) = 8$, J_3 and the higher priority jobs are not completed until the corresponding scheduling point $t = 12$, which will block the executions of J_4 and cause it to miss deadline. Next, we present an algorithm to determine the latest starting time that can guarantee the deadlines of a job and all lower priority jobs. We call this time the *effective latest starting time* for the job. Based on this time, we present our technique to determine the latest starting time for the entire job set.

As stated before, $lst(J_n)$ can guarantee the feasibility of job J_n but may cause jobs with lower priorities to miss their deadlines. A remedy for this problem is to compute the latest starting times in a similar way for all the lower priority jobs that may potentially be preempted, and pick the smallest one. The above idea is formulated in Algorithm 2.

To formally demonstrate that Algorithm 2 indeed produces the *effective* latest starting time for J_n, we present the following lemma and proof.

Lemma 2 *The effective latest starting time, i.e.,($\tilde{lst}(J_n)$), output from Algorithm 2, is the latest time that J_n and all the higher priority jobs can be delayed to such that J_n and all the lower priority jobs in $\mathcal{R}(J_n)$ will meet their deadlines.*

Algorithm 2 Compute the effective latest starting time $\tilde{lst}(J_n)$ for job J_n such that J_n and all the lower priority jobs in the J_n-reduced job set can meet their deadlines.

1: **Input:** The J_n-reduced job set, i.e., $\mathcal{R}(J_n)$.
2: **Output:** The effective latest starting time for J_n, i.e., $\tilde{lst}(J_n)$
3: $nlst = lst(J_n)$; //Equation (10)
4: $end = P(lst(J_n))$;//the scheduling point corresponding to $lst(J_n)$
5: **for** $J_k \in \mathcal{R}(J_n), k = n+1, n+2, ...$ **do**
6: **if** $r_k < end$ **then**
7: $nlst = \min\{nlst, lst(J_k)\}$;
8: $end = \max\{end, P(lst(J_k))\}$;
9: **end if**
10: **end for**
11: $\tilde{lst}(J_n) = nlst$;

Proof: We first prove schedulability. The schedulability of J_n is guaranteed by equation 10 and in line (3) of Algorithm 2, as well as the fact that *nlst* can only be smaller (line (7)) as the algorithm continues. For any low priority job with a release time earlier than *end*, which may be potentially preempted when delaying J_n and all other jobs with priorities higher than J_n to time *nlst*, its schedulability is guaranteed by line (7) similar to that of J_n. For other lower priority jobs (*i.e.*, with a release time later than *end* during each FOR loop), consider $J_k (k > n)$ and let $r_k > end$. Note that, any higher priority job that is delayed to *nlst* will finish no later than *end*. Therefore, delaying these jobs will not affect the schedulability of J_k. Moreover, the value of *nlst* can only be reduced later on, so J_k can meet its deadline if $\mathcal{J}$ is delayed to *nlst*.

We next prove that $\tilde{lst}(J_n)$ is the *latest*. From equation 10 as well as line (3) and line (7) in Algorithm 2, any further delay will cause J_n or some low priority jobs to miss their deadlines. Therefore, $\tilde{lst}(J_n)$ is the latest time that J_n and other higher priority jobs need to start such that J_n and all the lower priority jobs in $\mathcal{R}(J_n)$ can meet their deadlines. □

Recall that our goal is to identify the latest starting time for a job set such that *every* job can meet its deadline. Using $\tilde{lst}(J_n)$ cannot completely achieve this goal because (1) it is based on an adjusted job set and (2) the schedulability of jobs with a priority higher than that of J_n is not guaranteed in Lemma 2. To find the LST for the entire FP job set, we have the following theorem. The proof of the theorem is given in the appendix of this paper.

Theorem 1 *Given job set $\mathcal{J}$, the latest starting time for $\mathcal{J}$ can be computed as*

$$LST(\mathcal{J}) = \min_n \{\tilde{lst}(J_n))\}. \tag{11}$$

where $\tilde{lst}(J_n)$ is computed according to Algorithm 2.

For the example in Figure 3, according to Theorem 1, we have $\tilde{lst}(J_1) = 8$, $\tilde{lst}(J_2) = 16$, $\tilde{lst}(J_3) = 6$, $\tilde{lst}(J_4) = 10$, and therefore $LST(\mathcal{J}) = 6$, which is exactly the case shown in Figure 3(b). As shown in Figure 3(b), all the idle intervals are successfully merged into one single interval.

While equation (11) requires computing the *effective* LST for all jobs, it is not necessary in practice. Note that, to ensure the schedulability, a task set cannot be delayed past the earliest deadline of a job, which bounds the maximal value of LST. Therefore, we only need to compare the effective LSTs for jobs released before this bound and use the minimal one as the LST for the entire job set. To further reduce the on-line cost, we can compute the LST for each possible reduced job set off-line. Note that, whenever the processor is idle, the rest of the job set can always be viewed as a reduced job set. We therefore can construct all the possible reduced job sets off-line and then compute the corresponding LSTs. For a periodic task set, this means the computation of total $\sum_n \frac{P}{P_n}$ reduced job sets, where P is the least common multiple of the periods and P_n is the period for task n. During on-line phase, the LST can be readily determined by the LST associated with the next job that arrives. This on-line technique has a very low complexity, i.e., a constant time complexity for a single table lookup operation.

6 Experimental results

In this section, we evaluate the proposed technique using simulations. We consider the following scenarios in our experiments.

- **Base** The task sets are scheduled on a processor without DVS capability, *i.e.*, all jobs are always executed using the highest speed. A processor is shut down when there is enough idle time, and no task instance is delayed. This is the most primitive scheduling approach, and its results are used as the reference to compare other approaches.
- **DVS** The task sets are scheduled according to the DVS voltage schedules without considering leakage (*i.e.* the threshold speed), and no task instance is delayed.
- **DVS with No Delay (DVSND)** Task sets are scheduled with DVS voltage schedules and leakage is considered (i.e., the threshold speed is enforced), but no job execution is delayed.
- **DVS with Shut down and Delay (DVSSD-FP)** Task sets are scheduled with DVS voltage schedules, the threshold speed is enforced and execution delay is computed using (Algorithm 1). The LST computation is based on Theorem 1.

In addition, we have also implemented and compared our algorithm with the dual-priority approach introduced in [14], even though it is not strictly an FP approach. We call this approach DVS Dual Priority(**DVS-DP**).

We conducted two groups of experiments to evaluate the performance of our approaches. The first group experiments were based on synthesized task sets and a more theoretical processor. In our second group of experiments we intended to make our test conditions as close as possible to that in the practical scenarios. The test cases were drawn from practical applications, and a more practical processor model that supports only discrete voltage levels was used. The experiments and results are discussed in the following.

6.1 Experiments with synthesized task sets

In this group of experiments, the periodic real-time systems were randomly generated and used as the test cases. These systems consist of five periodic tasks, with task periods randomly chosen in the range of $[10,50]ms$, and deadlines assumed to be equal to their periods. We assumed that the actual execution time of a job is normally distributed between its best case execution time (BCET) and worst case execution time (WCET), with $BCET/WCET = 0.4$. We examine the performance of the above techniques for systems with different utilizations. Based on the utilization bound for periodic task set with five periodic tasks, i.e., $U = 5(2^{1/5} - 1) = 0.74$, we divide utilization ranging from 0.0 to 0.7 into intervals of length 0.1. Within each interval, we randomly generated no less than 10 periodic task sets.

For the processor model used for this group experiments, similar to [15], we assumed the processor voltage is continuously variable, and adopted the same threshold speed and sleep state power as that used in [15], *i.e.*, $s_{th} = 0.41$ and $P_{sleep} = 50\mu W$. We conservatively assumed that the power consumption when processor is idle comes only from the leakage power consumption, which is computed according to the model in [25]. We also made a conservative assumption that it takes only $1ms$ for the processor to be put into the low leakage mode even though the actual time can be much longer [9]. The shut-down energy overhead consists of two parts: the leakage energy consumption during the shut-down process, which was computed based on the power model in [25], and the fixed energy overhead to flush/restore the cache contents, which we used $483\mu J$ according to [15].

For approaches **DVS**, **DVSND**, and **DVSSD-FP**, we used VSLP [31] to find the unconstrained job-level DVS voltage schedule. This heuristic was chosen instead of the optimal algorithm [30] because the FP DVS problem is NP-Hard while the computation complexity of VSLP is much lower ($O(N^3)$). The optimal algorithm is not practical for systems with a large number of jobs. On the other hand, VSLP [31] cannot be applied for **DVS-DP** approach. **DVS-DP** computes the delay for the task set based on the worst case timing analysis, which is not possible if different

jobs from the same task have different worst case execution times. We hence used the task-level DVS scheduling algorithm (i.e. [35]) to find the DVS schedule in this approach. To dynamically reclaim the system resource when real-time jobs finish earlier than their worst case execution times, we adopted the technique in [34] for its simplicity, i.e., we prolonged the execution of a real-time job to its deadline or next arrivals of new jobs (whichever is the earliest) when it is the only job in the ready queue. To be overall energy efficient, the processor speed will never be set below the threshold speed when dynamically reclaiming run-time slack.

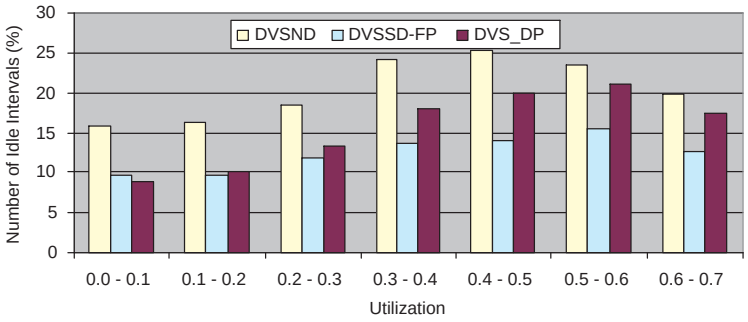


Figure 4. The average idle intervals by three different approaches for synthesized task sets.

We first study the number of idle intervals resulting from each scheduling strategy. A large number of idle intervals is undesirable in terms of energy reduction, especially for leakage energy reduction. Schedules with large number of idle intervals either incur higher transition overhead due to more frequent transitions or simply cannot go into the low leakage mode due to shorter idle interval lengths. Figure 4 compares the normalized (according to the results from the **Base** approach) number of idle intervals resulting from several approaches within the LCM of the task periods.

Figure 4 clearly shows that our proposed technique (i.e. **DVSSD-FP**) can merge the idle intervals very effectively. From Figure 4, **DVSSD-FP** can significantly cut the idle interval numbers by **DVSND**, i.e., ranging from 34.3% to 44.3%, with an average of 39.1%. The results for **DVSSD-FP** and **DVSDP** are interesting. Note that, when the utilization is low (i.e., less than 0.3), the numbers of idle intervals by **DVSSD-FP** and **DVSDP** are quite close. But when the utilization is relative high, **DVSDP** can lead to much larger number of idle intervals than **DVSSD-FP**. For example, when utilization is around 0.6-0.7, the number of idle intervals by **DVSDP** is 30% higher than that by **DVSSD-FP**. This is because of two reasons. First, **DVSDP** can exploit DVS capability only at the task level, *i.e.* different jobs of the same task always have the same processor speed, which may require jobs to run at much higher speeds than they actually need and increase the idle intervals. Second, **DVSDP** always computes the latest starting time based on the worst case response time. When utilization is relatively high, This strategy can severely underestimate the maximal delay that a job is allowed, and therefore cannot merge the idle intervals effectively.

We next study the overall energy consumption for the same task sets by different strategies. Figure 5 shows the normalized average total energy consumptions by four approaches, i.e., **DVS**, **DVSND**, **DVSSD-FP**, and **DVSDP**.

From Figure 5, one can readily conclude that using DVS without considering leakage current cannot effectively reduce the overall energy consumed. This is particularly true when the utilization of the task set is low. As shown in Figure 5, when the utilization is less than 0.1, the average overall energy using the “pure” DVS voltage schedule can be more than 25% higher than the leakage conscious approaches such as **DVSSD-FP** and **DVSDP**. This is due to the factor that, when the utilization is low, the processor is required to run at a very low speed according to the classical DVS approach. While reducing supplied voltage can reduce the dynamic and leakage *power* consumption, the extended execution time with a lower processor speed will rapidly increase the leakage energy consumption, which may actually increase the total *energy* consumption. We would expect a leakage conscious approach to be

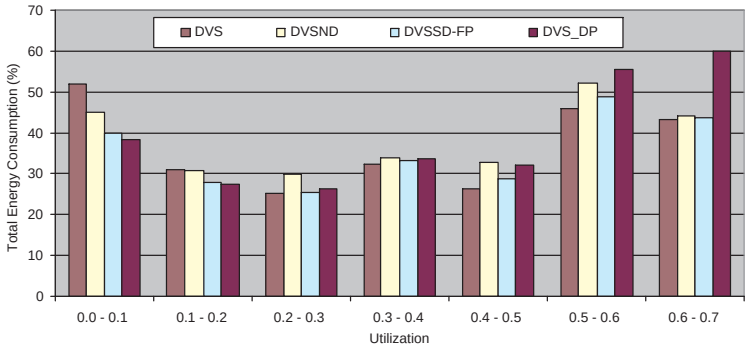


Figure 5. The average energy consumption by different approaches for synthesized task sets.

more effective when leakage power consumption becomes the dominant component in overall power consumption.

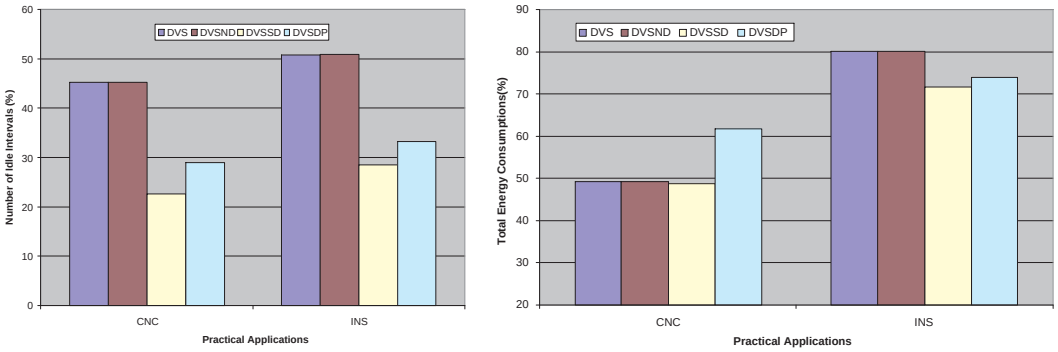
On the other hand, when utilization is relative high, we do observe that **DVS** can be slightly better than the others. There are several reasons for this result. First, we adopt the concept of the threshold speed as defined in equation (5). Note that such a speed can only optimize the energy consumption in executing a job (i.e., during the active mode), but not necessarily minimize the energy consumption during the entire life cycle (including both the active and idle period) of the system. When the system utilization is very high, reducing processor speed as low as possible is still beneficial since both dynamic and leakage power consumption are significantly reduced. The power reduction outweighs the extended execution times of tasks and thus leads to overall energy reduction. Second, we delay the job execution as late as possible, which is not necessarily the optimal approach to merge the idle intervals. To find the theoretically optimal solution for this problem is an interesting one and will be our future research.

Figure 5 also shows that our approach outperforms those that adopt a similar heuristic. By effectively merging the idle times, our approach, i.e., **DVSSD-FP** consumes over 14% less energy than that by **DVSND**. Compared with **DVSDP** which also delays job executions, we can see that the energy savings of **DVSSD-FP** *vs.* **DVSDP** varies depending on the utilization. **DVSDP** works under the assumption that each task is assigned a unique scaling factor, which is determined based on the worst case scenario. As a result, real-time jobs may be run at speeds much higher than necessary, which is not energy efficient and also cause more idle intervals. In our approach, different jobs may be assigned different processor speeds as necessary, and the idle intervals can be effectively merged. When the utilization is low, for example, within the interval $[0.0, 0.4]$, both approaches have the similar energy consumptions since most of the jobs are forced to execute with the threshold speed and most of the idle intervals can be long enough for shutting down the processor. When the task utilization is higher, **DVSSD-FP** can save much more energy than **DVSDP**, not only because it can take the advantage of the job-level DVS schedule but also because it can merge the idle intervals more effectively. As shown in Figure 5, when utilization is around 0.6, **DVSSD-FP** consumes 27% less energy than **DVSDP**.

6.2 Experiments with practical applications

In our second group of experiments, the test cases are drawn from two real world applications, i.e., CNC (Computerized Numerical Control) [28] and INS (Inertial Navigation System) [1]. The processor and power models used in this group of experiments are the same as that in the previous ones expect that the processor supports only five discrete voltage levels, with normalized speeds as 0.2, 0.4, 0.6, 0.8 and 1.0. The critical speed was chosen to be $s_{th} = 0.4$.

When the processor supports only a number of discrete voltage levels, more idle intervals are created since the processor speeds for real-time jobs often have to be rounded up to a higher level. Our experimental results exhibit



(a) Idle Interval (b) Total Energy

Figure 6. The average number of idle intervals and energy consumptions for INS and CNC.

the excellent performance of our approach in merging the idle intervals under this scenario. Figure 6(a) compares the normalized number of idle intervals for CNC and INS by different approaches. As can be seen from Figure 6(a), **DVSSD-FP** can cut approximately 50% of the idle intervals produced by **DVSND**, and around 29% of those produced by **DVSDP** for CNC application. For the INS application, **DVSSD-FP** can cut around 44% of the idle intervals produced by **DVS** and **DVSND**, and around 15% of those produced by **DVSDP**.

Our experimental results demonstrate the excellent performance of our approach for practical applications not only in merging the idle intervals, but also in total energy savings as well. It is interesting that Figure 6(b) shows that **DVSSD-FP** achieves different energy saving performance for CNC and INS. A closer look at our experimental results reveals that, for CNC application, the numbers of idle intervals are small (i.e. 28 for **DVSND** and 14 for **DVSSD-FP**), and the energy consumption during the active mode dominates the energy of the idle mode. Further, all the processor speeds from the DVS schedule are higher than the threshold speed. As a result, we see almost no difference between **DVS**, **DVSND**, and **DVSSD-FP**. For CNC, however, **DVSDP** consumes much more energy than the other approaches (over 21%) since it cannot effectively reduce the processor speed. For the INS application, the number of idle intervals is much higher (i.e., 747 for **DVSND** and 418 for **DVSSD-FP**), which makes the idle interval merging more profitable. As shown in Figure 6(b), **DVSSD-FP** can reduce the energy consumed by as much as 10.4% when compared with **DVS** and **DVSND**.

7 Summary

Reducing the overall power dissipation is critical in the design of future real-time embedded systems. As the IC technology continues to scale down, leakage power consumption is becoming a more and more significant part of the overall power consumption. In this paper, we investigate the problem of applying real-time scheduling techniques to reduce the overall energy consumption of real-time systems scheduled by the FP schemes.

As demonstrated in our experiments, applying a DVS based voltage schedule alone cannot effectively reduce the overall energy consumption for the system, and can even increase it significantly. A leakage-power-conscious voltage schedule may require the processor to adopt a speed higher-than-necessary to avoid the rapidly increasing leakage energy consumption at low voltage levels. This could result in a large number of small idle intervals during job execution, which is not energy-efficient considering the overhead associated with the process of shutting down and waking up the processor.

To reduce the processor shutdown overhead and improve the overall energy performance, previously proposed techniques are based on the task level DVS which require that real-time jobs run at unnecessarily high speeds and may generate even more idle intervals. In this paper, we proposed an efficient and effective approach to merge idle intervals and improve the overall energy performance of FP systems. Under the job-based analysis paradigm, our

techniques can be applied to both periodic and aperiodic real-time tasks, and are more flexible and efficient in dealing with the run-time variations. Extensive and comprehensive experiments are conducted and clearly demonstrate that our approaches can significantly outperform previous research in reducing the number of idle intervals and the overall power consumption.

References

[1] A.Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21:920–934, May 1995.

[2] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. Al-Hashimi. Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. *DATE'04*, 2004.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Alvarez. Dynamic and aggressive scheduling techniques for power aware real-time systems. *IEEE Real-Time System Symposium*, 2001.

[4] B. H. Calhoun, F. A. Honore, and A. Chandrakasan. Design methodology for fine-grained leakage control in mtcmos. *ISLPED*, pages 104–109, 2003.

[5] L. Clark, R. Patel, and T. Beatty. Managing standby and active mode leakage power in deep sub-micron design. *ISLPED*, pages 274–279, 2004.

[6] R. Davis and A. Burns. Optimal priority assignment for aperiodic tasks with firm deadlines in fixed-priority preemptive systems. *Information Processing Letters*, 53(5):249–254, 1995.

[7] D.Duarte, N.Vijaykrishnan, M.J.Irwin, H.-S. Kim, and G.McFarland. Impact of scaling on the effectiveness of dynamic power reduction schemes. *ICCD*, 2002.

[8] S. Duarte, Y. Tsai, N. Vijaykrishnan, and M. Irwin. Evaluating run-time techniques for leakage power reduction. *VLSID'02*, 2002.

[9] Intel. *PXA250 and PXA210 Applications Processors Design Guide*. Intel, 2002.

[10] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *SODA*, 2003.

[11] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. *ISLPED*, pages 197–202, August 1998.

[12] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. *ISLPED*, pages 197–202, August 1998.

[13] ITRS. *International Technology Roadmap for Semiconductors*. International SEMATECH, Austin, TX., http://public.itrs.net/.

[14] R. Jejurikar and R. Gupta. Procrastination scheduling in fixed priority real-time systems. *LCTES*, 2004.

[15] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. *DAC*, pages 275 – 280, 2004.

[16] M. Johnson, D. Somasekhar, L. Choiu, and K. Roy. Leakage control with efficient use of transistor stacks in single threshold cmos. *IEEE Trans. on VLSI*, 10(1):1–5, February 2002.

[17] W. Kim, J. Kim, and S. L. Min. Preemption-aware dynamic voltage scaling in hard real-time systems. *ISLPED*, pages 393–398, 2004.

[18] W. Kim, J. Kim, and S.L.Min. Dynamic voltage scaling algorithm for fixed-priority real-time systems using work-demand analysis. *ISLPED*, 2003.

[19] W. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *DAC*, pages 125–130, 2003.

[20] Y. Lee, K. Reddy, and C. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. *ECRTS*, 2003.

[21] J. Lehoczky and S. Ramos-Thue. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. *Proceedings of the 1992 IEEE Real-time System Symposium*, pages 110–123, 1992.

[22] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. *Proceedings of the 1989 IEEE Real-time System Symposium*, pages 166–171, 1989.

[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 17(2):46–61, 1973.

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

[25] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microporcessor under dynamic workloads. *ICCAD*, 2002.

[26] B. Mochocki, X. Hu, and G. Quan. A realistic variable voltage scheduling model for real-time applications. *IEEE/ACM 2002 International Conference on Computer Aided Design*, 2002.

[27] C. Neau and K. Roy. Optimal body bias selection for leakage improvement and process compensation over different technology generations. *ISLPED*, pages 116–121, 2003.
[28] N.Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin. Visual assessment of a real-time system design: a case study on a cnc controller. *IEEE Real-Time Systems Sumposium*, Dec 1996.
[29] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, 2001.
[30] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processors. *2002 European Design and Test Conference*, 2002.
[31] G. Quan and X. S. Hu. Energy efficient fixed-priority scheduling for real-time systems on voltage variable processors. *DAC*, pages 828–833, 2001.
[32] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Preceedings of IEEE*, 91(2):305–327, Feb 2003.
[33] K. Seth, A. Anantaraman, F. Müeller, and E. Rotenberg. FAST: Frequency-aware static timing analysis. *RTSS*, pages 40–51, 2003.
[34] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. *DAC*, pages 134–139, 1999.
[35] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. *International Conference on Computer-Aided Design*, pages 365–368, 2000.
[36] L. Yan, J. Luo, and N. K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. *ICCAD*, 2003.
[37] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *IEEE Annual Foundations of Comp. Sci.*, pages 374–382, 1995.
[38] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and practical limits of dynamic voltage scaling. *DAC*, pages 868–873, 2004.

8 Appendix

8.1 Proof for Theorem 1

The proof of Theorem 1 needs the following lemma regarding to the *effective latest starting time* for a job, i.e., $\tilde{lst}(J_n)$ (see section5.1.1).

Lemma 3 *For job set $\mathcal{J}$, let J_i, $J_k \in \mathcal{J}$, $i < k$. Then $\tilde{lst}(J_i) \leq \tilde{lst}(J_k)$ if $r_i < r_k$.*

Proof: The proof for the case $d_i \leq r_k$ is trivial since $\tilde{lst}(J_i)$ cannot exceed d_i. We use contradiction to prove that when $d_i > r_k$ and $r_i < r_k$, $\tilde{lst}(J_i) > \tilde{lst}(J_k)$ is not possible.

Let $\mathcal{J}_i$ and $\mathcal{J}_k$ represent the corresponding J_i- and J_k-reduced job sets, respectively, and $LP(J_p, \mathcal{J}_p)$ represent the jobs in $\mathcal{J}_p$ with priorities the same or lower than that of J_p. Then

$$\mathcal{J}_i \supset \mathcal{J}_k, \text{ and } LP(J_i, \mathcal{J}_i) \supset LP(J_k, \mathcal{J}_k).$$

According to Lemma 2, delaying the execution of $\mathcal{J}_i$ to $\tilde{lst}(J_i)$ can ensure that all jobs in $LP(J_i, \mathcal{J}_i)$ meet their deadlines. If $\tilde{lst}(J_i) > \tilde{lst}(J_k)$, this contradicts to the fact that $\tilde{lst}(J_k)$ is the latest time that $\mathcal{J}_k$ can be delayed to such that the jobs in $LP(J_k, \mathcal{J}_k)$ can meet their deadlines. □

To prove Theorem 1, let $LST(\mathcal{J}) = \tilde{lst}(J_i) = \min_n\{\tilde{lst}(J_n)\}$. We want to prove that any other $J_k \in \mathcal{J}$ can meet its deadline if $\mathcal{J}$ is delayed to $\tilde{lst}(J_i)$. We consider two different cases separately.

- Case 1: $k < i$.

  From Lemma 3, we have for any $k < i$, $r_k \geq r_i$. Let job r_q be the earliest arrival time for any job J_q such that $q < k$. If we have $r_q \geq r_k$, J_k can meet its deadline since $\tilde{lst}(J_k) \geq \tilde{lst}(J_i)$. On the other hand, if $r_q < r_k$, the schedulability of J_k is guaranteed according to Lemma 2 due to the fact that $\tilde{lst}(J_q) \geq \tilde{lst}(J_i)$ and J_k is a lower priority job of J_q.

- Case 2: $k > i$

  If all the jobs arrive later than J_i, Lemma 2 can guarantee J_k's deadline. Assume there is at least one job arriving earlier than J_i, and let J_k be the one with the earliest arrival time. Since $\tilde{lst}(J_i) \leq \tilde{lst}(J_k)$, J_k and all the lower priority jobs can meet their deadlines. Therefore, we only need to consider the job J_q such that $i < q < k$. Note that, for any such job J_q, removing J_k and all the lower priority jobs from $\mathcal{J}$ neither changes its feasibility nor *increase* $\tilde{lst}(J_q)$. If $r_q < r_i$ and r_q is the next earliest arrival time of the jobs, we can prove that J_q and all the lower priority jobs can meet their deadlines similarly. By repeating this process, we thus prove that all the lower priority jobs can meet their deadlines if $\mathcal{J}$ is delayed to $\tilde{lst}(J_i)$.