Practical On-line DVS Scheduling for Fixed-Priority Real-Time Systems[1]

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

Gang Quan
Department of CSE
University of South Carolina
Columbia, South Carolina 29208
gquan@cse.sc.edu

Abstract

We present an on-line Dynamic Voltage Scaling (DVS) algorithm for preemptive fixed-priority real-time systems called low power Limited Demand Analysis with Transition overhead (lpLDAT). It is the first algorithm in its class to explicitly account for transition overhead, and can reduce the energy consumption by as much as 40% when compared to previous methods.

1. Introduction

Real-time scheduling plays a key role in the low-power design of real-time embedded systems, not only because timing issues are critical, but also because low power design is essentially a resource usage optimization problem for such systems. How to exploit the modern dynamic configuration capabilities of embedded processors, such as Dynamic Voltage Scaling (DVS), to achieve the most energy efficient designs has become a wide spread research area. There has been substantial research conducted in this area, e.g., [7, 11, 15, 17, 19, 20, 8]. These approaches differ in many aspects, such as the scheduling algorithms being on-line/off-line, handling hard/soft deadline requirements, assuming fixed/dynamic priority assignment, allowing intra-task/inter-task voltage transitions, and single/multiple processor systems. In addition, the applicability of the work to real systems varies widely.

A significant limitation of DVS processors is the inability to change the operation voltage and frequency instantaneously. This limitation, known as *transition time overhead* can be on the order of tens of microseconds ([2, 4]) to tens of milliseconds ([6]). For systems that block execution during a transition (the dominant trend, e.g., [2, 6]), this translates to anywhere from 10^4 to 10^8 lost execution cycles by todays standards and will only worsen as clock speeds continue to scale. Ignoring time overhead in this case will likely cause deadline misses, which in turn results in degraded system performance or even system failure. A related problem is transition energy overhead, which can actually cause the system's energy consumption to increase if DVS is not used judiciously. Despite these limitations, a common practice in the real-time system research community is to focus on the "ideal case" in which time overhead is considered negligible.

In this paper, we study the problem of reducing the energy consumption for fixed-priority periodic real-time tasks executing on a single DVS processor with *non-negligible* time and energy transition overhead via an on-line voltage scheduling technique. Many real-time embedded applications adopt a fixed-priority scheme, such as Rate Monotonic (RM), due to its high predictability, low overhead, and ease of implementation [12]. While an off-line approach can fully leverage known system specifications, using off-line techniques alone may lead to a large waste of energy because, to guarantee timing constraints, one must always consider the worst case. However, the average workload of an application may vary largely from the worst case. Due to the dynamic and reactive nature of embedded systems, an on-line approach that makes scheduling decisions during run time is more robust, flexible, and adaptive.

Some previous research has been conducted regarding this problem, e.g., [7, 11, 15], none of which accounts for transition overhead. Pillai and Shin proposed an algorithm called ccRM, which first computes off-line the maximum speed necessary to meet all task deadlines based on worst-case response time analysis. On-line, the processor speed is scaled down when task instances complete early [15]. Although the ccRM approach guarantees job deadlines, it is not aggressive enough to fully exploit slack in the system when tasks finish closer to their best case execution times. In [7], Gruian presents a method of off-line task stretching coupled with on-line slack distribution. In addition, the paper presents an intra-task voltage scheduling method that computes the optimal speed for each execution cycle of the active task. Similar to ccRM, Gruian's off-line scheme is

1. This work is supported in part by NSF under grants CCR02-08992 and CNS-0410771

conservative. The intra-task method is not useful in practice because it may require the speed to change on a cycle by cycle basis. When considering transition overhead, this method is not feasible. Kim *et al.* in [11] developed a method called **lpWDA** that uses a greedy, on-line algorithm to estimate the amount of slack available and then apply it to the current job. It is unique in that it takes slack from lower priority tasks, as opposed to the methods presented in [7] and [15] that wait for slack to filter down from higher priority tasks. A serious drawback is that it often too aggressive, resulting in wasted energy.

A number of researchers have studied voltage scheduling when transition overhead is not negligible. Mochocki *et al.* present a method that accounts for transition overhead while scheduling a set of jobs using the Earliest Deadline First (EDF) priority scheme off-line ([14]). In [18], Saewong and Rajkumar present an algorithm to schedule fixed priority jobs sets with a very large transition time overhead off-line. AbouGhazaleh *et al.* propose an intra-task voltage scheduling method that uses compiler support and specially designed code to account for transition overhead ([1]). In [8], Hong *et al.* present two algorithms that solve the off-line voltage scheduling problem. The first is optimal and consists of a set of non-linear equations. The second is a more efficient heuristic that iteratively approaches the optimal solution. However, both methods assume that cycles can be executed during a transition, which is often not the case ([2, 6]). Zang *et al.* in [21] present an ILP formulation that optimally solves the voltage scheduling problem for multiple processors while considering transition energy overhead. They also present an approximation formulated as an LP. Both of these methods are too complex to be used on-line. Andrei *et al.* in [3] also solve the voltage scheduling problem for multiple processors using an ILP. Their formulation considers time and energy transition overhead as well as leakage energy dissipation. Again, this method is too complex to effectively adapt to a varying workload on-line. Zhang and Chakrabarty considers both overheads when scheduling voltage levels and checkpoint times for fault-tolerant, hard, real-time systems with periodic tasks ([22]). They assume that each task can meet all deadlines when running at the smallest processor speed if no faults are present. This assumption eliminates the benefit of DVS in a fault free environment.

Existing methods for handling transition overhead are all for off-line algorithms and thus cannot be readily used on-line. To the best of our knowledge, none of the existing on-line voltage scheduling algorithms can guarantee task deadlines when time overhead must be considered. In this paper, we present our algorithm, called low power Limited Demand Analysis with Transition overhead (lpLDAT) that explicitly accounts for both time and energy transition overhead. Through experimentation we demonstrate that our proposed approach can result in an energy reduction of more than 40% when compared to previous methods.

The remainder of this paper is organized as follows. Section 2 summarizes the background material, Section 3 describes our algorithm, Section 4 presents the experimental results and Section 5 concludes the paper.

2. Preliminaries

In this section, we first specify the type of systems under consideration and introduce the necessary notation. Next we briefly review low power Work Demand Analysis (lpWDA). A motivational example is given to illustrate the difficulties associated with accounting for transition overhead and also to show why lpWDA is not adequate in harvesting maximally the benefit provided by DVS.

2.1. System Model

We consider real-time applications consisting of a set of n periodic tasks, $\mathcal{T} = \{T_1, T_2, \cdots, T_n\}$. Task T_i is said to have a *higher priority* than task T_j if $i < j$. Each task, T_i, is described by its worst case execution cycles, wc_i, average case execution cycles, ac_i, and best case execution cycles, bc_i, with $wc_i \geq ac_i \geq bc_i$. In addition, each task has a period, p_i, and relative deadline, d_i, with $d_i \leq p_i$. The *utilization* of a task set is the sum of each task's worst case cycles over its period. That is, the worst-case utilization can be computed as

$$U_{wc} = \sum_{i=1}^{n} \frac{wc_i}{p_i}. \tag{1}$$

The average-case utilization, U_{ac}, and the best-case utilization, U_{bc}, can be computed similarly. Each task is invoked periodically and we refer to the k-th invocation of task T_i as job J_i^k. Each job is described by a release time, r_i^k, deadline, d_i^k, finish time, f_i^k, the number of cycles that have already been executed, ex_i^k, and *actual* total execution cycles, c_i^k, with $0 \leq ex_i^k \leq c_i^k$ and $bc_i \leq c_i^k \leq wc_i$. During run-time, we refer to the earliest job of each task that has not completed execution as the **current** job for that task, and we index that job with cur, e.g., J_i^{cur} is the current job for task T_i. The *estimated* work remaining for job J_i^{cur}, denoted by w_i^{cur}, is equal to $wc_i - ex_i^{cur}$. As in most DVS work, we assume that each job consumes an equal amount of energy per cycle at a given speed, which is a valid assumption for many applications. A **ready job** is any job J_i at time t that satisfies $r_i \leq t$ and $d_i > t$ and $f_i > t$, while the **Active Job** is the ready job at time t with the highest priority.

A **scheduling point** is any time point t that satisfies either $t = r_i^k$, $t = d_i^k$ or $t = f_i^k | i = 1..n, k = 1..\infty$. We use $\mathcal{TS}$ to represent all scheduling points sorted in ascending order. An individual scheduling point is indexed by i

and denoted by ts_i. Note that finish times are not known off-line, and are inserted into $\mathcal{TS}$ as they occur. Once a finish time is inserted, the corresponding deadline is removed from $\mathcal{TS}$. The subset of $\mathcal{TS}$ that includes all points greater than r_i^k and less than or equal to d_i^k is called the set of J_i^k-scheduling points and is denoted by $\mathcal{TS}_i^k$.

The DVS processor used in our system can operate at a finite set of voltage levels $\mathcal{V} = \{V_1, ..., V_{max}\}$, each with an associated speed. To simplify the discussion, we normalize the processor speeds by S_{max}, the speed corresponding to V_{max}, giving $\mathcal{S} = \{S_1, ..., 1\}$. Changing from one voltage level to another takes a fixed amount of time, referred to as the *transition interval* (denoted Δt) within which no tasks can be executed. The transition interval length for a DVS processor alone is usually on the order of 10 to 120 μs ([9, 2, 5, 16]). This results from the DC-DC converter changing V_{DD} and the phase-locked loop (or similar technology) changing f_{clk}. However, when considering synchronization with other components in a system, such as off chip memory, the length can be on the order of milliseconds ([18, 6]).

A voltage transition also consumes a variable amount of *transition energy*, denoted as ΔE. Transition energy includes three major parts: (1) the energy consumed by the DC/DC converter, (2) the energy consumed by the CPU during the transition and (3) the energy increase due to executing cycles displaced by the transition interval at higher speeds. This is similar to the model used in [14].

The above DVS processor model captures the main properties of most commercial DVS processors. A variable length transition interval (e.g. the one described in [5]) can be approximated by a fixed length interval equal to the maximum switching time. For processors that do not block instructions during a transition (e.g., [5]), a schedule that assumes blocking during a transition can be pessimistic, but it will guarantee that a valid voltage schedule is reached.

In the context of on-line scheduling algorithms, at each scheduling point there is a particular set of speeds, called feasible speeds, that are necessary to meet job deadlines. The following is a formal definition of this set.

Definition 1. ***Feasible Speed***– *Any speed less than or equal to S_{max}, which guarantees that if execution of J begins by at least time t at that speed, then all remaining cycles of J will be completed by its deadline.*

Because any feasible speed will guarantee the deadline of its associated job, it is tempting to select the smallest speed to save the most energy. However, this may cause lower priority jobs to miss their deadlines, because they cannot be executed until the higher priority job is complete. To alleviate this problem, we introduce effective feasible speeds.

Definition 2. ***Effective Feasible Speed***– *Any feasible speed of J_i at time t that guarantees the existence of a feasible speed of job J_k at time f_i (where f_i is the completion time of J_i) for every job J_k with $i \leq k$.*

Another interpretation of the effective feasible speed is that a higher priority job running at an effective feasible speed will not steal more slack than is available from lower priority jobs, thus allowing all lower priority jobs to also execute at feasible speeds when they become the active job.

2.2. Low-Power Work Demand Analysis (lpWDA)

To help put our contributions in perspective, we briefly review the on-line DVS algorithm called **lpWDA**, given in Algorithms 1 and 2 (for more details on lpWDA, see [11]). For now, ignore lines marked by ***. Algorithm lpWDA works in the following manner. First, the system is initialized by setting the execution cycles and deadlines of each task and by setting the initial values of H, where H_i is an *over estimate* of the higher priority cycles that must be executed before d_i^{cur} (Lines 1–4). Next, on each preemption or completion, the remaining cycles of the preempted or completed job (w_α^{cur}) and the estimates of higher priority cycles are updated by the **updateLoadInfo** algorithm. Finally, when a job J_α^{cur} is scheduled for execution, the processor speed is scaled according to the amount of slack available (see Lines 8–10). Essentially, lpWDA takes all the slack that it can steal from lower priority tasks in linear time and applies that slack to the active job. Lemma 1 and Theorem 1 show that lpWDA will always produce a valid schedule as long as transition time overhead is negligible.

Algorithm 1 lpWDA ***(lpLDA with ***)***

```
 1: if on system start then
 2:     for Each Task $T_i \in \mathcal{T}$ do
 3:         $d_i^{cur}$:=$d_i$; $w_i^{cur}$:=$wc_i$;
 4:         $H_i$:=$\sum_{j=0}^{i-1}(\lceil \frac{d_i^{cur}}{p_j} \rceil \times wc_j)$;
 5:         *** $A_i$:=$\sum_{j=0}^{i-1}(\lceil \frac{d_i^{cur}}{p_j} \rceil \times ac_j)$;
 6: if finish/preempt the active job $J_\alpha^{cur}$ then
    updateLoadInfo($\mathcal{T}$,$\alpha$);
 7: if on execute the active job $J_\alpha$ then
 8:     Identify $T_\beta | \beta \leq \alpha$ AND $d_\beta^{cur}$ is minimized;
 9:     Compute $slack_\alpha$ based on workload with respect to
        $T_\beta$;
10:     $f_{clk} := \frac{w_\alpha^{cur}}{slack_\alpha + w_\alpha^{cur}} \times f_{max}$;
11:     *** $f_{ACL} := max\{\frac{A_i + ac_\alpha - ex_\alpha^{cur}}{d_i^{cur} - t} | i = 1..n\}$;
12:     *** $f_{clk} := max\{f_{clk}, f_{ACL}\}$;
13:     Set the voltage according to $f_{clk}$;
```

Lemma 1. *Algorithm lpWDA selects an effective feasible speed for the active job at every scheduling point.*

Algorithm 2 updateLoadInfo($\mathcal{T}$,α)

```
1: input: Tasks $\mathcal{T}$ and the preempted/completed task index $\alpha$.
2: output: Workloads are updated to reflect current execution information.
3: if $T_\alpha$ is completed then
4:     for each task $T_i \in \mathcal{T}$ with $i < \alpha$ do
5:         $d_\alpha^{cur} := d_\alpha^{cur} + p_\alpha$;
6:         $H_\alpha := H_\alpha + \sum_{j=0}^{\alpha-1}(\lceil \frac{d_i^{cur}}{p_j} \rceil - \lceil \frac{d_i^{cur}-p_\alpha}{p_j} \rceil) \times wc_j$;
7:         *** $A_\alpha := A_\alpha + \sum_{j=0}^{\alpha-1}(\lceil \frac{d_i^{cur}}{p_j} \rceil - \lceil \frac{d_i^{cur}-p_\alpha}{p_j} \rceil) \times ac_j$;
8:     for each task $T_i \in \mathcal{T}$ with $i > \alpha$ do
9:         $H_i := H_i - (wc_\alpha - ex_\alpha^k)$;
10:        *** $A_i := A_i - max\{0, ac_\alpha - ex_\alpha^k\}$;
11:    $w_\alpha^{cur} := wc_\alpha$; // reset for next job of $T_\alpha$
12: else
13:    $temp := wc_\alpha - ex_i^{cur}$;
14:    for each task $T_i \in \mathcal{T}$ with $i > \alpha$ do
15:        $H_i := H_i - w_\alpha^{cur} + temp$;
16:        *** $A_i := A_i - w_\alpha^{cur} + temp$;
17:    $w_\alpha^{cur} := temp$;
```

Proof. See [11]. □

Theorem 1. *The schedule produced by* **lpWDA** *will guarantee all system deadlines, and has a computational complexity of $O(n)$ per scheduling point, where n is the number of tasks in the system.*

Proof. According to Lemma 1, the speed selected by *lpWDA* is always an effective feasible speed. It is trivial to see that all deadlines will be met in this case. For a proof of the computational complexity, see [11]. □

2.3. Motivational Example

Although lpWDA is effective in estimating the amount of slack available to the active job, this slack estimation cannot be easily modified to account for transition time overhead. Observe the two-task system in Figure 1, where the task parameters are given at the top of the figure. Figure 1(a) shows the task execution schedule when tasks always take the worst-case execution cycles and lpWDA is used. Next, assume a transition interval of $\Delta t = 1.2$ (i.e., 120 μs). Note that lpWDA does not explicitly account for time overhead. We could try to solve this problem by reducing the slack identified in Line 9 of lpWDA by Δt time units during every slack calculation. The result of this modification is depicted in Figure 1(b).

Notice that the idle time from 8 to 10.5 consists of two transition intervals and the deadline of J_2^3 is missed at time 12. After the slack calculation at time 8, the scheduler finds that J_2^3 can run a the speed of 0.56 to complete by its deadline. After the transition from a speed of 1.0 to a speed of

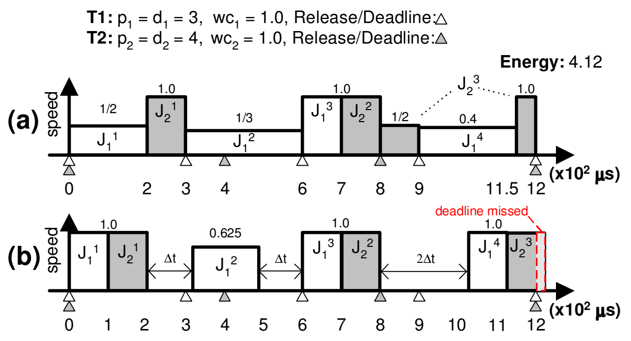


Figure 1. An example task set consisting of two tasks (a) scheduled by lpWDA, (b) scheduled by lpWDA with a simple modification to account for transition time overhead.

0.56 at time 9.2, J_1^4 preempts J_2^3. The scheduler then calculates a speed of 1.0 for J_1^4 after reducing its slack by 1.20. However, 1.0 is no longer sufficient to meet the deadline of J_2^3. Clearly, when transition time overhead is significantly large, one cannot be too aggressive when employing DVS, otherwise deadlines will be missed. Additionally, one can easily verify that executing at the speed 2/3 without any transitions will meet all job deadlines, and will consume only 2.22 units of energy, less than lpWDA without time overhead. Including transition energy overhead will only increase the deficiencies of lpWDA.

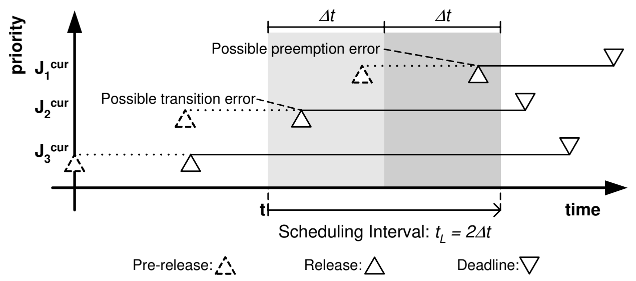
In the next section, we first present our method to correctly account for transition time overhead and then prove that no deadlines will be missed. Next, we improve upon lpWDA to reduce system energy consumption by as much as 40%.

3. Our Approach

First we develop a method to correctly account for transition time overhead. Next, we develop an on-line algorithm that employs this method and uses stochastic information about each task to reduce energy consumption when compared to previous methods. Finally, we account for transition energy overhead to improve the energy performance even more.

3.1. Transition Time Overhead

Transition time overhead can complicate voltage scheduling in several ways. The most straightforward effect is on slack utilization, since time overhead essentially reduces available slack. To guarantee deadlines, we will always reduce the estimated slack to ensure that there is enough time to make a transition now to a speed lower than S_{max}, and also make another transition to S_{max} later when



priority
J_1^{cur}
J_2^{cur}
J_3^{cur}
Δt
Δt
Possible preemption error
Possible transition error
t
time
Scheduling Interval: $t_L = 2\Delta t$
Pre-release:
Release:
Deadline:

Figure 2. An example of pre-release points, the lookahead interval at time t, the system lookahead (t_L) and possible transition and preemption errors. The critical job at time t is J_1^{cur}.

necessary. Second, a job may be released during a transition. This happens in the task set of Figure 1 when $\Delta t > 1$. We refer to this problem as a *transition error*. The presence of a transition error may induce an unexpected voltage transition. Third, notice that lpWDA only checks the deadlines of tasks with an equal or lesser priority than the currently executing job (Line 8 of Algorithm 1). This means that we could scale down to a speed that guarantees the current job, only to be preempted later by a higher priority job that requires a higher speed. Recall that this situation occurs in Figure 1(b) at time 9.2. This is not a problem when transitions happen instantly, but with time overhead we must be more careful. We refer to this problem as a *preemption error*. These scenarios are illustrated graphically in Figure 2.

To deal with time overhead we must look ahead and predict potential future necessary transitions. How much *lookahead*, is needed deserves careful examination since too much will increase scheduling complexity and too little may not be sufficient for meeting deadlines. To simplify our discussion, we first present the following formal definitions.

Definition 3. ***Pre-release Scheduling Point***– *Any time point t that satisfies $t = r_i^k - \Delta t | i = 1..n, k = 1..\infty$. Pre-release scheduling points replace the corresponding scheduling points that satisfy $t = r_i^k | i = 1..n, k = 1..\infty$ in $\mathcal{TS}$ whenever time transition overhead is not negligible (see Figure 2).*

Pre-release scheduling points warn the system that a preemption may occur in the near future, and are essential if a feasible speed is to remain feasible after the voltage transition. However, the pre-release points alone will do no good if the associated job is not included in the scheduling process. To this end, we introduce the concepts of a *lookahead interval* and the *critical job*.

Definition 4. ***Lookahead Interval***– *An interval that begins at a specific scheduling point, ts_i, and ends at time $ts_i + t_L$. The value t_L is referred to as the system lookahead. The lookahead interval at time ts_i is denoted by L_i (see Figure 2).*

Definition 5. ***Critical Job***– *The job with the highest priority that is ready any time during a particular lookahead interval. The critical job of the lookahead interval L_i is denoted by $JC(L_i)$ (see Figure 2).*

Identifying the *critical job* of each *lookahead interval* is the fist step in identifying a valid voltage schedule with a non-negligible transition time overhead. Once these two parameters are known, the final step is to define a clear goal for the scheduler. Our goal will be finding an *overhead feasible speed*.

Definition 6. ***Overhead Feasible Speed***– *An effective feasible speed associated with a job J_i at time t that further satisfies one of the following:* ***(a)*** *is equal to the current speed of the processor and permits an idle interval of length Δt that begins in the range $[t, f_i]$,* ***(b)*** *is different from the current speed of the processor and permits two idle intervals of length Δt, one that begins at t and a second that begins in the range $[t + \Delta t, f_i]$,* ***(c)*** *is different from the current speed of the processor, is equal to S_{max} and permits one idle interval of length Δt that begins at t, or* ***(d)*** *is the same as the current speed of the processor and is equal to S_{max}. The overhead feasible speed of job J_i at time t is denoted by $O(J_i, t)$,*

Next, we describe our method to account for time overhead using the terminology just defined. First, pre-release scheduling points are inserted to ensure that scheduling occurs with enough time to have a speed/voltage transition. Second, when a scheduling point is encountered, instead of selecting the speed according to the active job, we identify the critical job using a system lookahead of $2\Delta t$. Looking ahead $1\Delta t$ prevents transition errors, while looking ahead an additional Δt ensures that when a transition is complete, a higher priority job that requires a larger speed is not closer than Δt away from the current time, thus preventing possible preemption errors directly after a transition (See the case of J_1^{cur} in Figure 2). Lemma 2 shows that an overhead feasible speed always exists if $t_L = 2\Delta t$, regardless of the length of the transition interval.

Lemma 2. *Given a transition time overhead of Δt, a schedulable set of tasks, $\mathcal{T}$, and an initial speed of S_{max}. If the system lookahead, t_L, is set to $2\Delta t$, then an overhead feasible speed exists at each scheduling point.*

Proof. The proof is by induction on $\mathcal{TS}$.

BASE CASE: At ts_1, at least one overhead feasible speed exists, i.e. S_{max}, because $\mathcal{T}$ is schedulable and the initial speed is S_{max}.

INDUCTION STEP: Assume that the current time is ts_i, and that an overhead feasible speed was selected at the previous scheduling point, ts_{i-1}, for the previous critical job, $JC(L_{i-1})$. There are two possibilities for $O(JC(L_{i-1}), ts_{i-1})$: (1) it is equal to S_{max}, or (2) it is less than S_{max}. In case 1, S_{max} is always an overhead feasible speed for $JC(L_i)$. In case 2, we need to consider three possible outcomes of scheduling at ts_i: (i) $JC(L_{i-1})$ completes its execution, (ii) Only lower priority jobs are released in L_i, and (iii) One or more higher priority jobs are released in L_i.

(i) Due to the definition of an overhead feasible speed and the fact that the current speed is overhead feasible, there must at least be time now for one idle interval of size Δt. This means that part (c) of the definition of an overhead feasible speed is valid and S_{max} is an overhead feasible speed of $JC(L_i)$.

(ii) $JC(L_{i-1})$ is the highest priority task and is, therefore, also $JC(L_i)$. The current speed must still be overhead feasible based on part (a) of the definition of an overhead feasible speed.

(iii) $JC(L_{i-1})$ will be preempted sometime during L_i by $JC(L_i)$. Assume an overhead feasible speed for $JC(L_i)$ does not exist. This means either that (a) there is no speed that can guarantee a feasible speed will exist for lower priority jobs when they are ready to execute, or (b) no speed less than or equal to S_{max} can guarantee the deadline of $JC(L_i)$. We know (a) cannot be true, because the formulation of $O(JC(L_{i-1}), ts_{i-1})$ provides that guarantee. If (b) is true, this means either $\mathcal{T}$ is not schedulable, a contradiction, or the execution of some cycles of $JC(L_i)$ will be delayed. The only way for cycles of $JC(L_i)$ to be delayed is the Δt idle time that is introduced by the transition from $O(JC(Li-1), ts_{i-1})$ to $O(JC(L_i), ts_i)$. Recall that there is a pre-release scheduling point exactly Δt time before each job release. If scheduling occurs at the pre-release point of $JC(L_i)$, then its cycles will not be delayed. The only way for scheduling not to occur at the pre-release point is if the pre-release point is inside the transition interval that occurred at the previous scheduling point, ts_{i-1}. However, because the lookahead is equal to $2\Delta t$, if the pre-release point of $JC(L_i)$ is inside the previous transition interval, then the release time of $JC(L_i)$ is inside L_{i-1}. This means that $JC(L_i)$ was not chosen to be the critical job instead of $JC(L_{i-1})$, even though it has a higher priority and was active during L_{i-1}, a contradiction (J_1^{cur} in Figure 2 illustrates this point). □

3.2. Limited Demand Analysis

Now that we have a detailed description of a class of algorithms that correctly account for time overhead, we will develop an algorithm, which is part of this class while at the same time consumes less energy than previous methods.

As shown in Figure 1(a), lpWDA is effective at computing the slack of lower priority jobs, but may suffer from high energy spikes when executing near the deadlines of lower priority tasks. An effective algorithm would utilize any available stochastic information of the given task set to prevent these spikes.

Limiting the slack used by higher priority tasks in lpWDA requires a careful trade-off between being aggressive and being conservative. If one could compute an efficient speed based on the average-case workload, this speed could be used as a limiter. By limiter we mean that, if this speed is higher than the speed predicted by lpWDA, we know that lpWDA is being too aggressive in stealing slack from lower priority jobs and the limiting speed should be used.

In off-line voltage scheduling algorithms, an often used concept is the *minimum constant speed* that can meet all job deadlines (e.g., [15, 17, 20]). Due to the convexity of the power function, it is generally not energy efficient for the processor to go below this speed and then switch to a higher speed later, unless there is reason to expect newly available slack ([15]). Thus the minimum constant speed can serve as a proper limiter. To find the minimum constant speed for a periodic task system where every job of a task assumes the same execution cycles, one can simply examine the case when all tasks are released simultaneously, i.e.,

$$S_{MC} = \max_{i=1}^{n} \min_{ts \in \mathcal{TS}_i^1} Speed(i, ts) \tag{2}$$

and

$$Speed(i, ts) = \frac{\sum_{j=1}^{i} \lceil \frac{ts}{p_j} \rceil \times wc_j}{ts} \tag{3}$$

where $\mathcal{TS}_i^1$ is the set of J_i^1-scheduling points.

Our idea is to perform a similar operation as above online. Directly applying the formulas in (2) and (3) is not desirable due to its pessimism and time complexity. To overcome unnecessary pessimism, we recompute the minimum constant speed for each job whenever it starts/resumes execution. This allows the actual execution cycles of jobs executed earlier to be considered when appropriate. This also removes the pessimistic assumption of the worst-case phasing. Furthermore, instead of using the worst-case execution cycles, we use the *average*-case execution cycles. Finally, we opt to use the deadline d_i^{cur} of job J_i^{cur} rather than checking every scheduling point in $\mathcal{TS}_i^{cur}$ for the minimum speed. This reduces the time needed to calculate the limiter.

The necessary changes to lpWDA are marked by *** in Algorithms 1 and 2. We refer to this addition to lpWDA as the Average Case Limiter (ACL). In Algorithm 1, we add Line 5 which initializes the average number of cycles that must be completed before each job deadline. Lines 7, 10

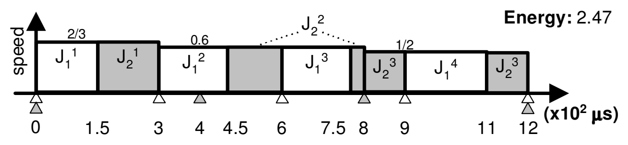


Figure 3. The schedule produced by lpLDA when executing the task set from Figure 1.

and 16 in Algorithm 2 ensure that the current phasing and execution information is stored. Line 11 in Algorithm 1 calculates the speed required by each job to meet its deadline on average. Finally, Line 12 of Algorithm 1 selects the maximum of the speeds requested by lpWDA and the limiter, essentially restricting the amount of slack that lpWDA can use. We refer to this algorithm as low-power Limited Demand Analysis (lpLDA). Applying lpLDA to the example task set in Figure 1 produces the results in Figure 3 when all jobs require their worst case cycles. Notice that the energy is over 40% less than the schedule in Figure 1. Lemma 3 and Theorem 2 state the correctness of lpLDA in terms of satisfying real-time requirements.

Lemma 3. *Algorithm lpLDA selects an effective feasible speed for the critical job at every scheduling point.*

Proof. The speed selected by lpLDA is always greater than or equal to the speed selected by lpWDA. It follows from Theorem 1 that the speed selected by $lpLDA$ is also an effective feasible speed for the critical job. □

Theorem 2. *The schedule produced by lpLDA will guarantee all system deadlines, and has a computational complexity of $O(n)$ per scheduling point, where n is the number of tasks in the system.*

Proof. The deadline guarantee is provided by Lemma 3. The computational complexity is on the same order of lpWDA (only a constant factor larger), which is $O(n)$ by Theorem 1. □

Note that lpLDA still suffers from the same drawback as lpWDA with regard to transition time overhead. However, all we need to do to create a valid algorithm is to (i) replace release scheduling points with pre-release scheduling points, (ii) set the system lookahead to $2\Delta t$ and (iii) select an overhead feasible speed for the critical job of each scheduling point. The implementation of this algorithm, known as lpLDAt, is given in Algorithm 3. For now ignore lines marked with ***.

Theorem 3. *The schedule produced by* **lpLDAt** *will guarantee all system deadlines, and has a computational complexity of $O(n)$ per scheduling point, where n is the number of tasks in the system.*

Algorithm 3 lpLDAt *(lpLDAT with ***)*

```
1:  if system start then
2:      Initialize each task as lpLDA;
3:      *** $energyS_{max}$ := estimate($\mathcal{T}$,$S_{max}$);
4:      *** $energyS_{MC}$ := estimate($\mathcal{T}$,$S_{MC}$);
5:      $S'_{max} := S_{max}$;
6:      *** if $energyS_{max} >= energyS_{MC}$ then $S'_{max} := S_{MC}$;
7:      $f_{clk} := S'_{max}$;
8:  if current time $ts \in \mathcal{TS}$ and $ts$ is a job completion/pre-release then
9:      Find $J_\alpha$, the active job;
10:     updateLoadInfo($\mathcal{T}$,$\alpha$);
11:     $J_\alpha$ := JC([$ts$,$ts + 2\Delta t$]);
12:     $t := max\{ts, r_\alpha^{cur}\}$;
13:     Compute $slack_\alpha$ based on workload starting at $t$;
14:     if $slack_\alpha$ is large enough for 2 transitions at the speed $S_{2\Delta t}$ where $S_{2\Delta t} \leq S'_{max}$ then $f_{clk} := S_{2\Delta t}$;
15:     else if $slack_\alpha$ is large enough for 1 transition at the speed $S_{\Delta t}$ where $S_{\Delta t} \leq f_{prev}$ and $f_{prev}$ is the previous speed then $f_{clk} := f_{prev}$;
16:     else $f_{clk} := S'_{max}$;
17:     if $f_{clk} < f_{ACL}$ AND $f_{clk} \neq f_{prev}$ then $f_{clk} := f_{ACL}$;
18:     *** if $f_{clk} < f_{prev}$ then check_energy_overhead();

19:     Set the voltage according to $f_{clk}$;
```

Proof. All we need to show is that the policy adopted by lpLDAt conforms to the policy described in Lemma 2. First, the speed is initialized to S_{max} (lines 5 and 7). When a scheduling point ts_i is encountered (line 8), the critical job J_α is selected by scanning each task and finding the highest priority task that has a ready job in the interval $[t, t + 2\Delta t]$ (line 11). Next, lpWDA is used to estimate the slack available to J_α (line 13). The speed calculated for J_α using this slack estimation is an effective feasible speed for J_α based on Lemma 1. Next, the speeds for parts (a) and (b) of Definition 6 are calculated by reducing the estimated slack by one or two transition intervals (lines 14–16). Even if there appears not to be enough slack to allow for an overhead feasible speed of type (a) or (b) based on the definition, there actually is slack available to switch the speed to S_{max} (parts (c) and (d) of Definition 6). This is due to two facts: (1) The speed is not lowered unless a previous slack estimation allows for a transition to the lower speed and one back to S_{max} later and (2) lpWDA always estimates less slack than is actually available. If a speed different from the current speed and lower than S_{max} is selected, then the processor is set either to that speed or the ACL, whichever speed is higher. The selected speed therefore matches Definition 6 based on the *actual* slack available for speed transitions and lpLDAt follows the policy outlined in Lemma 2.

Finding the critical job takes one comparison for each task in the system. The speed decision step takes $O(n)$ time for the slack estimation (because it uses lpWDA) and constant time to select the correct speed based on the slack. Calculating the ACL also takes $O(n)$ time. Hence, the overall time complexity is $O(n)$. $\square$

3.3. Dealing with transition energy overhead

Extra energy will be consumed during a voltage transition because (1) a voltage transition itself consumes energy and (2) voltage transitions take a fixed amount of time within which no jobs can be executed. Therefore the processor needs to adopt a higher speed elsewhere to accommodate this interval.

To illustrate effect (2) from above, let us examine the data in Figure 4(a). This figure shows the energy consumed by executing the task set from Figure 1 at the speeds selected by lpLDAt for various sizes of Δt, and various values for bc/wc of each task. The solid black line represents the energy consumed when always executing at the minimum constant speed, i.e., S_{MC} according to equations (2) and (3). All energy numbers are normalized against the energy consumed when executing at the maximum processor speed, S_{max}, without DVS. The reader will immediately notice that as Δt grows, the benefit gained from DVS quickly vanishes. Such results are due to the fact that lpLDAt aggressively varies the processor speed to exploit slack, which can adversely introduce more transitions than necessary when transition overhead is not negligible.

To ensure that time overhead does not cause an energy increase over using just S_{MC}, we propose exploiting slack less aggressively, thus avoiding transitions that would increase energy instead of reducing it. The maximum speed, S_{max} (which is assumed to equal 1 when normalized) is used implicitly in Lines 9 and 10 of Algorithm 1 when determining f_{clk}. This leads to an "overestimation" of slack time when transition overhead is not negligible. If we choose a lower speed for S_{max}, it can be readily used to scale these workload values and will result in a more conservative slack exploitation. We refer to the adjusted maximum speed as S'_{max}.

An interesting problem is how to select this S'_{max}. One may be tempted to select S_{MC} as the speed for S'_{max}. Though this ensures that no curve will appear above the S_{MC} line in Figure 4 and also guarantees all task deadlines, doing so would drastically reduce the amount of slack available when Δt is zero and jobs finish much earlier than the worst case. This situation is illustrated in Figure 4(b). Setting S'_{max} to a speed between S_{MC} and S_{max} is not efficient in general. This is because the reason that we scale back the S_{max} is that we expect there to be very little slack available (due to near worst-case cycles or high Δt). Because there is little slack, we expect to execute at S'_{max} a large percentage of the time. If this is the case then the lower S'_{max} is the less energy the system will consume. The decision is made by estimating the energy that lpLDAt will consume at each speed, which is done by simulating the execution of tasks up to the system hyperperiod. During this process, we assume that every job instance requires the average case execution cycles. The process is repeated twice, once with $S'_{max} = S_{max}$ and once with $S'_{max} = S_{MC}$. The level that consumes less energy during this estimation step is selected.

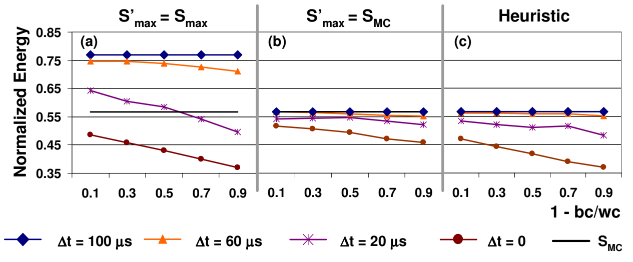


Figure 4. The energy consumed when applying (a) lpLDAt with $S'_{max} = S_{max}$, (b) lpLDAt with $S'_{max} = S_{MC}$ and (c) lpLDAT to the task set from Figure 1.

Algorithm 3 with the lines marked by *** represents the new algorithm, called lpLDAT. Here we will focus on the parts that are different from lpLDAt. Lines 3–6 determine the adjusted maximum speed S'_{max} as described above. This step occurs off-line.

On-line, after a new speed is selected, if the selection results in a voltage transition from S_i to S_j, then there is one final check (Line 18), which ensures that ΔE doesn't locally dominate the energy saved by changing the voltage level. If executing the workload of J_{α}^{cur} at S_i consumes less energy than executing at $S_j + 2\Delta E$ and $S_i > S_j$, then the voltage transition is rejected. Otherwise, S_j is adopted as the new processor speed. Theorem 4 states the correctness of Algorithm 3.

Theorem 4. *The policy followed by **lpLDAT** will guarantee all system deadlines, and has a computational complexity of $O(n)$ per scheduling point, where n is the number of tasks in the system.*

Proof. Theorem 3 states that lpLDAt guarantees all deadlines. Algorithm lpLDAT is identical to lpLDAt, with two key differences. First, S_{max} is conditionally set to the minimum constant speed that meets all deadlines under the worst case phasing condition off-line. If S_{max} is scaled down, then Lemma 3 still holds for lpLDAT, because the system is still schedulable at the new speed S'_{max}. The second change accounts for transition energy overhead. The modification is to only scale down to a lower speed if the energy overhead of the transition is offset by the energy gained from executing at a lower speed. Because the alternative is executing at a higher speed than the identified overhead-feasible speed, the alternative speed is also overhead-feasible. The first modification is off-line, so it doesn't alter the on-line complexity. The energy estimation for the second modification takes constant time, so the complexity of lpLDAT is $O(n)$. □

4. Experimental Results

In this section we quantify the effectiveness of lpLDAT on several real-world and randomly generated task sets, and compare its energy consumption with ccRM, lpWDA, FC and SMC. Both ccRM and lpWDA were modified to account for time transition overhead. SMC simply runs at S_{MC} and enters the sleep state when idle. FC, the off-line algorithm from [17], is used as reference lower bound to all three algorithms. When scheduling with FC, it is assumed that $\Delta t = 0$ and exact job execution cycles are known.

The processor model we use is representative of the ARM8 core. For all experiments we assume there are 32 frequency levels available in the range of 10 to 100 MHz, with corresponding voltage levels of 1 to 3.3 Volts. When idling, the processor is assumed to consume one half the power consumed when executing at the minimum processor speed. Transition energy overhead is modeled using $\Delta E = \eta \times C_{DD} \times |V_1^2 - V_2^2|$, with $\eta = 0.9$ and $C_{DD} = 5$ μF as presented by Burd in [4]. The energy of all the results presented in this section are normalized against a processor running at the maximum processor speed without DVS.

First, each algorithm was applied to two real-world examples: A Computerized Numeric Controller (CNC) task set based on the work by Kim *et al.* in [10] and an avionics task set based on Locke's work in [13]. For each task set, Δt was varied from 0 to 300 μs in 100 μs steps. The results are displayed in Figures 5(a–d) and 5(e–h) respectively.

For the CNC tasks set, lpLDAT is always as good as or better than ccRM and lpWDA. Note that instead of displaying $\frac{bc}{wc}$ on the x-axis, we display $1 - \frac{bc}{wc}$. With $\Delta t = 0$, lpLDAT is within 2% of the lower bound and consumes as much as 29% less energy than ccRM or lpWDA.

The second set of experiments was conducted on randomly generated task sets with the number of tasks per set varied from 2 to 10 in two task increments. Each grouping has 100 separate task sets, with periods and deadlines uniformly distributed in the range [1, 100] ms, hyper periods less than or equal to 5 s, and a U_{wc} normally distributed in the range [0.2, 0.8]. Δt is assumed to be 100 μs. The results are given in Figure 6. Clearly lpLDAT outperforms the other two algorithms in all cases, and the margin of its improvement increases with the number of jobs in the system and as the best case execution cycle to worst-case execution cycle ratio (bc/wc) decreases.

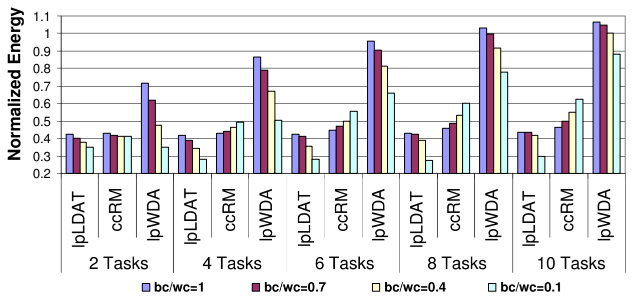


Figure 6. Energy consumption of various algorithms when scheduling randomly generated task sets. The transition time overhead is 100 μs.

5. Summary

In this paper we presented an on-line DVS scheduling algorithm called low power Limited Demand Analysis with Transition overhead (lpLDAT). This algorithm is the first in its class that correctly accounts for time transition overhead to guarantee hard deadlines for real-time systems. Additionally, stochastic information on task execution and transition energy overhead are both considered during the scheduling process, resulting in an overall energy reduction of up to 40% when compared to previous methods.

Although lpLDAT does perform well, it is not optimal. For example, lpLDAT could benefit from a more sophisticated off-line analysis of the task set, especially if more is information is known about each task. Also, the heuristic used to choose between S_{max} and S_{MC} can also be improved. Future work should conduct an in depth analysis of these issues.

References

[1] N. AbouGhazaleh, D. Mossé, B. Childers, R. Melhem, and M. Craven. Collaborative operating system and compiler power management for real-time applications. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 133–141, May 2003.

[2] AMD. Mobile amd athlon 4 processor model 6 cpga data sheet rev:e. Technical Report 24319, Advanced Micro Devices, Nov. 2001.

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

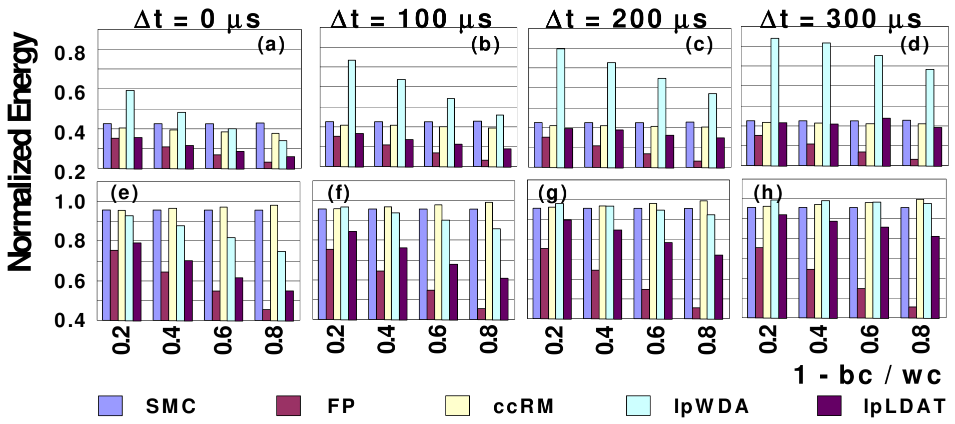


Figure 5. Energy consumption of various algorithms on the CNC (a–d) and Avionics (e–h) task sets.

In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2004.

[4] T. D. Burd. *Energy-Efficient Processor System Design*. PhD thesis, University of California, Berkeley, Berkeley, CA, May 2001.

[5] T. D. Burd and R. W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISPLED)*, pages 9–14, July 2000.

[6] Compaq ipaq h3600 hardware design specification - version 0.2f. online- http: //www.handhelds.org/ Compaq/ iPAQH3600/ iPAQ_H3600.html.

[7] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISPLED)*, pages 46–51, Aug. 2001.

[8] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings of the 19th Real-Time Systems Symposium (RTSS)*, pages 178–187, Dec. 1998.

[9] Intel. The intel xscale microarchitecture. Technical report, Intel Corporation, 2000.

[10] N. Kim, M. Ryu, S. Hong, M. Saksena, C. ho Choi, and H. Shin. Visual assessment of a real-time system design: a case study on a cnc controller. In *Proceedings of the 17th Real-Time Systems Symposium (RTSS)*, pages 300–310, Dec. 1996.

[11] W. Kim, J. Kim, and S. L. Min. Dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using work-demand analysis. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISPLED)*, pages 396–401, Aug. 2003.

[12] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

[13] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in ada: a case study. In *Proceedings of the 12th Real-Time Systems Symposium (RTSS)*, pages 181–189, Dec. 1991.

[14] B. Mochocki, X. S. Hu, and G. Quan. A realistic variable voltage scheduling model for real-time applications. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-Aided design (ICCAD)*, pages 726–731, Nov. 2002.

[15] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP)*, pages 89–102, 2001.

[16] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking (MOBICOM)*, pages 251–259, July 2001.

[17] G. Quan and X. S. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the 2001 Design Automation Conference (DAC)*, pages 828–833, June 2001.

[18] S. Saewong and R. Rajkumar. Practical voltage-scaling for fixed-priority rt systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 106–114, May 2003.

[19] D. Shin, S. Lee, and J. Kim. Intra-task voltage scheduling for low-energy hard real-time applications. *Design & Test of Computers*, 18(2):20–30, March – April 2001.

[20] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on the Foundations of Computer Science (FOCS)*, pages 374–382, Oct. 1995.

[21] Y. Zang, X. S. Hu, and D. Z. Chen. Energy minimization of real-time tasks on viariable voltage processors with transition energy overhead. In *Proceedings of the 2003 Asian and South-Pacific Design Automation Conference (ASPDAC)*, pages 65–70, 2003.

[22] Y. Zhang and K. Chakrabarty. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE)*, page 21170, 2004.