Fixed Priority Scheduling for Reducing Overall Energy on Variable Voltage Processors

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

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

Abstract

While Dynamic Voltage Scaling (DVS) is an efficient technique in reducing the dynamic energy consumption of a CMOS processor, methods that employ DVS without considering leakage current are quickly becoming less efficient when considering the processor's overall energy consumption. A leakage conscious DVS voltage schedule may require the processor to run at a higher-than-necessary speed to execute a given set of real-time tasks, which can result in a large number of idle intervals. To effectively reduce the energy consumption during these idle intervals, and therefore the overall energy consumption, the DVS schedule must judiciously allow the processor to enter and leave the power down state during these idle intervals, while considering the time and energy cost of doing so. In this paper, we present a scheduling technique that can effectively reduce the overall energy consumption for hard real-time systems scheduled according to a fixed priority (FP) scheme. Experimental results demonstrate that a processor using our strategy consumes as less as 15% of the idle energy of a processor employing the conventional strategy.

1. Introduction

Power consumption has become one of the primary design issues of next-generation portable, scalable and sophisticated embedded systems. For CMOS circuits, power consumption includes dynamic power and leakage power. Dynamic power is due to the switching activities of the transistors, while leakage power is consumed even when no logic operations are performed. A major component of leakage current is the subthreshold current that flows through the transistors when they should be logically "off". Current power saving techniques mainly focus on reducing dynamic power because it has been the dominant component in the overall power consumption for most embedded systems today. However, as VLSI technology continues its evolution 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 [1]. More advanced techniques required for the development of future generations of low-power embedded systems.

Due to 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 [2, 3, 4, 5], have been proposed to control leakage power. 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 is one of the most effective ways of reducing power consumption, through the exploitation of advanced power-management features available in many of today's processors.

Dynamic Voltage Scaling (DVS) can effectively reduce dynamic power consumption in real-time systems. 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 of the leakage power consumption, a five-fold increase per technology generation according to [1]. In fact, as shown in our experiments, using DVS alone with no consideration of leakage power consumption may actually increase the total energy consumption! This situation occurs because DVS and leakage reduction techniques are at odds. The most effective way to reduce leakage power is to put the processor into a low-power sleep state during idle intervals. On the other hand, DVS reduces the processor's execution speed to minimize dynamic power. By reducing the execution speed, the processor utilization is increased, thus reducing and fragmenting available idle times. It is this tradeoff that makes leakage reduction a considerable challenge.

In this paper, we study scheduling techniques that can minimize the overall power consumption for a real-time system scheduled using a fixed-priority (FP) scheme. Many DVS based real-time scheduling techniques, *e.g.* [8, 9, 10, 11], have been proposed to conserve energy in a real-time system. Some of these approaches, such as [11, 12, 13, 14, 15, 16, 17], are targeted at FP systems.

Recently, some work has been reported that deals with the leakage power consumption in real-time scheduling. Lee *et. al.* [18] proposed a leakage reduction scheduling technique called **LC-EDF**. They assumed a non-DVS processor, which makes shutting down the processor during idle intervals the most effective way to reduce the overall energy consumption. Considering the timing and energy overhead associated with shutting down the processor, LC-EDF carefully delays the execution of arriving task instances in order to expand the length of idle intervals. Due to the limitation of their processor model, the overall energy consumed cannot be minimized. Irani *et. al.* [19] theoretically proved that the optimal voltage schedule, which also considers the leakage power, can be constructed from the corresponding DVS voltage schedule without the leakage power consideration. In this case, higher-than-necessary processor speeds may be required in the optimal schedule to balance the dynamic and leakage power consumption. To better save idle energy during idle periods, Jejurikar *et. al.* [20] proposed a better approach, called **CS-DVSP** to extend idle intervals. They showed that the minimal length of the idle intervals according to **CS-DVSP** is no less that that by **LC-EDF**. However, all these approaches are targeted at the real-time systems scheduled according to the earliest deadline first (EDF) scheme [21].

We are more interested in real-time systems scheduled according to a FP scheme. Because of their high predictability, low overhead, and ease of implementation, FP schemes are among the most popular in real-time embedded applications [22]. Lee *et. al.* [18] proposed a leakage reduction scheduling technique for FP systems, called **LC-DP**, by extending the Dual-Priority (DP) scheduling model presented in [23]. In **LC-DP**, idle time is treated as a "soft task" in the DP model. A task instance is delayed by first being released in the lower priority queue if the processor is idle. It is promoted to the higher priority queue for execution at an optimal promotion time to avoid any deadline misses. **LC-DP** also immediately promotes a task instance to the higher priority queue when the processor is not idle in order to reduce the number of idle intervals. However, Jejurikar *et. al.* [30] pointed out that this may potentially lead to some task instances missing their deadlines. They further proved that using the optimal promotion time as the allowable delay for each task instance can guarantee schedulability for both dual priority and fixed priority policies. However, since the computation of the optimal promotion time for each task instance is performed based on the exact response time analysis, which is NP-hard in nature [23], this approach cannot be readily applied on-line or for large task sets. If the promotion time is computed based on the worst case response time (by assuming a task instance arrives simultaneously with all the higher priority ones), the possible delay for each task instance can be estimated rather pessimistically which severely limits the energy performance of this approach.

In this paper, we present a scheduling technique that combines both the DVS and a shut-down strategy to effectively reduce the overall energy consumption of FP hard real-time systems. As shown by Irani *et. al.* [19], such a technique may require that the processor run at a higher-than-necessary speed and hence produce a large number of processor idle intervals. The major source of energy consumption in these intervals is the result of leakage current, which will soon become a major portion of the overall energy consumption. In this regard, we present an efficient technique that delays the execution of tasks in order to merge scattered idle intervals, thus greatly reduces leakage power as well as the impact of processor shutdown overhead. The proposed technique has a very low on-line computation cost, and experimental results show that our method can significantly reduce the energy consumption when compared with the traditional non-delay strategy.

The rest of the paper is organized as follows. Section 2 introduces preliminaries related to our problem. Section 3 discusses our delay analysis technique. Section 3.3 presents our on-line leakage conscious DVS algorithm. Section 4 demonstrates the effectiveness of our approach based on simulations. Section 5 concludes the paper.

2. Preliminaries

This section, describes the real-time system and power model used in this paper.

2.1. System model

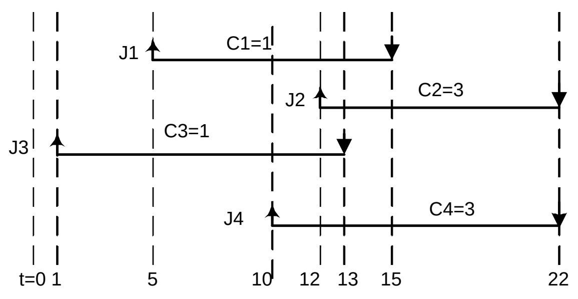
We conduct our study for a set of N independent jobs, denoted by $\mathcal{J} = \{J_1, J_2, \cdots J_N\}$. Each individual job is denoted by $J_i = (r_i, c_i, d_i)$, where r_i, c_i, and d_i are arrival time, worst case execution cycle, and absolute deadline for the job, respectively. Additionally, each job is statically assigned a priority. We assume that J_i has a higher priority than J_j, if $i < j$. Often a real-time system is described by a set of periodic tasks, where each task instance represents one job. In these cases, it is sufficient to schedule the set of jobs produced up until the Least Common Multiple (LCM) of the periods of each task.

2.2. Power model

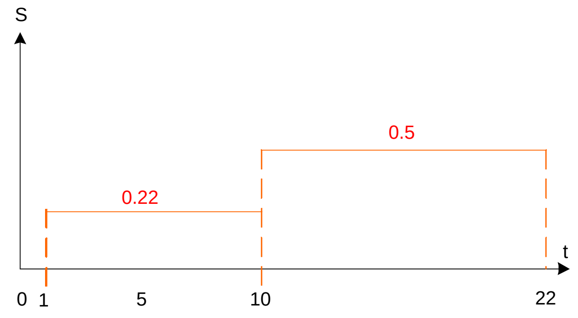
In a CMOS circuit, the power consumption includes both dynamic and static components during its active operation. The dynamic power consumption (P_{dyn}) mainly consists of the switching power for charging and discharging the load capacitance, which can be represented [24] as

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

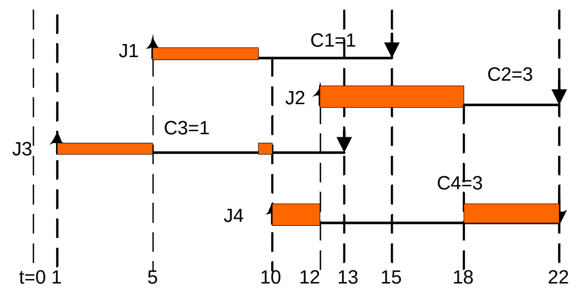
where α is the switching activity, C_L is the load capacitance, V is the supply voltage, and f is the system clock frequency.



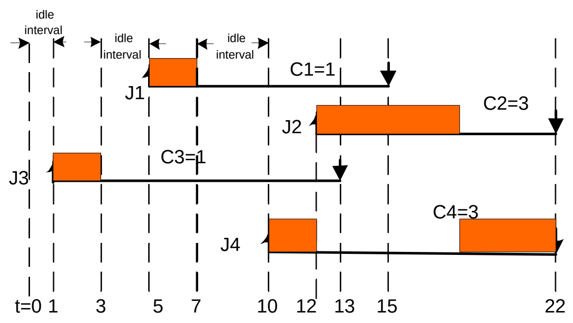
(a)



(b)



(c)



(d)

The static power (P_{leak}) can be expressed [25] 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 the devices and becomes particularly significant with the reduction of the threshold voltage [26]. Therefore, the leakage power consumption is becoming a major part of the the active power consumption (P_{act}), i.e.,

$$P_{act} = P_{dyn} + P_{leak}, \tag{3}$$

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 the processor is idle, the major portion of the power consumption comes from the leakage, which is increasing rapidly with newer CMOS technologies. Shutting down the processor, i.e., putting the processor into a "sleep mode" can greatly reduce the energy consumption during these idle periods. For example, it has been reported in [27] that the power dissipation when the processor is idle can be on the order of 10^3 times that when it is sleeping. While the processor consumes less power in sleep mode, extra energy and time are needed for it to enter and later leave this state, because one must save/restore the context as well as initiate architectural components such as the cache, translation look aside buffers, and branch target buffers. This energy overhead may outweigh the energy saved if the idle interval is not long enough. Assume that the energy overhead of shutdown/wakeup is E_o, the timing overhead is t_o, and the power consumption of a processor in its idle and sleeping state are P_{idle} and P_{sleep}, respectively. Then, the energy can be saved only when the length of the idle interval is 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*.

2.3. A motivational example

Our goal is to minimize the *overall* energy consumption while guaranteeing task deadlines. As indicated in equation (1), the dynamic energy consumption is quadratically related to the supplied voltage. Therefore, traditional DVS scheduling techniques [14, 15, 16] try to reduce the the supply voltage to as low a level as possible. As an illustrative example, Figure 1(a) shows a job set with four jobs. Figure 1(b) is the voltage schedule according to the DVS scheduling technique presented in [14], and Figure 1(c) shows the actual executions of the jobs based on the voltage schedule from Figure 1(b).

As shown in Figure 1(b) and Figure 1(c), previous DVS techniques [14, 15, 16] can effectively reduce the processor speed and guarantee the deadlines of the real-time jobs. However, such a voltage schedule is not always feasible and/or energy efficient overall. First, practical processors have a minimal voltage supply limitation. Second, they only provide a discrete set of voltages, including the minimum level. 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. On the other hand, 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

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

Hence, to minimize the $E_{act}(s)$ in equation 4, we have

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

Equation (5) computes the most energy efficient speed to execute one job. We call this speed as the *threshold speed*, and denoted 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 power consumption.

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, *i.e.*, jobs with higher priorities can always block jobs with lower priorities until they are finished. Given a voltage schedule, a job that is required to run at a speed higher than s_{th} must be executed with that higher 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), it can result in a large number of scattered idle intervals. While 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 a 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 performance of the system. In what follows, we introduce our approach to save the idle energy when scheduling a FP task set by extending the length of idle intervals.

3. Leakage conscious scheduling algorithm

In this section, we present our scheduling technique to reduce the idle energy for a set of real-time jobs. We first analyze how a job set can be delayed without missing deadlines. Then we construct an algorithm that can be applied on-line to reduce energy consumption during idle intervals.

3.1. Basic concepts

The power down strategy is in favor of 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 overall power consumption. A better approach, as suggested in [18, 19, 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.

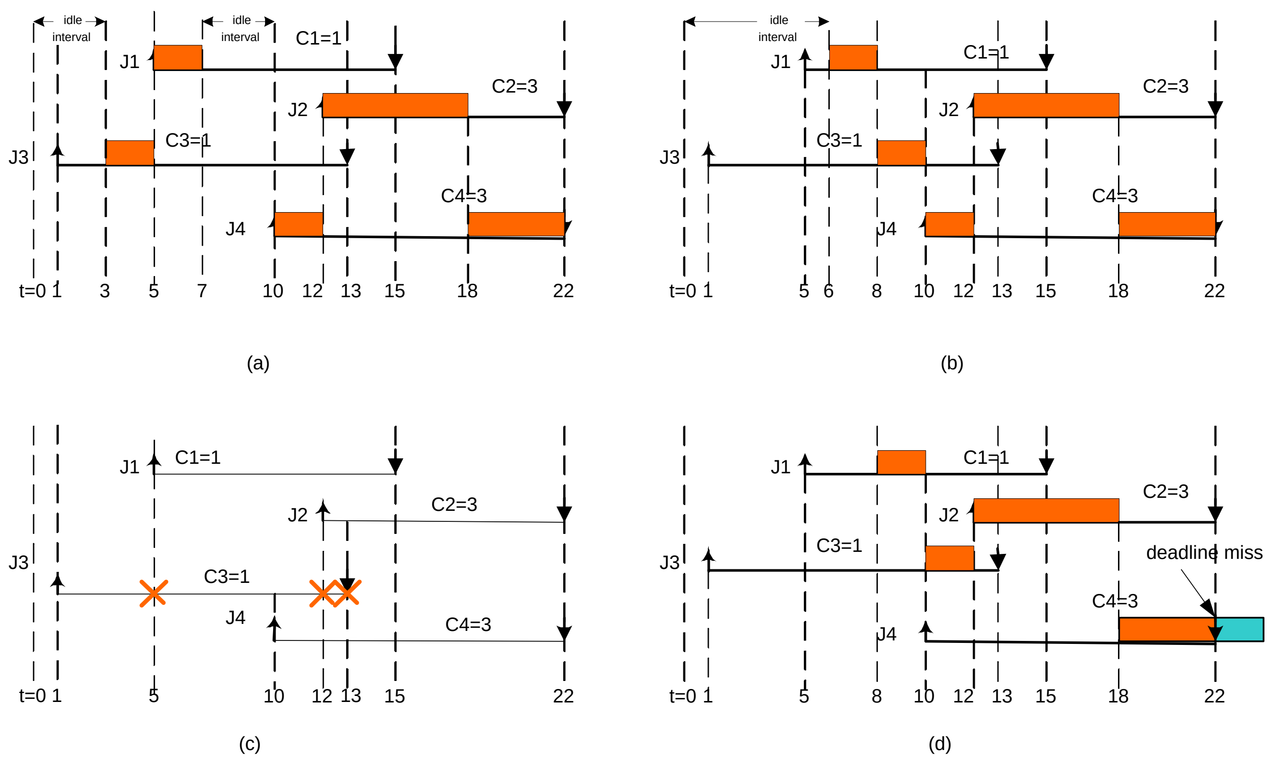
Delaying job executions helps to merge scattered idle intervals into longer ones. More energy can be saved because energy transition overhead for entering and leaving the low-power sleep state is reduced. Moreover, intervals that were previously shorter than T_{min} can now be shut down. As mentioned before, the power dissipation when the processor is idle can be in the order of 10^3 times of that when the processor is shut down.

The main difficulty 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. Chetto [28] introduced a static scheduling technique called EDL (earliest deadline as late as possible) to determine the longest time that a job can be delayed. However, it requires the jobs be scheduled according to the earliest deadline scheduling algorithm. For job set scheduled by a FP (fixed priority) scheme, we derived a new approach to determine the latest time point to which the job set can be delayed. To facilitate a clear explanation, we first introduce the following definitions.

Definition 1 *Let job set ($\mathcal{J}$) be executed with a constant speed s^*.*

- *The latest starting time of a job , e.g., $J_i \in \mathcal{J}$, (denoted as $lst(J_i)$) is the latest time such that, if the execution of J_i or jobs with a priority higher than J_i start no later than $lst(J_i)$, J_i will meet its deadline.*
- *The latest starting time of a job set , e.g. $\mathcal{J}$, (denoted as $LST(\mathcal{J})$) is the latest time such that, if the execution of any jobs in $\mathcal{J}$ starts no later than $LST(\mathcal{J})$, all jobs will meet their deadlines.*

In [29], 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 *[29] Let job set ($\mathcal{J}$) be executed with a constant speed s^*. Then,*

$$lst(J_i) = d_i - \sum_{J_k \in hp(J_i)} \frac{c_k}{s^*}, \tag{6}$$

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

$$LST(\mathcal{J}) = \min_i \{lst(J_i)\}. \tag{7}$$

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. In addition, the minimal latest starting time of all the jobs can certainly guarantee all the deadlines. It is not difficult to see that using equation (7) to compute the starting time for a FP job set can still guarantee the feasibility of this job set. Unfortunately, using Lemma 1 may not ensure that the *feasible* starting time for the FP job set is always the latest. For example, in Figure 2(a), according to equation (6) and (7), assuming $s^* = 0.5$, 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, as shown in Figure 2(b), if the job set is delayed to $t = 6$, all the jobs can meet their deadlines. The consequence is that all of the short idle intervals cannot be effectively merged as shown in Figure 2(a).

Note that accumulating the workload from all the higher priority jobs in equation (6) is equivalent to *assuming* that all the higher priority jobs have to finish before the deadline of current job. This is true for job sets scheduled according to EDF, but is not necessarily true for FP job sets since higher priority jobs in a FP job set can arrive much later than the deadline of the current job. In what follows, we present a more effective technique to accurately identify the latest starting point for FP job sets.

3.2. Analyzing the latest starting time for FP job sets

Recall that the jobs with required speeds higher than s_{th} should run at their required speeds in order to guarantee deadlines. These jobs cannot be delayed at all and must be executed within the intervals in the DVS voltage schedule. For ease of computation, we "shrink" the intervals during which jobs with a required speed higher than s_{th} are executed. This includes removing all jobs in these intervals, and also adjusting the deadlines and arrival times of the rest of the jobs. Specifically, we have the following definition.

Definition 2 *(Adjusted job set) A job set is called an adjusted job set of $\mathcal{J}$, if all jobs in $\mathcal{J}$ having a speed requirement higher than s_{th} (as well as the intervals containing these jobs) are removed, and the arrival times and deadlines of the rest of the jobs are adjusted correspondingly.*

Before we explain our strategy in detail, we also want to introduce several important terminology.

Definition 3 *(Scheduling point) Time t is called a J_n-scheduling point if $t = d_n$ or $t = r_i$, $i < n$ and $r_n < r_i < d_n$.*

As explained before, a job set is delayed only when the processor is idle. Therefore, when identifying the delay that a job can tolerate, we are more interested in the case that the processor is idle when a job arrives. Specifically, we have the following definition.

Definition 4 *(Reduced job set) An adjusted job set is called a J_n-reduced job set if every job J_i in the set satisfies $r_i \geq r_n$.*

We use Figure 2 to illustrate these definitions. Figure 2(c) shows the J_3-reduced job set and all the J_3-scheduling points (as marked by "x"). Note that in Figure 2(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, for each J_n-scheduling point t, the execution of J_n or any higher priority jobs must begin no later than $st_n(t)$, where

$$st_n(t) = t - \sum_{J_k \in hp(J_n)} \frac{c_k}{s_{th}}, r_k < t, \tag{8}$$

where $hp(J_n)$ is the set of jobs with a priority greater than or equal to J_n and arriving before t. It is not difficult to see that different J_n-scheduling points can lead to different $st_n(t)$. Specifically, we have the following Lemma.

Lemma 2 *Let job set ($\mathcal{J}$) be the J_n-reduced job set and $\mathcal{S}(J_n)$ be the set of all J_n-scheduling points. Then,*

$$lst(J_n) = max\{st_n(t), t \in \mathcal{S}(J_n)\}. \tag{9}$$

The corresponding J_n-scheduling point is denoted as $P(lst(J_n))$. The proof for this lemma is trivial according to Definition 1 and is therefore omitted. From Figure 2(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 $lst(J_3) = 8$.

We are interested in finding the latest starting time for a FP job set. Unfortunately, $lst(J_n)$ can only guarantee the feasibility of job J_n but not necessarily any other job in the J_n-reduced job set. For example, as shown in Figure 2(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 its deadline. 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. We call this latest starting time the *effective* latest starting time for the job, denoted as $\tilde{lst}(J_n)$. The above idea is formulated in Algorithm 1.

For the J_n-reduced job set, Algorithm 1 helps to compute the latest time for a J_n-reduced job set. This conclusion is formally presented in the following lemma.

Lemma 3 *Let $\mathcal{J}$ be the J_n-reduced job set. The effective latest starting time ($\tilde{lst}(J_n)$), output from Algorithm 1, 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 $\mathcal{J}$ will meet their deadlines.*

Algorithm 1 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 $\mathcal{J}$.
2:  Output: The effective latest starting time $\tilde{lst}(J_n)$
3:  $nlst = lst(J_n)$; //Equation (9)
4:  $end = P(lst(J_n))$;//the scheduling point corresponding
    to $lst(J_n)$
5:  for $J_k \in \mathcal{J}, k = n+1, n+2, ...$ 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: According to Lemma 2, the schedulability for J_n is guaranteed in line (3) of Algorithm 1 as well as the fact that $nlst$ can only be smaller later on with the progress of the algorithm. Variable end helps to keep track of all lower priority jobs that are potentially preempted when delaying J_n and jobs with a priority higher than J_n to $nlst$. The schedulability for each of these jobs is guaranteed in line (7) for the same reason as that of J_n.

Therefore, to prove Lemma 3, we only need to check if other lower priority jobs (*i.e.*, with a release time later than end during each FOR loop) can meet their deadlines. Consider a lower priority job J_k in one of the FOR loops and let $r_k > end$. Note that, when considering J_k with respect to $nlst$ and end, any job with priority the same or higher than that of J_k 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 $nlst$. □

With Algorithm 1, we can compute the effective latest starting time for each of the jobs in the adjusted job set. For example, in Figure 2(c), we have $\tilde{lst}(J_1) = 8$, $\tilde{lst}(J_2) = 16$, $\tilde{lst}(J_3) = 6$, $\tilde{lst}(J_4) = 10$. Also, we observe the following interesting property of $\tilde{lst}(J_n)$.

Lemma 4 *For adjusted 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

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

According to Lemma 3, 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. $\square$

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 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. However, based on Lemma 3 and Lemma 4, we can derive the following theorem.

Theorem 1 *Given a general job set $\mathcal{J}$ and threshold speed s_{th}, the latest starting time for $\mathcal{J}$ can be computed as*

$$LST(\mathcal{J}) = \min_n\{\tilde{lst}(J_n))\}. \quad (10)$$

where $\tilde{lst}(J_n)$ is r_n if J_n requires a speed higher than s_{th} in the DVS voltage schedule, otherwise $\tilde{lst}(J_n)$ is computed according to Algorithm 1.

Proof: We first assume $\mathcal{J}$ is an adjusted job set. Let $LST(\mathcal{J}) = \tilde{lst}(J_i) = \min_n\{\tilde{lst}(J_n)\}$. We want to prove that any one of the jobs, *i.e.* J_k, can meet their deadlines if job set $\mathcal{J}$ is delayed to $\tilde{lst}(J_i)$.

Note that from Lemma 4, in $\mathcal{J}$, we have for any $k < i$, $r_k \geq r_i$. We consider three different cases separately.

- Case 1: $k < i$.

  Let job r_q be the earliest arrival time for any job J_q such that $q < k$. If we have $r_q \geq r_k$, according to Lemma 3, 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 with respect to $\tilde{lst}(J_q)$. Since $\tilde{lst}(J_q) \geq \tilde{lst}(J_k) \geq \tilde{lst}(J_i)$, J_k can meet its deadline if job set $\mathcal{J}$ is delayed to $\tilde{lst}(J_i)$.

- Case 2: $k = i$.

  The only difference between job set $\mathcal{J}$ and the J_k-reduced job set is that $\mathcal{J}$ may contain some jobs with priorities lower than that of J_k. According to Lemma 3, J_i can meet its deadline since adding any lower priority job to the J_k-reduced job set cannot change the schedulability of J_i and can only decrease $\tilde{lst}(J_i)$.

- Case 3: $k > i$

  If all the jobs arrive later than J_i, Lemma 3 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. Note that, for job J_q such that $i < q < k$, 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 $\tilde{lst}(J_i)$.

When $\mathcal{J}$ is a general job set, any job with speed requirement higher than s_{th} cannot be delayed according to equation (10). Hence, no such job will miss its deadline. In addition, the latest starting time for the rest of the jobs is no later than that computed with the adjusted job set. Therefore, all the jobs can meet their deadlines. $\square$

From Theorem 1, we have $LST(\mathcal{J}) = 6$, which is exactly the case shown in Figure 2(b).

3.3. The algorithm

After studying how long a job set can be safely delayed, we are now ready to present our scheduling strategy to reduce the overall energy consumption. Our approach consists of two phases, an off-line phase and an online phase. In the off-line phase (Algorithm 2), we compute for *each* job, assuming the processor is idle upon the arrival of the job, how long the remaining job set can be delayed; while in the on-line phase of our approach (Algorithm 3), we apply the results produced in the off-line phase and make the scheduling decision on-line.

Algorithm 2 The off-line phase to determine the processor speed (s_n) for each job (J_n), and, assuming J_n is the next arrival job, to compute the maximal delay (δ_n) for the remaining job set.

```
1:  Input: $\mathcal{J}, s_{th}$
2:  Output: $s_n, \delta_n, n = 1, 2, ..., N$
3:  Compute the DVS voltage schedule for $\mathcal{J}$ and thus
    $s_n, n = 1, 2...N$;
4:  for $J_n \in \mathcal{J}$ do
5:    $\mathcal{J}_c = \mathcal{J}$;
6:    // make a copy of $\mathcal{J}$
7:    Remove all $J_i \in \mathcal{J}_c$ with $r_i < r_n$;
8:    $s_n = \max\{s_n, s_{th}\}$;
9:    $T_B = \tilde{lst}(J_n)$;
10:   for $J_k$ with $r_k < T_B$ and $k < n$ do
11:     if $\tilde{lst}(J_k) < T_B$ then
12:       $T_B = \tilde{lst}(J_k)$;
13:       //computed by Algorithm 1
14:     end if
15:   end for
16:   $\delta_n = T_B - r_n$;
17: end for
```

In Algorithm 2, the latest starting time for the job set is computed according to Theorem 1. Note that even though equation (10) requires the computation of the *effective* latest starting time for all jobs, it is not necessary in practice. Note that T_B (in line 8 of Algorithm 2) actually sets up an upper bound on delay, *i.e.* the job set cannot be delayed to any time later than T_B without missing a deadline. Therefore, we only need to check the higher priority jobs (Lemma 4) released before T_B to determine the latest starting time for the job set (line 9-13). In order to do so, we only need to perform a linear scan within the interval from the earliest arrival time to T_B, which has a complexity of $O(N')$, where N' is the total number of higher priority jobs within this interval. The complexity of the rest of the algorithm is also linear related to N'. Since N' is usually very small for a periodic task set, Algorithm 2 typically has a very low computation complexity.

The on-line algorithm follows the principles discussed earlier. It takes the desired processor speed (s_n) and the maximal delay (δ_n) for each job J_n (output from Algorithm 2) as input. When the processor is not idle, it will run the jobs in the ready queue according to the fixed priority scheme; when it is idle, the later jobs will be delayed to the latest starting time (line 7) computed based on the first job arrival. The algorithm is called **FPLK** and illustrated in Algorithm 3. FPLK has a constant time complexity, because it only requires a single table lookup to identify δ_n.

Algorithm 3 (**FPLK**) The on-line leakage conscious fixed-priority scheduling algorithm

```
1:  Input: $(\mathcal{J}, s_n, \delta_n, n = 1, ..., N)$
2:  if processor is not idle then
3:      Run job $J_n$ in the ready queue according to FP, using speed $s_n$;
4:  else
5:      Let $J_n$ be the next coming job;
6:      $nlst = r_n + \delta_n$;
7:      if $nlst - t_{cur} > T_{min}$ then
8:          // $t_{cur}$ is the current time
9:          Shut down the processor and set up the wake up time to be $nlst - t_{cur}$;
10:     end if
11: end if
```

4. Experimental results

In this section, we evaluate the overall energy efficiency of the proposed technique with several experiments. In our experiments, we compare five strategies:

- **No DVS, No Delay (NSND)** The task sets are scheduled without DVS, *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.
- **DVS, No Threshold, No Delay (SNTND)** The task sets are scheduled with DVS but with no consideration of the leakage (*i.e.* the threshold speed), and no task instance is delayed;
- **DVS, Threshold, No Delay (STND)** The task sets are scheduled with DVS, and the jobs are executed with the threshold speed, *i.e.* s_{th}, if its speed requirement is lower than s_{th} in the DVS voltage schedule. However, no job execution is delayed.

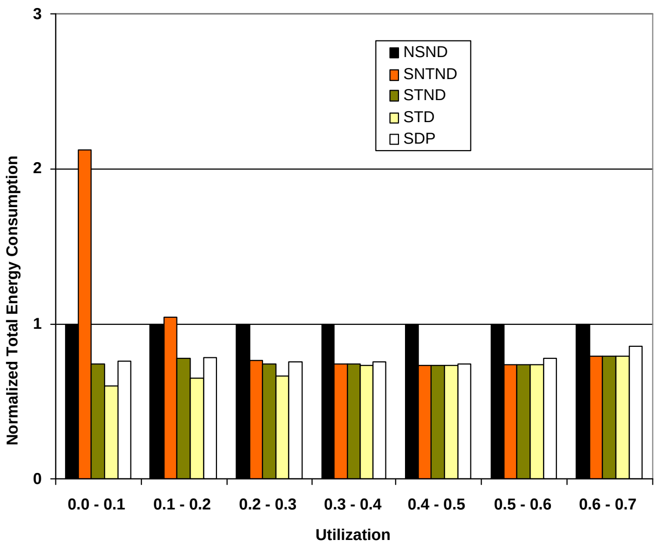


Figure 3. The average total energy consumptions of five different approaches.

- **DVS, Threshold, Delay (STD) – Our approach** Task sets are scheduled with DVS with consideration of both the leakage (*i.e.* the threshold speed) and execution delay (Algorithm 2).
- **DVS, Dual Priority (SDP)** Task sets are scheduled with DVS and delayed with Dual Priority. This is the approach in [30]. We compute the promotion time once for each task based on the worst case scenario and use it for each jobs in the task.

For the algorithms employing DVS above (SNTND, STND and STD), the method in [14] is used to find the off-line voltage schedule under FP scheme. This method is chosen because, while the FP DVS problem is NP-Hard, this heuristic can provide results very close to that by the optimal one [15] in polynomial time ($O(N^3)$). The power model and technology parameters of the processor used in our simulation are adopted from [25]. The threshold speed is around 0.4 [20] for this processor model. For the processor power down/up overhead, we use the same values as that used in [20], *i.e.*, $P_{idle} = 240mW$, $E_o = 483\mu J$, and $t_o = 2ms$.

The real-time systems tested in our experiments are periodic task systems randomly generated with five periodic tasks each. All tasks are scheduled with the RM method. The period of each task is randomly chosen in the range of $[5, 30]$ms. The deadline of each task is set to be equal to its period. We examine the different energy saving performance of the above four approaches for systems with different utilization. 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 50 periodic task sets. For each task set, we collect both overall energy consumption and the idle energy consumption for the task set starting from 0 to the LCM of its periods. We accumulated the values for each utilization interval, normalized[1] the results by **NSND**. The results are shown in Figure 3 and Figure 4, respectively.

From Figure 3, it is interesting to note that using DVS without the consideration of the leakage current (**SNTND**) cannot effectively reduce the overall energy consumed. This is particularly true when the utilization of the task set is low. For example, when the utilization is less than 0.2, the average overall energy consumption with **SNTND** is in fact larger than that by **NSND**. When the utilization is less than 0.1, the average overall energy with "pure" DVS voltage schedule (**SNTND**) is 1.2 times higher than that by **NSND**, and almost as three times as that by the other two strategies. This is because, when the utilization is low, the processor is running at a very low speed in **SNTND**. The processor consumes more energy due to the large leakage current. Also, the overall energy consumption of **STD** is about 19.6% less than **STND** and **SDP**. When the utilization of the task set is high, from Figure 3, the overall energy consumption for **SNTND** seems to be very close to that of **STND**, **STD** or **SDP**. This is because the processor usually has to run at a speed higher than the threshold speed to guarantee the deadlines of the tasks. Therefore, all these strategies use the similar speed most of the time and have similar energy consumption.

Since the leakage power consumption is becoming comparable or even exceeding the dynamic power consumption, the energy consumption during the processor idle time, mainly due to the leakage current, will soon become a significant part of the overall energy consumption. We are therefore interested in investigating how our approach can help to reduce this part of energy consumption compared with other approaches. In Figure 4, it is not surprising to see that **SNTND** consumes very little idle energy since there is less slack time during task execution. Unfortunately, the type of processors required by **SNTND** cannot be built in practice since leakage current is no longer negligible. It is interesting to see that by delaying the execution of the jobs (**STD**), the idle energy is greatly reduced compared with **STND**. The smaller the utilization is, the more energy is saved during idle periods by delaying jobs. When the utilization is low, the threshold speed can be much larger than the speed required for each job and results in a large number of idle intervals. **STD** can effectively merge many of the intervals by delaying the execution of jobs and is therefore a much better approach than **STND**. As shown in Figure 4, when the utilization is within 0.1-0.2, the average idle energy consumed by **STD** is less than 15% of that consumed by **STND**. When the utilization is high, there is only a limited number of idle intervals. In addition, many jobs may require speeds higher than

1 For example, normalizing C to N means using $\frac{C}{N}$ as the value of C for comparisons.

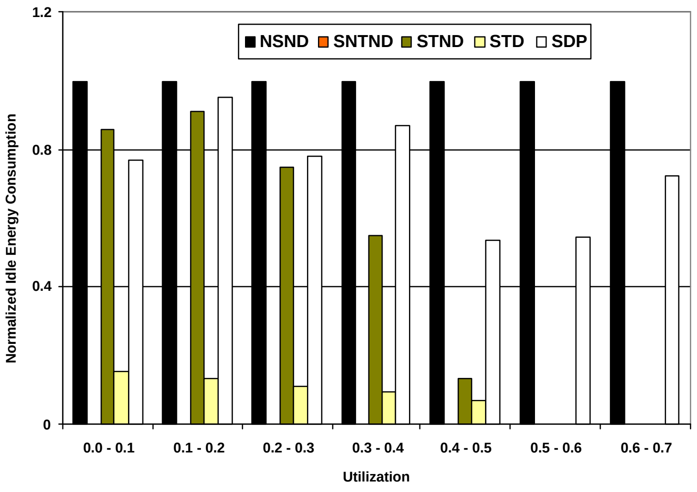


Figure 4. The average idle energy consumptions by five different approaches.

the threshold speed which cannot be delayed at all. Hence, the idle energy efficiency of **STD** is limited. Even so, when the utilization of a task set is within [0.4-0.5], the idle energy consumed using **STD** is, on average, around 50% of that using **STND** as shown in Figure 4. Our experiments also show that pessimistically estimating the delay amount for each job in **SDP** can severely degrade the energy efficiency of this approach. As shown in Figure 4, when the utilization is within 0.3-0.4, the average idle energy consumed by **STD** is less than 10% of that consumed by **SDP**.

5. 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 investigated the problem of applying scheduling techniques to reduce the overall energy consumption for fixed-priority real-time systems.

As shown by our experiments and discussions, 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 DVS voltage schedule may require the processor to adopt a speed higher-than-necessary to avoid the rapidly increasing leakage current at low voltage levels. This may result in a large number of small idle intervals during job execution. We proposed an efficient approach to merge these intervals by delaying the execution of the jobs to reduce the processor shutdown overhead and improve the overall energy performance. In our approach, the maximal delay for each job is statically computed, and is then applied on-line to extend idle intervals. Based on a practical processor model, our experimental results clearly demonstrate that this approach has a great potential in future embedded systems to reduce the overall power consumption. Finally, it is worth mentioning that our approach is a greedy approach. How to achieve the optimal overall energy performance is another very interesting problem and needs further study.

References

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

[2] C. Neau and K. Roy, "Optimal body bias selection for leakage improvement and process compensation over different technology generations," *ISLPED*, pp. 116–121, 2003.

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

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

[5] M. Johnson, D. Somasekhar, and K. Roy, "Leakage control with efficient use of transistor stacks in single threshold cmos," *DAC*, pp. 442–445, 1999.

[6] J. Halter and F. Najm, "A gate-level leakage power reduction method for ultra low power cmos circuits," *CICC*, pp. 475–478, 1997.

[7] F. Assaderaghi, D. Sinitsky, S. A. Parke, J. Bokor, P. Ko, and C. Hu, "Dynamic threshold-voltage mosfet (dtmos) for ultra-low voltage vlsi," *IEEE Trans. on Elec. Dev.*, vol. 44, no. 3, pp. 414–422, Mar 1997.

[8] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," *IEEE Annual Foundations of Comp. Sci.*, pp. 374–382, 1995.

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

[10] 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.

[11] 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.

[12] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," *DAC*, pp. 134–139, 1999.

[13] Y. Shin, K. Choi, and T. Sakurai, "Power optimization of real-time embedded systems on variable speed processors," *International Conference on Computer-Aided Design*, pp. 365–368, 2000.

[14] G. Quan and X. S. Hu, "Energy efficient fixed-priority scheduling for real-time systems on voltage variable processors," *DAC*, pp. 828–833, 2001.

[15] G. Quan and X. Hu, "Minimum energy fixed-priority scheduling for variable voltage processors," *2002 European Design and Test Conference*, 2002.

[16] H.-S. Yun and J. Kim, "On energy optimal voltage scheduling for fixed-prioirty hard real-time systems," *ACM Transactions on Embedded Computing Systems*, vol. vol 2, 2003.

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

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

[19] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," *ISDA*, 2003.

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

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 17, no. 2, pp. 46–61, 1973.

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

[23] R. Davis and A. Burns, "Optimal priority assignment for aperiodic tasks with firm deadlines in fixed-priority preemptive systems," *Information Processing Letters*, vol. 53, no. 5, pp. 249–254, 1995.

[24] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power cmos digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, April 1992.

[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] 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.

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

[28] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Transction On Software Engineering*, vol. 15, 1989.

[29] 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.

[30] R. Jejurikar and R. Gupta, "procrastination scheduling in fixed priority real-time systems," *LCTES*, 2004.