DVSST: A Dynamic Voltage Scaling Algorithm for Sporadic Tasks*

Ala′ Qadi Steve Goddard
Computer Science & Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0115
{*aqadi,goddard}@cse.unl.edu*

Shane Farritor
Mechanical Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0656
{*sfarritor}@unl.edu*

Technical Report TR-CSE-UNL-2003-2

May 2003

Abstract

Dynamic voltage scaling (DVS) algorithms save energy by scaling down the processor frequency when the processor is not fully loaded. Many algorithms have been proposed for periodic and aperiodic task models but none support the canonical sporadic task model. A DVS algorithm, called DVSST, is presented that can be used with sporadic tasks in conjunction with the preemptive EDF scheduling algorithm. The algorithm is proven to guarantee each task meets its deadline while saving the maximum amount of energy possible with processor frequency scaling when tasks execute with their worst-case execution times.

DVSST was implemented in the μC/OS-II real-time operating system for embedded systems and its overhead was measured using a stand-alone Rabbit 2000 test board. Though theoretically optimal, the actual power savings realized with DVSST is a function of the sporadic task set and the processor's DVS support. It is shown that the DVSST algorithm achieves 83% of the theoretical power savings for a Robotic Highway Safety Marker real-time application. The difference between the theoretical power savings and the actual power savings is due to the limited number of frequency levels the Rabbit 2000 processor supports.

1. Introduction

Many embedded real-time systems consist of a battery operated microprocessor system with a limited battery life. Some of these systems use rechargeable batteries (like cellular phones and robots) while others use dry batteries. In both cases it is very important to maximize the battery life. Dynamic

*This work is sponsored, in part, by grants form the National Science Foundation (CCR-0208619) and the National Academy of Sciences Transportation Board—NCHRP IDEA program (Project #90).

Voltage Scaling (DVS) aims at reducing the power consumption of the system by operating the processor at a lower frequency and thus on a lower voltage.

In CMOS circuits the power consumed by a CMOS gate is proportional to the square of the voltage applied to the circuit, as shown by Equation (1) where C_L is the gate load capacitance (output capacitance),V_{DD} is the supply voltage and f is the clock frequency [27]. The circuit delay t_d is given by Equation (2) where k is a constant depending on the output gate size and the output capacitance and V_T is the threshold voltage [27]. The clock frequency is inversely proportional to the circuit delay; it is expressed using t_d and the logic depth of a critical path as in Equation (3) where L_d is the depth of the critical path [27].

$$P_{CMOS} = C_L V_{DD}^2 f \tag{1}$$

$$t_d = k \frac{V_{DD}}{(V_{DD} - V_T)^2} \tag{2}$$

$$f = \frac{1}{L_d \cdot t_d} \tag{3}$$

It is clear form Equation (1) that reducing the supply voltage will reduce the power consumption, however it also reduces the clock frequency, as shown by Equations (2) and (3), which slows down the processor. Thus, the challenge in applying DVS algorithms to real-time systems is to save power while still meeting all temporal requirements of the system.

In recent years significant research has been done in the area of DVS (e.g., [1, 4, 5, 7, 8, 9, 12, 13, 15, 16, 18, 19, 20, 22, 23, 24, 25, 26, 28, 29]). These efforts have resulted in a number of DVS algorithms supporting various task models for embedded and real-time systems. Successful DVS implementations in commercial processors include Intel's Xscale processor [6], Transmeta's Crusoe processor [2] and Rabbit Semiconductors' Rabbit processor [21]. DVS algorithms in [1, 4, 5, 7, 9, 12, 13, 15, 18, 19, 22, 23, 24, 25, 29] support variations of the Liu and Layland periodic task model [14] under Rate Monotonic (RM) scheduling or Earliest Deadline First (EDF) scheduling. Algorithms presented in [15, 16, 20] considered task models that also support aperiodic requests with soft deadlines or non-periodic tasks with hard deadlines in which job release times were known a priori.

However, to date, no DVS algorithms support the sporadic task model defined by Mok [17] in which tasks have a minimal inter-execution time rather than a fixed period. In this work, a DVS algorithm is presented and evaluated that supports the canonical sporadic task model executed under EDF scheduling. Each task in a sporadic task set $\mathrm{T} = \{T_1, T_2, \ldots, T_n\}$ has three associated parameters, *p*, *e*, and *d:*

p is the minimum separation period between the release of two consecutive jobs of a task;

e is the worst-case execution time of the job;

d is the relative deadline for the job.

It is assumed in this work that $d = p$ for all tasks. Thus, each task can be described using the tuple (p,e).

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents our DVS algorithm. Section 4 proves the optimality of the algorithm in terms schedulability and theoretical power savings. Section 5 presents the implementation and evaluation of the algorithm in a stand-alone environment and in an embedded real-time system. We conclude with a discussion of results in Section 6.

2. Related Work

The algorithms in [1, 18, 22, 23] assume the periodic task model and rely on the principles of intra-task DVS. That is, they adjust the processor voltage level, and hence the processor speed, based on the execution path a task takes and commonly rely on compiler support rather than operating system support to conserve power.

The algorithms in [4, 5, 7, 8, 9, 12, 13, 19, 20, 24, 25, 29] also assume the periodic task model, but rely on an alternative approach to intra-task DVS, called inter-task DVS. In general, inter-task DVS algorithms determine the processor voltage on a task-by-task basis. That is, they adjust the supply voltage at a task level such that idle time is removed from the schedule while guaranteeing that all tasks meet their respective deadlines.

The approach used in this work falls into the category of inter-task DVS. Of the periodic inter-task DVS work identified, the Static Voltage Scaling algorithm developed by Pillai and Shin in [19] is the most closely related to this work. Their Static Voltage Scaling algorithm is an offline algorithm that scales the processor voltage by a factor equal to α where α is the minimum utilization required for the task to remain schedulable under EDF or RM scheduling. This technique is also used in this work to remove deterministic idle time from the schedule, as computed using worst-case execution times (WCET) for each task, but in a slightly different way. The other two on-line algorithms in [19], Cycle Preserving and Look Ahead, conserve energy by first using Static Voltage Scaling to set the base processor frequency and then further reduce the voltage level when a job executes for less than its WCET or by deferring task execution as much as possible.

The algorithm presented by Shin and Choi in [24, 25] also sets the initial voltage level using Static Voltage Scaling. They then lower the voltage level further whenever a single task is eligible for execution. Lee *et al*. [13] developed their DVS algorithms using only two voltage levels and distributing the tasks into two sets, each corresponding to one of the voltage levels: High and Low. Their work was based on the results of Ishihara and Yasuura [11] who formulated the processor energy optimization problem as a discrete optimization problem that could be solved using linear integer programming techniques. They showed that, in theory, if the processor has a finite number of discrete voltage levels then it is enough to have at most two voltage levels to minimize the energy consumption of the processor.

Kawaguchi *et al*. [8] presented an approach to schedule a periodic task set by means of task slicing and queues for fixed priority preemptive scheduling, which mainly makes use of the fact that tasks often do not execute with their WCET.

Hong *et al*. [4, 5] proposed a synthesis technique for variable voltage core based systems containing a set of independent, asynchronous periodic tasks with arbitrary start times (phases) that were scheduled with non-preemptive fixed priority scheduling. Zhang and Chanson present three algorithms in [29] that apply DVS to a periodic task model with non-preemptable sections. This work assumes all tasks are independent and fully preemptive.

The algorithms in [15, 16, 20] consider variations of the periodic task model that support aperiodic requests with soft deadlines or non-periodic tasks with hard deadlines in which job release times are known a priori. Lou and Jha [15] presented an algorithm to schedule periodic tasks, soft aperiodic tasks and hard aperiodic tasks with precedence constraints using task graphs, cyclic scheduling and slack steeling. Manzak and Chankrabarti [16] used the method of Lagrange multipliers in an iterative way to determine the minimum optimal voltage at which to execute the task set subject to both time and minimum energy constraints. Their method can be applied to a periodic task set or to what they referred to as Aperiodic Single Tasks, which is a system of tasks where each task releases a job once in a certain period of time called *Ttotal*. They considered both EDF and RM and presented exact and approximate algorithms to find the minimum voltage, with the exact algorithm having higher complexity. Quan *et al*. [20] presented an algorithm that can be applied to a periodic task system or to a type of sporadic task system in which all timing parameters, including task release times, are known a priori.

None of the existing algorithms support the canonical sporadic task model defined by Mok in [17]. The next section presents a DVS algorithm for Mok's sporadic task model with preemptive EDF scheduling, which assumes deadlines are equal to periods.

3. The Algorithm

The Dynamic Voltage Scaling for Sporadic Tasks (DVSST) algorithm presented here is classified as an inter-task DVS algorithm for sporadic tasks. That is, it adjusts the processor voltage on a job-by-job basis, where a job represents the release of a sporadic task. Recall from Equations (2) and (3) that the processor frequency is proportional to the voltage level. As with most DVS algorithms, DVSST is defined in terms of processor frequency, rather than voltage levels, since the relationship between the processor frequency and task execution times can be expressed directly.

The DVSST maintains a frequency-scaling factor, α, that represents the percent of the maximum processor frequency. Rather than using the Static Voltage Scaling algorithm of [19] to set the initial frequency level, DVSST starts with a minimum possible frequency-scaling factor, which can be theoretically zero, and scales the processor frequency up and down depending when jobs are released. The scaling factor α is increased by an amount of e_i/p_i when task T_i is first released. Let r_i be the last

release time of task T_i. DVSST reduces α at time $r_i + p_i$ (or any time after that) by the amount of e_i/p_i if the next job if task T_i was not yet released. When task T_i later releases the job, α is increased by the same amount. The algorithm is explained in detail after we introduce a few definitions.

Definition 1: *The frequency-scaling factor, α, is defined as the ratio between the new processor frequency and the maximum processor frequency:*

$$\alpha = \frac{f_{new}}{f_{\max}}. \tag{4}$$

Corollary 1*:* $\alpha \le 1$.

Proof: The maximum value that we can scale the frequency to is f_{max}. Therefore $\alpha \le \alpha_{\max} = \frac{f_{\max}}{f_{\max}} = 1$. □

Definition 2: *The idle-state scaling factor, α_{idle}, is the minimum scaling factor possible that puts the processor in a sleep mode when there is no job to execute.*

Theoretically α_{idle} = 0, but in many systems α_{idle} must be greater than zero to support platform requirements, or to interact with external devices that trigger the release of a sporadic task. In this section it assumed that α_{idle} = 0. This assumption is relaxed in the next section when we describe the implementation of the algorithm on a real system.

Definition 3: *The delayed release task set, TD, is a subset of the task set* $\mathrm{T} = \{T_1, T_2, \ldots, T_n\}$ *whose last release was at least p_i time units ago. That is, at any time t,* $TD = \{T_i \mid T_i \in \mathrm{T} \wedge (t \ge r_i + p_i)\}$ *where r_i is the last release time of task T_i or -∞ if task T_i has not yet released a job.*

The DVSST is shown in Figure 1. Initially, *TD* = T and the processor frequency is set to α_{idle}. When a task T_i releases a job, the algorithm immediately increases the scaling factor α by an amount equal to $\frac{e_i}{p_i}$ and removes task T_i from the set *TD*. If any task T_i does not release a job at the end of its minimum separation period p_i, the algorithm reduces the scaling factor α by an amount equal to $\frac{e_i}{p_i}$ and task T_i is added to the set *TD*. If the algorithm detects that no job is currently executing, then it sets α to the

```
DVSST( ):
  set α = α_idle and TD = T          // set initial conditions
  while(true) {
    sleep until (∃ T_i : (T_i releases a job and T_i ∈ TD) or
                         (T_i ∉ TD and current_time ≥ r_i + p_i)) or
                         (no task is executing)
    if T_i released a job and T_i ∈ TD then
                              // scale up the processor frequency
        set α = α + e_i/p_i and TD = TD – {T_i }
    else if T_i ∉ TD and current_time ≥ r_i + p_i
                              // scale down the processor frequency
        set α = α – e_i/p_i and TD = TD + {T_i }
    else                      // set processor to idle mode
        set α = α_idle and TD = T
  }
```

Figure 1: The DVSST algorithm.

minimum possible value $\alpha_{idle,}$ or in other words, it sets the processor to the idle or sleep mode.

The value of α may depend on the previous value of α and since α changes with time, we use α_n to represent the n^{th} change to α at time t. Equation (5) shows how, if at all, α_n is changed at time t.

$$\alpha_n = \begin{cases} \alpha_{idle} \ , \ t = 0 \ \text{or} \ \textit{no task is executing} \\ \alpha_{n-1} - \dfrac{e_i}{p_i} \ , \ t \geq r_i + p_i = 0 \ \text{and} \ T_i \notin TD \\ \alpha_{n-1} + \dfrac{e_i}{p_i} \ , \ T_i \ \textit{is released at time } t \ \text{and} \ T_i \in TD \\ \textit{no change otherwise} \end{cases} \tag{5}$$

The following example illustrates how the DVSST algorithm scales the processor frequency (voltage) under EDF scheduling.

Example 1: Let us consider the sporadic task set T_1 = (1,4), T_2 = (1,5), T_3 = (3,10). The (un-scaled) system utilization is U = 0.75. Let us consider scheduling the jobs that were released in the interval [0, 20) under preemptive EDF while using the DVSST algorithm to scale the processor frequency (voltage). Assume that the tasks released jobs as follows: T_1 at times 0, 4, 10, and 17; T_2 at times 0, 6, and 11; and T_3 at times 8 and 18. Let job $J_{i,j}$ represent the j^{th} release of task T_i. Figure 2 illustrates the execution of these jobs without DVSST, and Figure 3 illustrates the same jobs executed with DVSST. The specific job attributes for both executions are listed in Table 1.

Notice that in Figure 2 the processor is idle in the intervals [2,4), [5,6), and [13,17) under EDF scheduling without DVSST. For this set of release times, the DVSST algorithm resulted in an execution in which the processor was never idle during the observed period shown in Figure 3. However, no task missed its deadline—a fact proven in the next section for all feasible task sets.

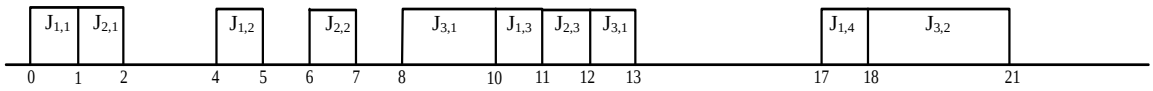

Figure 2: Executing the example task set under EDF without DVSST.

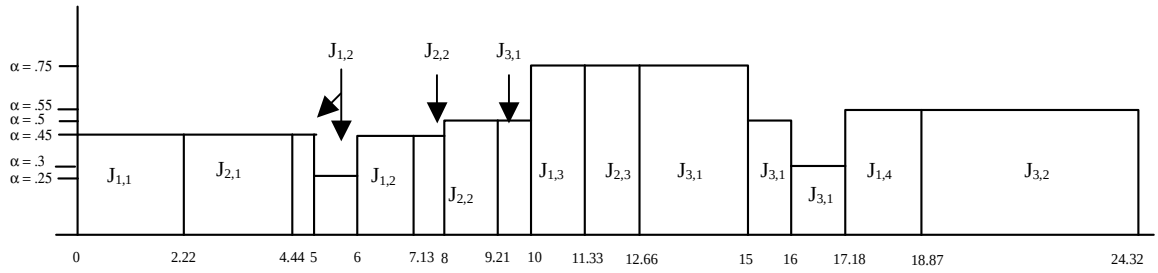

Figure 3: Executing the example task set under EDF with DVSST. The x-axis represents time and the y-axis represents the frequency scaling factor α, which is set at the start of each execution interval.

Job	Release Time	Deadline	EDF without DVSST			EDF with DVSST		
			Execution Interval	α	% of Job Executed	Execution Interval	α	% of Job Executed
$J_{1,1}$	0	4	[0,1)	1	100%	[0,2.22)	.45	100%
$J_{2,1}$	0	5	[1,2)	1	100%	[2.22,4.44)	.45	100%
$J_{1,2}$	4	8	[4,5)	1	100%	[4.44,5)	.45	24.3%
						[5,6)	.25	25%
						[6,7.13)	.45	50.7%
$J_{2,2}$	6	11	[6,7)	1	100%	[7.13,8)	.45	39.33%
				1		[8,9.21)	.5	60.67%
$J_{3,1}$	8	18	[8,10)	1	66.67%	[9.21,10)	.5	13.13%
			[12,13)	1	33.33%	[12.66,15)	.75	58.34%
						[15,16)	.5	16.66%
						[16,17.18)	.3	11.87%
$J_{1,3}$	10	14	[10,11)	1	100%	[10,11.33)	.75	100%
$J_{2,3}$	11	16	[11,12)	1	100%	[11.33,12.66)	.75	100%
$J_{1,4}$	17	21	[17,18)	1	100%	[17.18,18.87)	.55	100%
$J_{3,2}$	18	28	[18,21)	1	100%	[18.87,24.32)	.55	100%

Table 1: Job attributes of the example task set when executed under EDF with and without DVSST. The scaling factor α is set at the start of each execution interval.

4. Theoretical Validation

This section addresses the temporal correctness and energy savings possible when sporadic task sets are executed under EDF with DVSST. Section 4.1 presents the temporal correctness and optimality of EDF with DVSST. Section 4.2 quantifies the power savings possible when both the processor voltage and frequency can be scaled, as well as when only the processor frequency can be scaled. It is shown that DVSST is optimal with respect to power savings when only the frequency can be scaled and all tasks execute with their WCET. While this may seem like a strict constraint on the processor and task set, Section 5 presents an implementation on a Rabbit 2000 processor for which these constraints hold.

4.1 Temporal Correctness

A voltage (frequency) scaling scheduling algorithm for real-time systems is correct if it guarantees that all jobs meet their deadlines under a specified scheduling algorithm. Example 1, in Section 3 demonstrated that scaling the processor frequency results in new task execution times that are proportional to the frequency-scaling factor. Theorem 1 states that under DVSST, these scaled task execution times results in a scaled processor utilization of one if the sporadic tasks execute at their maximum rate. This theorem provides an intuitive understanding of Theorem 4, which states that (un-scaled) processor utilization less than or equal to one is a necessary and sufficient feasibility condition for sporadic task sets. Before presenting these theorems, however, new definitions are required.

Definition 4: *Scaled-Mode Execution Time, e_s, is the execution time needed to execute a job under a frequency-scaling factor.*

Over any time interval where the scaling factor α is constant then e_s can be calculated by Equation (6) where e_{si} is the scaled execution time of task T_i, e_i is the normal execution time of T_i, and α_c is the current scaling factor.

$$e_{si} = \frac{e_i}{\alpha_c} \tag{6}$$

Definition 5: *Scaled Mode Utilization, U_s, is the processor utilization while executing at a scaled frequency.*

U_s can be calculated over any time interval τ by Equation (7) where α is constant over τ.

$$U_{s\tau} = \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_{si}}{p_i} \tag{7}$$

Definition 6*: Scaling Factor Change Interval, $\tau_{Si,}$ is the time interval between two consecutive scaling factor changes α_i and α_{i+1}.*

Theorem 1: *If $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$ and the DVSST algorithm is used to scale the processor frequency then the processor scaled-mode utilization is always equal to 1 if the tasks are released at their maximum rate.*

Proof: Let τ_S be the set of all scaling-factor change intervals between the end of any idle interval and the beginning of the next idle interval where $\tau_S = \{\tau_{S0}, \tau_{S1}, \ldots, \tau_{Sm}\}$. If there are no idle intervals, then let τ_S be equal to the length of the hyperperiod of the task set and begin at a hyperperiod boundary. Let $\alpha_S = \{\alpha_{S1}, \alpha_{S2}, \ldots, \alpha_{Sm}\}$ be the set of all the scaling factors corresponding to the time intervals in τ_S. From Equation (7), the scaled utilization over any scaling-factor change interval τ_{Sj} is $U_{s\tau_j} = \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_{si}}{p_i}$, with $e_{si} = \frac{e_i}{\alpha_{s_j}}$ by Equation (7). Substituting for e_{si} we have $U_{s\tau_j} = \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_{si}}{p_i} = \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_i}{p_i \cdot \alpha_{s_j}} = \frac{1}{\alpha_{s_j}} \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_i}{p_i}$ but $\alpha_{s_j} = \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_i}{p_i}$ since *TD* contains all the tasks that did not release a job at its minimum time separation before the start of the interval τ_{Sj}. Substituting for α_{sj} we have $U_{s\tau_j} = \frac{1}{\sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_i}{p_i}} \sum_{\substack{i=1 \\ T_i \notin TD}}^{n} \frac{e_i}{p_i} = 1$.

The scaled-mode utilization over the whole time interval τ_S can be calculated as a sum of the products of the utilizations over subintervals times the ratio of the interval to the sum of all intervals, as expressed in the following equation.

$$U_{S\tau} = \sum_{j=1}^{m} \frac{\tau_{sj}}{\sum_{j=1}^{m} \tau_{sj}} \cdot U_{S\tau_j}$$

$$= \frac{1}{\sum_{j=1}^{m} \tau_{sj}} \cdot \sum_{j=1}^{m} \tau_{sj} \cdot U_{S\tau_j} \text{ since } \sum_{j=1}^{m} \tau_{sj} \text{is a constant with respect to the outer summation}$$

$$= \frac{1}{\sum_{j=1}^{m} \tau_{sj}} \cdot \sum_{j=1}^{m} \tau_{sj} \quad \text{since } U_{S\tau_j} = 1$$

$$= 1$$

□

From Theorem 1, one might suspect that U ≤ 1 is a necessary and sufficient feasibility condition for preemptive EDF scheduling with the DVSST algorithm. We will prove this after we introduce a few definitions and concepts.

Definition 7: *Job Inter-Release Time is the time interval between the release of any job of task T_i and the release of the next job of the same task. That is the time interval between the release of job $J_{i,j}$ and the release of job $J_{i,j+1}$.*

We use the notation $\delta_{i,j}$ to denote the inter-release time between jobs $J_{i,j}$ and $J_{i,j+1}$ where $r_{i,j}$ is the release time of job $J_{i,j}$:

$$\delta_{i,j} = r_{i,j+1} - r_{i,j} \tag{8}$$

Corollary 2: $\delta_{i,j} \geq p_i$

This corollary follows directly from the definition of the sporadic task in which tasks must have a minimum separation *p* between the releases of jobs. Therefore the job inter-release time cannot be less than p_i.

Definition 8: *Unreleased Jobs are all the jobs that could have been released in a time interval τ at their corresponding minimum separation periods but were not released at that time.*

Theorem 2: *Let $T = \{T_1, T_2 ..., T_n\}$ be a set of sporadic tasks with deadlines equal to their periods. Let τ be an interval in time and $J_R = \{J_{1,1}, J_{1,2} ... J_{1,m_1}, ..., J_{n,1}, J_{n,2} ... J_{n,m_n}\}$ be the set of all the jobs released in τ, where $J_{i,1}$ denotes the first job that was released by task T_i in τ. Let $\Delta_R = \{\delta_{1,0}, \delta_{1,1}, ..., \delta_{1,m_1}, ..., \delta_{n,0}, \delta_{n,1}, ..., \delta_{n,m_n}\}$ be the set of the corresponding job inter-release time intervals corresponding to the jobs in J_R except for $\delta_{i,0}$ and δ_{i,m_i}. Define $\delta_{i,0}$ for each job to be the time*

interval between the beginning of the time interval τ *and the release of job* $J_{i,1}$. *Define* δ_{i,m_i} *to be the interval between the release job* $J_{i.m_i}$ *and the end of the interval* τ. *The number of unreleased jobs N in* τ *is given by*

$$N = \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \tag{9}$$

Proof: We know that

$$\delta_{i,j} = r_{i,j+1} - r_{i,j}$$

The number of unreleased jobs in interval $(r_{i,j},r_{i,j+1})$ is the same number of jobs that would have been released by task T_i if T_i was a periodic task. If T_i was periodic and $\delta_{i,j}$ was a multiple of p_i then the number of jobs that would have been released is $\frac{\delta_{i,j}}{p_i} - 1$, we have the -1 here because we are considering the open interval $(r_{i,j},r_{i,j+1})$ and because we already know that a job has been released at $r_{i,j+1}$; It is clear that we need to exclude that job because it was released when it was expected to be released.

In order to extend this to the general case where $\delta_{i,j}$ is not a multiple of p_i we add a ceiling function to $\frac{\delta_{i,j}}{p_i}$ to extend $r_{i,j+1}$ to the next multiple of p_i and exclude the job that would be released at the end of the interval. Hence we have the following expression for the jobs that where not released by task T_i in $\delta_{i,j}$.

$$Number\,Of\ \ Unreleased\,Jobs\,By\ T_i\ \ in\ \delta_{i,j} = \left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \tag{10}$$

To find the number of all unreleased jobs in τ we sum Equation (10) over all $\delta_{i,j}$ in Δ_R, therefore we have

$$N = \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right)$$

□

The following two examples help to clarify the theorem

Example 2: Let $T_1 = (1,.2)$ with $J_{1,1}$ released at $t = 0$ and $J_{1,2}$ released at $t = 3.2$. If T_1 was periodic then it would have released 3 jobs in $(0,3.2)$, if we use Equation (9) we get the same answer

$$N = \left\lceil \frac{3.2}{1} \right\rceil - 1 = 4 - 1 = 3$$

Example 3: Let $T_1 = (1,.2)$ with $J_{1,1}$ released at $t = 0$ and $J_{1,2}$ released at $t = 3$. If T_1 was periodic then it would have released 2 jobs in $(0,3)$, if we use Equation (9) we get the same answer

$$N = \left\lceil \frac{3}{1} \right\rceil - 1 = 3 - 1 = 2$$

Now we will introduce the definition of *Processor Time Capacity*, because if we are scaling the processor frequency then there are two ways to look at the system: we can either look at the system running in real time and the execution times of jobs are scaled, or as though the execution times are the same but time itself is scaled.

Definition 9: *Processor Time Capacity ρ is the amount of scaled time available in any real time interval when the processor is running in scaled mode.*

Over an any time interval $[t_1,t_2)$ where α is constant, the processor time capacity ρ, or the scaled time, is the product of the length of the interval times the scaling factor as given in Equation (11).

$$\rho = \alpha(t_2 - t_1) \tag{11}$$

Now let τ be an interval of time. Let τ_S be the set of all Scaling Factor Change Intervals where $\tau_S = \{\tau_{S0}, \tau_{S1}, ..., \tau_{Sm}\}$ and let $\alpha_S = \{\alpha_{S1}, \alpha_{S2}, ..., \alpha_{Sm}\}$ be the set of all the scaling factors corresponding to the time intervals in τ_S. Then the processor time capacity ρ over τ is given by

$$\rho = \sum_{i=1}^{m} \alpha_{s_i} \cdot \tau_{s_i} \tag{12}$$

Another way to evaluate the value of ρ is to look at it as the sum of all products of the positive changes in α times the change interval.

$$\rho = \sum Postive\,Change\,in\,\alpha \times Time\,Interval\,For\,The\,Change \tag{13}$$

Example 4 shows how Equations (12) and (13) give the same answer for calculating the processor time capacity.

Example 4: Let us consider the scaling factor α_i plotted against time *t* in Figure 4 for a task set, where $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ are the scaling factor change intervals, $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ are the scaling factors corresponding to $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$. Let $\{\Delta\alpha_1, \Delta\alpha_2, \Delta\alpha_3, \Delta\alpha_4\}$ be the positive changes in α and $\{\bar{\delta}_1, \bar{\delta}_2, \bar{\delta}_3, \bar{\delta}_4\}$ be the time intervals corresponding to $\{\Delta\alpha_1, \Delta\alpha_2, \Delta\alpha_3, \Delta\alpha_4\}$.

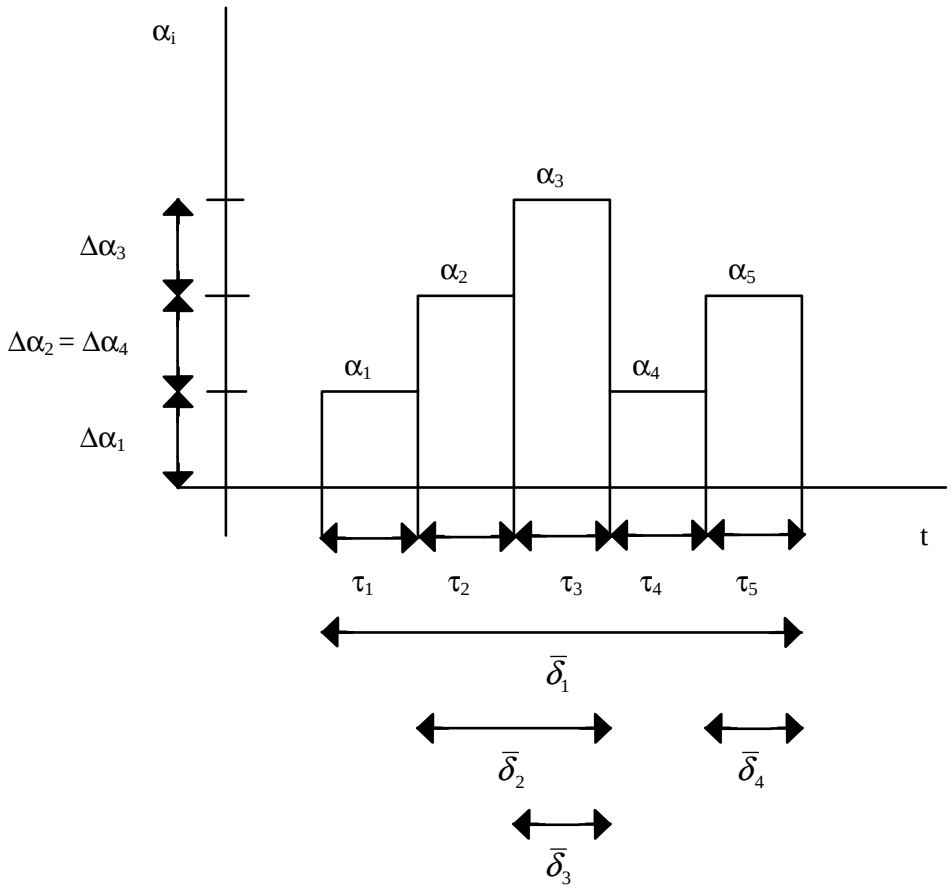

Figure 4: α_i for a task set.

From Figure 4, we can derive the following equations.

$$\alpha_1 = \Delta\alpha_1$$
$$\alpha_2 = \Delta\alpha_1 + \Delta\alpha_2$$
$$\alpha_3 = \Delta\alpha_1 + \Delta\alpha_2 + \Delta\alpha_3$$
$$\alpha_4 = \Delta\alpha_1$$
$$\alpha_5 = \Delta\alpha_1 + \Delta\alpha_4$$

$$\bar{\delta}_1 = \tau_1 + \tau_2 + \tau_3 + \tau_4 + \tau_5$$
$$\bar{\delta}_2 = \tau_2 + \tau_3$$
$$\bar{\delta}_3 = \tau_3$$
$$\bar{\delta}_4 = \tau_5$$

We not that $\Delta\alpha_2 = \Delta\alpha_4$ because both of these changes are the result of releasing different jobs from the same task. Using the previous equations we can calculate the processor time capacity

$$\begin{aligned}
\rho &= \sum_{i=1}^{5} \alpha_i \cdot \tau_i \\
&= \alpha_1 \cdot \tau_1 + \alpha_2 \cdot \tau_2 + \alpha_3 \cdot \tau_3 + \alpha_4 \cdot \tau_4 + \alpha_5 \cdot \tau_5 \\
&= \Delta\alpha_1 \cdot \tau_1 + (\Delta\alpha_1 + \Delta\alpha_2) \cdot \tau_2 + (\Delta\alpha_1 + \Delta\alpha_2 + \Delta\alpha_3) \cdot \tau_3 + \Delta\alpha_1 \cdot \tau_4 + (\Delta\alpha_1 + \Delta\alpha_4) \cdot \tau_5 \\
&= \Delta\alpha_1 \cdot (\tau_1 + \tau_2 + \tau_3 + \tau_4 + \tau_5) + \Delta\alpha_2 \cdot (\tau_2 + \tau_3) + \Delta\alpha_3 \cdot \tau_3 + \Delta\alpha_4 \cdot \tau_5 \\
&= \Delta\alpha_1 \cdot \bar{\delta}_1 + \Delta\alpha_2 \cdot \bar{\delta}_2 + \Delta\alpha_3 \cdot \bar{\delta}_3 + \Delta\alpha_4 \cdot \bar{\delta}_4 \\
&= \sum Change\ in\ \alpha \times Time\ Interval\ For The\ Change
\end{aligned}$$

Definition 10: *Let $J_{i,j}$ be a job released in any time interval with an execution time of e_i and a minimum separation period p_i. Then the Job Scaling Factor Active Interval is the time interval starting at the instant when the processor frequency was scaled up by a factor equal to $\frac{e_i}{p_i}$ until the instant where the frequency is scaled down by a factor of $\frac{e_i}{p_i}$.*

Theorem 3: *The Job Scaling Factor Active Interval $\bar{\delta}$ is a multiple of the minimum separation period p_i of the task that released the job.*

Proof:

The frequency is scaled up by a factor of $\frac{e_i}{p_i}$ only if $J_{i,j}$ was released after it has not been released at least once at its minimum separation period. The frequency would be decreased by $\frac{e_i}{p_i}$ only at the end of the minimum separation period if the job $J_{i,j}$ was not released. Let us assume that job $J_{i,j}$ was not released prior to t_1, then it was released at time t_1. Assume that T_i has released k jobs at their minimum separation consecutively where job $J_{i,j+k}$ was released at time t_2. Let t_3 be the first moment after t_1 for T_i not to release a job when it was expected to be released, as shown in Figure 5. The length of the Job Scaling Factor Active Interval is equal to $t_3 - t_1$, because we will increase α by $\frac{e_i}{p_i}$ at t_1 and not decrease it until t_3. T_i has released all its jobs in $[t_1, t_2)$ at their minimum separation periods. Therefore $[t_1, t_2)$ is a multiple of p_i. It is also clear that t_3 is at the end of a minimum separation period because we reduced the frequency at t_3, therefore $(t_3 - t_2)$ must be equal to p_i. Hence the total period $[t_1, t_3)$ is a multiple of p_i. □

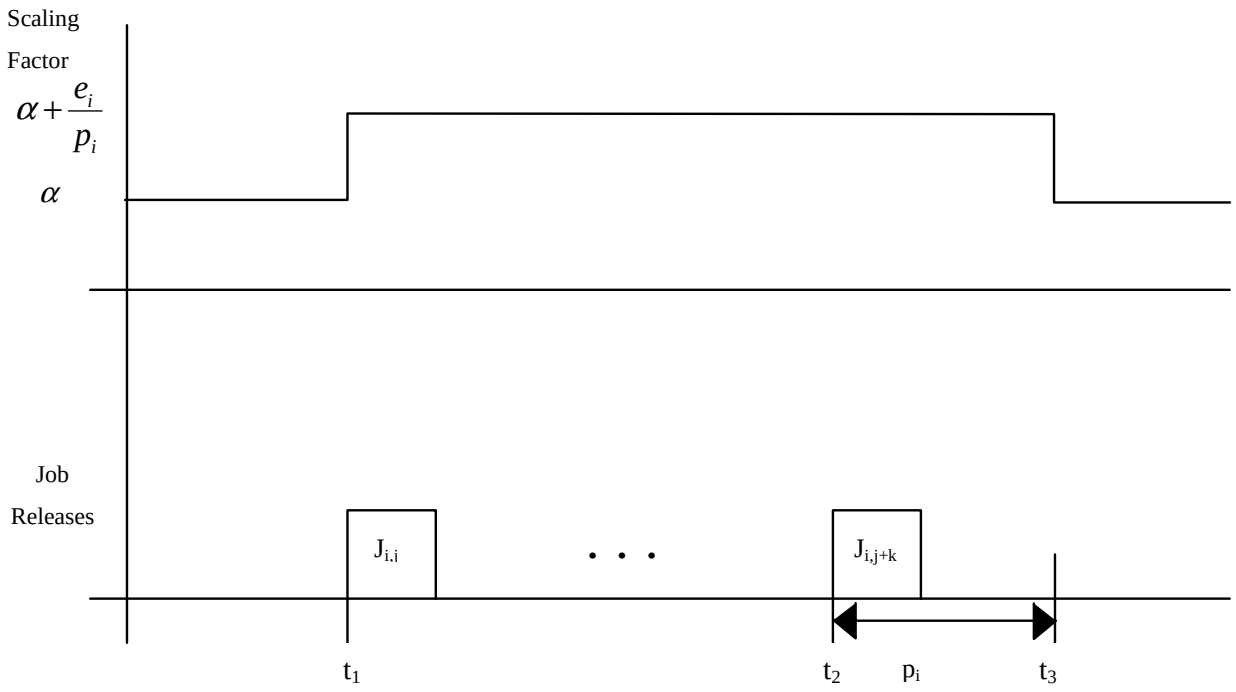

Figure 5: Illustration of scaling factor active interval.

We now introduce a way of calculating the processor time capacity.

Let $J_R = \{J_{1,1}, J_{1,2}...J_{1,m_1},..., J_{n,1}, J_{n,2}...J_{n,m_n}\}$ be the set of all the jobs released in an interval τ, where $J_{i,1}$ denotes the first job that was released by task T_i in τ. Let $\Delta_R = \{\delta_{1,0}, \delta_{1,1},..., \delta_{1,m_1}..., \delta_{n,0}, \delta_{n,1},..., \delta_{n,m_n}\}$ be the set of the corresponding job inter-release time intervals corresponding to the jobs in J_R except for $\delta_{i,0}$ and δ_{i,m_i}. Define $\delta_{i,0}$ for each job to be the interval between the beginning of the time interval τ and the release of job $J_{i,1}$. Define δ_{i,m_i} to be the interval between the release job $J_{i,n}$ and the end of the interval τ. Let $\overline{\Delta_R} = \{\overline{\delta_{1,1}}, \overline{\delta_{1,2}},....\overline{\delta_{1,k_1-1}},..., \overline{\delta_{n,1}}, \overline{\delta_{n,2}},...\overline{\delta_{n,k_n-1}}\}$ be the set of all job scaling factor active intervals in τ. Let $\alpha_R = \{\alpha_{R1,1}, \alpha_{R1,2}...\alpha_{R1,k_1-1}..., \alpha_{Rn,1}, \alpha_{Rn,2}...\alpha_{Rn,k_n-1}\}$ be the scaling factors corresponding to the intervals in $\overline{\Delta_R}$ where $\alpha_{Ri,j}$ is defined as the scaling factor resulting from the fact that only job $J_{i,j}$ was released.

$$\alpha_{Ri,j} = \frac{e_i}{p_i} \tag{14}$$

Now we can calculate ρ using these new parameters as follows.

$$\rho = \sum Postive\ Change\ in\ \alpha \times Time\ Interval\ For\ The\ Change$$

We know that over a job scaling factor active interval $\overline{\delta}_{i,j}$, a change of $\alpha_{Ri,j}$ occurred over $\overline{\delta}_{i,j}$ regardless of any other change in α. Because this change is independent of all other changes we can sum all these changes over all $\overline{\delta}_{i,j}$, substituting these parameters in Equation (13) we get Equation (15).

$$\rho = \sum_{i=1}^{n} \sum_{j=1}^{k_{i-1}} \overline{\delta_{i,j}} \cdot \alpha_{R_{i,j}} \tag{15}$$

From Equation (14) we have $\alpha_{Ri,j} = \frac{e_i}{p_i}$ therefore

$$\rho = \sum_{i=1}^{n} \sum_{j=1}^{k_{i-1}} \overline{\delta_{i,j}} \cdot \frac{e_i}{p_i}$$

$$= \sum_{i=1}^{n} \left(\frac{e_i}{p_i} \cdot \sum_{j=1}^{k_{i-1}} \overline{\delta_{i,j}} \right)$$

From Theorem 3 we know that $\overline{\delta}_{i,j}$ is always a multiple of p_i. Therefore we can express $\overline{\delta}_{i,j}$ as

$$\overline{\delta_{i,j}} = M_{i,j} \cdot p_i \tag{16}$$

where $M_{i,j}$ is the number of jobs released in $\overline{\delta}_{i,j}$. If we let M_i represent the number of all the released jobs of task T_i in τ then we have

$$\sum_{j=1}^{k_{j-1}} \overline{\delta_{i,j}} = M_i \cdot p_i \tag{17}$$

Now we can substitute Equation (17) in Equation (15) to calculate the value of ρ

$$\rho = \sum_{i=1}^{n} \left(\frac{e_i}{p_i} \cdot M_i \cdot p_i \right)$$

$$\rho = \sum_{i=1}^{n} M_i \cdot e_i \tag{18}$$

M_i can be calculated as

M_i = Maximum Number Of Jobs That Can Be Released in τ By T_i – Number Of Unreleased Jobs of T_i in τ.

The maximum number of jobs that can be released in τ is $\left\lceil \frac{\tau}{p_i} \right\rceil$. The number of unreleased jobs by T_i in τ can be calculated using Equation (9). Therefore we have

$$M_i = \left\lceil \frac{\tau}{p_i} \right\rceil - \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right)$$

Substituting this value in Equation (15) we get

$$\rho = \sum_{i=1}^{n} \left[\left\lceil \frac{\tau}{p_i} \right\rceil - \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \right] \cdot e_i$$
$$= \sum_{i=1}^{n} \left\lceil \frac{\tau}{p_i} \right\rceil \cdot e_i - \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \cdot e_i \qquad (19)$$

Now we can proceed to the proof of correctness of the DVSST algorithm. We will state the correctness Theorem and prove it.

Theorem 4: *Let* $\mathrm{T} = \{T_1, T_2, \ldots, T_n\}$ *be a sporadic task set with* $d_i = p_i$*. Preemptive EDF with DVSST will succeed in scheduling* τ *if and only if* $\sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$.

Proof: Establishing the contrapositive shows necessity, i.e. a negative result from the equation implies that *T* is not feasible. Let us assume a negative result for the equation, that is, $U > 1$. But we know that if $U > 1$ then EDF will not find a feasible schedule, therefore DVSST combined with EDF will not find a feasible schedule.

To show the sufficiency of the theorem, we assume that $U \leq 1$ and DVSST and EDF are used to schedule *T* yet there is a job that misses its deadline.

Let job J_d be the first job to miss its deadline at time t_d. Let t_0 denote the last processor idle instant. We note that at the worst case, we will at least have an idle instant at t = 0. Let τ be the time interval $[t_0,t_d)$ and let $J_R = \{J_{1,1}, J_{1,2} \ldots J_{1,m_1}, \ldots, J_{n,1}, J_{n,2} \ldots J_{n,m_n}\}$ be the set of all the jobs released in τ, where $J_{i,1}$ denotes the first job that was released by task T_i in $[t_0,t_d)$. Let $\Delta_R = \{\delta_{1,0}, \delta_{1,1}, \ldots, \delta_{1,m_1} \ldots, \delta_{n,0}, \delta_{n,1}, \ldots, \delta_{n,m_n}\}$ be the set of the corresponding job inter-release time intervals corresponding to the jobs in J_R except for $\delta_{i,0}$ and δ_{i,m_i}. Define $\delta_{i,0}$ for each job to be the time interval between the beginning of the time interval τ and the release of job $J_{i,1}$. Define δ_{i,m_i} to be the interval between the release of job $J_{i.m_i}$ and the end of the interval τ. If job J_d missed its deadline at t_d then the demand in $[t_0,t_d)$ must have been greater than the processor time capacity in $[t_0,t_d)$. That is

$$processer\ time\ capacity < demand \qquad (20)$$

From Equation (19) we have

$$processer\ time\ capcity = \sum_{i=1}^{n} \left\lceil \frac{t_d - t_0}{p_i} \right\rceil \cdot e_i - \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \cdot e_i \qquad (21)$$

The demand can be calculated as

$$demand = \left(\sum_{i=1}^{n} Maximum\ possible\ number\ of\ \ jobs\ released\ by\ T_i in [t_0, t_d) with\ deadlines \le t_d \times e_i \right) - \left(\sum_{i=1}^{n} number\ of\ \ unreleased\ \ jobs\ by\ T_i in [t_0, t_d) \times e_i \right)$$

Equation (9) gives the number of unreleased jobs by each task. Substituting Equation (9) in the above equation we have

$$demand = \sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor \cdot e_i - \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \cdot e_i \tag{22}$$

Substituting Equations (21) and (22) in (20) we have

$$\sum_{i=1}^{n} \left\lceil \frac{t_d - t_0}{p_i} \right\rceil \cdot e_i - \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \cdot e_i < \sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor \cdot e_i - \sum_{i=1}^{n} \sum_{j=0}^{m_i} \left(\left\lceil \frac{\delta_{i,j}}{p_i} \right\rceil - 1 \right) \cdot e_i$$

$$\sum_{i=1}^{n} \left\lceil \frac{t_d - t_0}{p_i} \right\rceil \cdot e_i < \sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor \cdot e_i$$

$$0 < \sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor \cdot e - \sum_{i=1}^{n} \left\lceil \frac{t_d - t_0}{p_i} \right\rceil \cdot e_i$$

$$0 < \sum_{i=1}^{n} \left(\left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor - \left\lceil \frac{t_d - t_0}{p_i} \right\rceil \right) \cdot e_i \tag{23}$$

We know that e_i is always positive, and $\frac{t_d - t_0}{p_i}$ is always positive. From basic mathematics we know that $\forall x \in \Re, x > 0, \lfloor x \rfloor - \lceil x \rceil \le 0$. Therefore $\sum_{i=1}^{n} \left(\left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor - \left\lceil \frac{t_d - t_0}{p_i} \right\rceil \right) \cdot e_i$ must be less or equal to zero, which contradicts Equation (23). This means that under DVSST no task will miss its deadline because demand cannot be greater than processor time capacity if the task set is feasible under EDF. □

4.2 Power Savings

The amount of power that can be saved depends on whether both frequency and voltage are scaled or frequency alone is scaled. Some processors, such as the Crusoe processor [2], have a feed back loop to scale voltage when the frequency is scaled. Other processors, such as the Rabbit processor [21], can operate on multiple voltage levels but cannot scale the voltage with frequency changes.

Equation (1) shows that power is linearly proportional to the frequency and quadratically proportional to the voltage. If the processor automatically scales the voltage when the frequency is scaled, then there will be a voltage level corresponding to each frequency level. Let α be the frequency-scaling factor and β be the voltage-scaling factor corresponding to α. From Equation (2) it is clear that the frequency and voltage are related, but the relation between α and β depends on the gate threshold voltage V_T and the voltage itself, V. Equation (24) shows the relation between α and β.

$$\alpha = \frac{(\beta V - V_T)^2}{\beta (V - V_T)^2} \tag{24}$$

Let us compute the power savings of the DVSST algorithm in both cases. First consider the case where only the frequency is scaled, which is the case for the Rabbit processor used in the application described in Section 5. Over any interval in time τ, the normalized power savings will be given by

$$Power\ Savings = \frac{P_{\max} - P_{DVSST}}{P_{\max}} \tag{25}$$

where P_{max} is the average power consumed by the processor operating at frequency f_{max} and P_{DVSST} is the average power consumed by the processor operating under the DVSST algorithm. Let τ_S be the set of all scaling-factor change intervals in τ where $\tau_S = \{\tau_0, \tau_1, \ldots, \tau_n\}$. Let $\alpha_S = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ be the set of all scaling factors corresponding to the intervals in τ_S. The power savings can be calculated as follows where P_i is the average power consumed during interval τ_i.

$$\begin{aligned}
Power\ Savings &= \frac{P_{\max} - \frac{1}{\tau}\sum_{i=0}^{n} P_i \cdot \tau_i}{P_{\max}} = \frac{Cf_{\max}V_{\max}^2 - \frac{1}{\tau}\sum_{i=0}^{n} Cf_i V_{\max}^2 \cdot \tau_i}{Cf_{\max}V_{\max}^2} \quad \text{but } f_i = \alpha_i f_{\max} \\
&= \frac{Cf_{\max}V_{\max}^2 - \frac{1}{\tau}\sum_{i=0}^{n} C\alpha_i f_{\max}V_{\max}^2 \cdot \tau_i}{Cf_{\max}V_{\max}^2} = \frac{Cf_{\max}V_{\max}^2 - \frac{Cf_{\max}V_{\max}^2}{\tau}\sum_{i=0}^{n} \alpha_i \cdot \tau_i}{Cf_{\max}V_{\max}^2} \\
&= 1 - \frac{1}{\tau}\sum_{i=0}^{n} \alpha_i \cdot \tau_i \\
&= 1 - U_{s\tau}
\end{aligned}$$

Now consider the case where both frequency and voltage are scaled. In the case let us keep all the previous assumptions but add $\beta_S = \{\beta_1, \beta_2, \ldots, \beta_n\}$ where β_S is the set of voltage-scaling factors corresponding to the intervals in τ_S. The normalized power savings is then computed as follows.

$$Power\ Savings = \frac{P_{\max} - \frac{1}{\tau}\sum_{i=0}^{n} P_i \cdot \tau_i}{P_{\max}} = \frac{Cf_{\max}V_{\max}^2 - \frac{1}{\tau}\sum_{i=0}^{n} Cf_i V_i^2 \cdot \tau_i}{Cf_{\max}V_{\max}^2} \quad \text{but } f_i = \alpha_i f_{\max} \text{ and } V_i = \beta_i V_{\max}$$

$$= \frac{Cf_{\max}V_{\max}^2 - \frac{1}{\tau}\sum_{i=0}^{n} C\alpha_i\beta_i^2 f_{\max}V_{\max}^2 \cdot \tau_i}{Cf_{\max}V_{\max}^2}$$

$$= \frac{Cf_{\max}V_{\max}^2 - \frac{Cf_{\max}V_{\max}^2}{\tau}\sum_{i=0}^{n} \alpha_i\beta_i^2\tau_i}{Cf_{\max}V_{\max}^2} = 1 - \frac{1}{\tau}\sum_{i=0}^{n} \alpha_i\beta_i^2\tau_i$$

We note that in this case the power savings is not equal to $1-U_{s_\tau}$ because of the voltage-scaling factor β_i. However, the maximum power that can be saved is still achieved by operating the processor at a frequency equal to the processor utilization.

Theorem 5: *If only the frequency can be scaled and the task set is feasibly scheduled, then the processor will save the maximum possible amount of power under DVSST when all tasks execute with the their WCET.*

Proof: If the processor only scales the frequency, then the minimum average power consumed in any feasible time interval occurs when the processor is run at a frequency equal to the system utilization over that time interval, assuming WCET is realized. Equation (26) shows the power consumed in this case.

$$P = C \cdot f \cdot V^2 \quad \text{but} \quad f = \frac{1}{\tau}\sum_{i=1}^{n} \alpha_i \cdot \tau_i \cdot f_{\max}$$

$$P = \frac{C \cdot V^2 \cdot f_{\max}}{\tau}\sum_{i=1}^{n} \alpha_i \cdot \tau_i \tag{26}$$

If the processor is running the DVSST algorithm then average power consumed can be computed using Equation (27).

$$P = \frac{1}{\tau}\sum_{i=1}^{n} P_i \cdot \tau_i$$

$$P = \frac{1}{\tau}\sum_{i=1}^{n} C \cdot V^2 \cdot f_i \cdot \tau_i \qquad \text{but} \quad f_i = \alpha_i \cdot f_{\max}$$

$$P = \frac{1}{\tau}\sum_{i=1}^{n} C \cdot V^2 \cdot \alpha_i \cdot f_{\max} \cdot \tau$$

$$P = \frac{C \cdot V^2 \cdot f_{\max}}{\tau}\sum_{i=1}^{n} \alpha_i \cdot \tau_i \tag{27}$$

Equations (26) and (27) are equal. This proves that if the processor runs the DVSST algorithm, it will consume the same amount of power as if it was running on a single frequency equal to the task utilization over the whole time interval τ. Thus, DVSST achieves maximum power savings when only the processor frequency can be dynamically scaled. □

Theorem 4 states that $U \leq 1$ is a necessary and sufficient condition for schedulability under EDF with DVSST. Thus, DVSST does not affect the optimality of EDF scheduling for sporadic task sets. Theorem 5 shows that, in theory, DVSST is optimal with respect to power savings when only the frequency can be scaled and all tasks execute with their WCET. However, in practice, it is much harder to achieve optimal power savings due to algorithm overhead and limited frequency levels supported by many processors. The next section discusses these implementation issues.

5. Implementation and Evaluation

The DVSST algorithm was implemented in a modified version of Jean Labrosse's μC/OS-II (MicroC/OS-II) real time operating system [10]. The original version of μC/OS-II uses the RM algorithm to preemptively schedule up to 64 tasks. The modified version used in this study also supports EDF scheduling of up to 64K tasks [11]. Algorithm overhead was measured using a stand-alone Rabbit 2000 test board [21]. The actual power savings realized with DVSST is a function of the sporadic task set and the processor. Rather than create random task sets, we measured the power savings produced by a specific application, the Robotic Highway Safety Marker.

Section 5.1 describes frequency scaling in the Rabbit 2000. Section 5.2 presents slight modifications to the DVSST algorithm required in practice since currently available embedded processors have a limited number of frequency scaling levels. The overhead created by DVSST under EDF scheduling on the Rabbit 2000 is reported in Section 5.3. Section 5.4 describes the Robotic Highway Safety Marker and power savings realized for that application.

5.1 Frequency Scaling in the Rabbit 2000

There are two crystal oscillators built into the Rabbit 2000. The main oscillator accepts crystals up to a frequency of 29.4912 MHz and is used to derive the clock for the processor and peripherals. The low power clock oscillator requires a 32.768 kHz crystal, and is used to clock the watchdog timer, a battery backed time/date clock, and a periodic interrupt. The main oscillator can be shut down in a special low-power mode of operation, and the 32.768 kHz oscillator is then used to clock all the things normally clocked by the main oscillator.

The main oscillator can be doubled in frequency and/or divided by 8. If both doubling and dividing are enabled, then there will be a net frequency division by 4. Our model of the Rabbit 2000 has an 18.532 MHz main oscillator. Thus, there are four frequency levels available from the main oscillator: 18.532MHz, 9.266MHz, 4.633MHz and 2.3165MHz— which correspond to 100%, 50%, 25% and 12.5% of the maximum frequency. Since the maximum frequency at which we can operate the processor is 18.532 MHz and the low power mode frequency is 32.768 kHz, the idle-state scaling factor used by

DVSST is $\alpha_{idle} = \frac{32.768kHz}{18.532MHz} = .00176$. In practice, the value of α_{idle} can be close to zero but never zero as assumed in the theoretical presentation of DVSST.

The Rabbit 2000 processor can operate at different voltages but it does not change the voltage level dynamically when the frequency level is changed. Thus, only the processor frequency will be scaled dynamically, which will result in a linear savings in average power as explained in Section 4.2.

5.2 Modifying DVSST for the Rabbit Processor

There are four non-idle scaling levels available on the Rabbit 2000, rather than the infinite number of levels often assumed in theory. Fortunately, the algorithm can be modified slightly to allow scaling the frequency to a discrete number of levels by rounding the value of α to the next upper scaling level. For example, if we have a processor with scaling levels 0.25, 0.5, 0.75, and 1.0 and the value of α_n at some point in time *t* as calculated by DVSST is 0.58, then the next upper scaling level to which we set α_n is 0.75.

Another challenge in implementing DVSST on the Rabbit 2000 is that serial communication baud rates cannot be derived from the low-power oscillator. Thus, α_{idle} = 0.00176 cannot be used with any application that requires serial communication. Since the wireless transceiver used in the Robotic Highway Safety Marker uses a serial interface to the processor, we use $\alpha_{idle} = \alpha_{\min} = 0.125$ so that the application will not lose communication with the other robots.

5.3 Algorithm Overhead

There are two primary sources of overhead created by DVSST: changing frequency levels and detecting when the frequency can be scaled. Changing the processor frequency from one level to another is (approximately) constant, and was measured on the Rabbit 2000 processor to be 120 μs per frequency change with the main oscillator.

The second source of overhead is largely dependent on how the algorithm detects when it is possible to scale the processor frequency. When a task is released, a check is made to see if the frequency needs to be increased (i.e., if the task $\in$ *TD*). A timer list is used to detect when it is possible to scale down the processor frequency. A timer is set when the task is released and canceled if the task is released again before the timer expires. The processor frequency is scaled down by e_i/p_i whenever a timer expires for task T_i.

The timer list is implemented as a sorted linked list with no effort made to optimize list insertion since most applications that use the Rabbit 2000 have very few tasks; our application has only six tasks and the version of μC/OS-II that comes with the board only supports 64 tasks. Thus, insertion into a list of size *n* has cost O(*n*). The worst case occurs when an entry needs to be inserted at the end of the list. The list insertion time was measured for up to 512 tasks with random deadlines. For each list length from

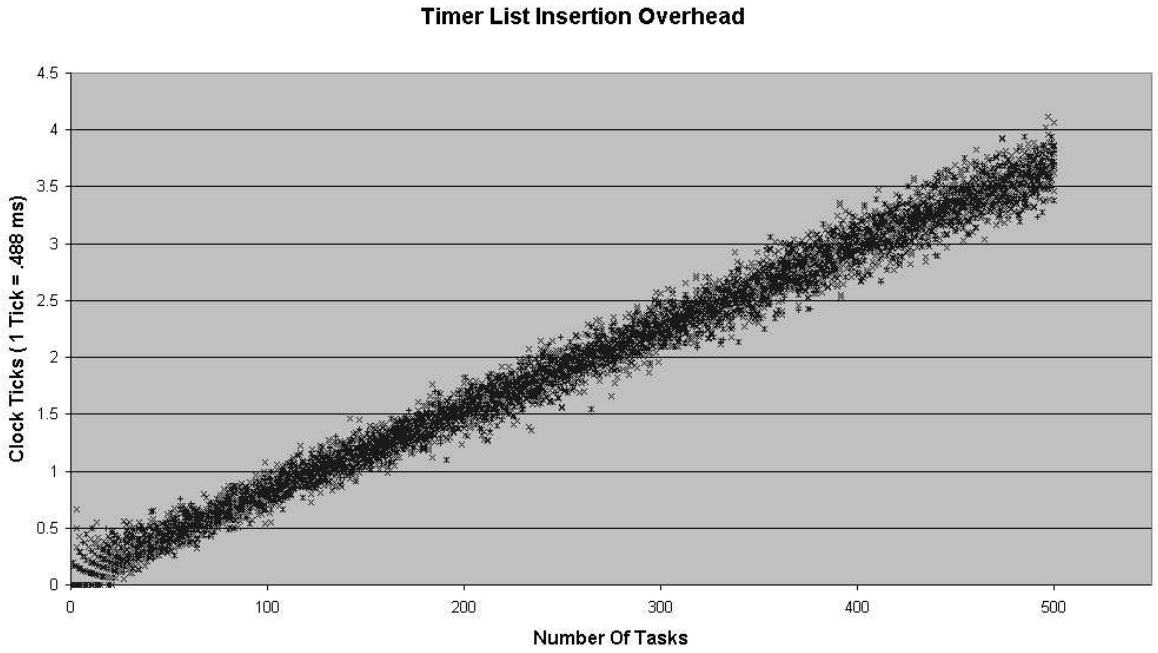

Figure 6: Timer list insertion overhead.

one to 512, the test was repeated a number of times equal to the list length with random timer values to be inserted. The insertion time was measured for each insertion and the average time of these values for each list length was recorded. The graph shown in Figure 6 plots the average timer list insertion time verses the number of tasks from 20 such experiments. Time is measured in terms of periodic clock ticks on the Rabbit 2000, which occur at a rate of 2kHz or one clock tick every 488μS.

The average insertion time is less than 1 clock tick for a list with less than 125 entries, as shown in Figure 4. The insertion time is about 4 clock ticks (2 ms) for 512 entries. Clearly a more efficient implementation of the timer list should be used for large task sets.

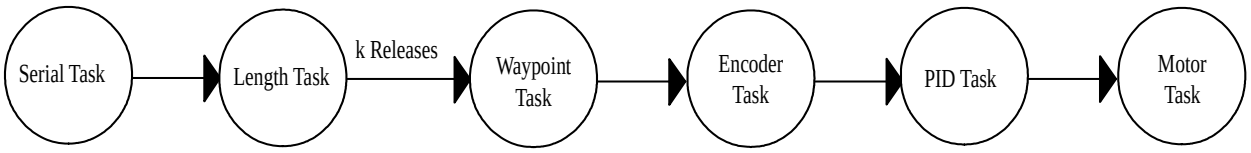

Figure 7: RSM processing graph.

5.4 Power Savings for a Robotic Highway Safety Marker

The Robotic Highway Safety Marker (RSM) is an automated safety devices designed to improve road construction work-zone design and safety. A RSM is a semi-autonomous mobile robot that carries a highway safety marker, commonly called a barrel. The RSMs operate in groups that consist of a single lead robot—called the foreman—and worker robots. To date, one foreman and six worker prototype RSMs have been developed. Each worker RSM has a Rabbit 2000 processor running our modified μC/OS-II. The prototype foreman is more sophisticated than the worker RSMs.

Control of the RSM group is hierarchical and broken into two levels—global and local control—to reduce the per-robot cost. The foreman robot performs global control. To move the robots, the foreman locates each RSM, plans its path, communicates destinations points (global waypoints), and monitors performance. Local control is distributed to each individual RSM, which do not have knowledge of other robots and only performs local tasks.

The code for the RSM is implemented as a sporadic task set. The task set only executes after it receives a new waypoint from the foreman. A path from the initial position of the RSM to the new waypoint is computed as a parabola decomposed into multiple local waypoints. The number of local waypoints depends on the length of the path. The following six sporadic tasks comprise the RSM task set.

- **Serial Task:** reads commands from the foreman via a RF transceiver, converts the command to target destinations, and stores the destinations in a shared queue data structure.
- **Length Task:** calculates the path length, number of iterations, and other values for each target destination.
- **Waypoint Task:** calculates the desired wheel angles for each iteration of a PID control loop.
- **PID Task:** does the PID control for each iteration.
- **Encoder Task:** reads the current wheel angles.
- **Motor Task:** sends commands to each motor.

An abstract processing graph for this task set is shown in Figure 7. The precedence relations shown in Figure 7 represent the logical precedence constraints on the data processing and do not reflect actual

release patterns. For example, to reduce latency in the processing graph, the last four nodes in the processing graph can be released simultaneously with deadline ties broken in favor of producer nodes, as described in [3]. The Serial task is released when data is available on the serial port. When data arrives, the Serial task converts it to a target destination, places it in a shared data structure and releases the Length task. Semaphores are not need to synchronize access to the data structure, which results in a fully preemptable task set. The Length task calculates the first two local waypoints before the robot begins to move. As the robot moves to waypoint i, waypoint i+2 is computed. The design ensures that waypoint i+2 is computed before waypoint i is reached.

This task set is modeled as a sporadic task set because the serial task receives commands with a minimum separation of 7.8125ms. The length task is executed the same number of times the serial task is executed. The number of times that Waypoint, PID, Encoder and Motor are executed depends on the number of local waypoints that need to be computed to reach the next global waypoint, which is dependent on the path length. Thus, for each execution of the serial task there may be a different number of executions for the Waypoint, PID, Encoder and Motor tasks. However, each task has a minimum separation period, as shown in Table 2.

The execution time for these tasks is very deterministic for two reasons. First the Rabbit 2000 has no cache memory, which eliminates memory-caching effects on execution time. Second the tasks repeat almost the same operation each time, with the exception of system initialization where some of the tasks execute a few more lines. Therefore the execution time of these tasks is usually very close to their WCET. The task execution times, shown in Table 2, were determined using an oscilloscope and free I/O pins on the processor.

Task	Period	Execution Time	e_i / p_i
Serial	7.8125ms	100µs	.0128
Length	7.8125ms	1ms	.128
Way Point	3 ∗7.8125ms	2.5ms	.1066
Encoder	3 ∗7.8125ms	350µs	.0149
PID	3 ∗7.8125ms	1.06ms	.04522
Motor	3 ∗7.8125ms	250µs	.0106

Table 2: RSM sporadic task set parameters.

The maximum utilization for the task set is U = 0.31812, which occurs when all of the tasks execute in a periodic mode for an extended interval of time. If we have no idle periods over an extended interval of time, the lower bound on utilization is when we have only one execution of the serial and length task followed by a very large number of executions of the other tasks. This will result in a processor utilization slightly greater than

$$\frac{e_{Waypoint}}{p_{Waypoint}} + \frac{e_{Encoder}}{p_{Encoder}} + \frac{e_{PID}}{p_{PID}} + \frac{e_{Motor}}{p_{Motor}} = .17732$$

Depending on when commands arrive and the length of the path to be computed, a wide range of utilization values is possible. For any case, the theoretical maximum power savings will be $1-U_\tau$ (as shown in Section 4.2), where U_τ is the utilization over the time interval τ. The actual power savings achieved is less because we cannot scale the frequency to the desired value; instead we scale it to the nearest upper level of frequency available on the Rabbit 2000, as described in Section 5.2.

As mentioned in Section 5.1, the Rabbit 2000 provides frequency scaling but does not directly adjust the voltage with the frequency. Thus, power savings can be linearly proportional to frequency scaling at best. However, since the Rabbit 2000 provides only a limited number of levels, rather than the unlimited number assumed in theory, there will be a difference between the actual savings and the theoretical power savings.

Figures 8a and 8b show the difference between the actual and the theoretical power savings. The normalized average power savings is plotted against relative utilization values, where the relative utilization is the ratio of a possible task utilization value to the maximum task utilization (0.31812). Figure 8a shows the normalized theoretical and actual power savings for the task set verses the relative utilization when there are no idle periods. That is, the robot is constantly moving but with destination commands of varying distance. In this case, the minimum relative utilization is 0.55739. Figure 8b shows the normalized theoretical and actual power savings when we have idle periods. That is, when the robot stops for intervals of time.

Note that the actual power savings deviate from a linear pattern even though only the processor frequency is scaled and the voltage remains constant. This is because when the frequency is scaled on the Rabbit 2000, it draws less current and the rate at which the current increases or decreases with each frequency level is not exactly linear.

The average ratio of the actual savings to the theoretical savings in both cases is about 83%. This means that DVSST achieved 83% of the theoretical power savings on the Rabbit 2000 for this application.

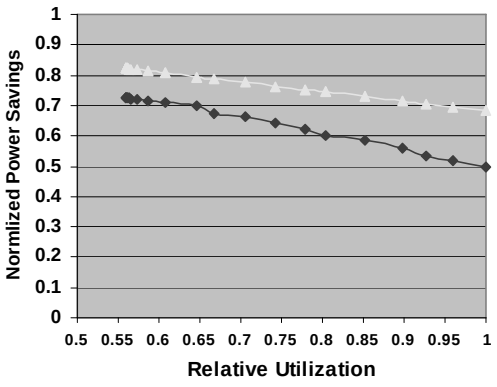

Figure 8a: Power savings with robot constantly moving.

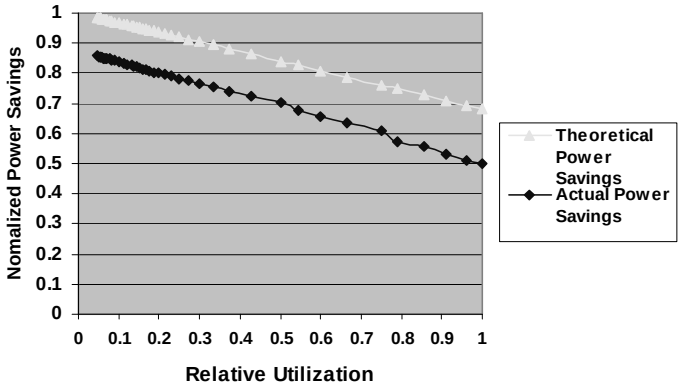

Figure 8b: Power savings with robot not constantly moving.

If the task set were executed at a periodic rate, the DVSST would run the processor at a frequency equal to the task utilization, which is the same as the Static Voltage Scaling algorithm of [19]. Other DVS algorithms from the literature are unlikely to improve power savings much, even if the task set executes periodically, because they try to take advantage of the case when tasks do not execute with their WCET. In this application, however, task execution time is very deterministic and there is very little difference between average execution time and WCET.

6. Conclusion

A dynamic voltage-scaling algorithm called DVSST was presented for sporadic task sets executed under EDF scheduling. It was shown that $U \leq 1$ is a necessary and sufficient schedulability condition for fully preemptive task sets. DVSST is an inter-task DVS algorithm and the only attempt to save power when jobs execute for less than their WCET is to scale the processor to a minimum frequency level whenever no jobs are pending. DVSST assumes that resources are not shared between tasks; DVS for resource-sharing sporadic tasks remains an open problem.

DVSST has been implemented in a modified version of μC/OS-II that supports EDF scheduling, and tested with a real-time embedded application, the Robotic Highway Safety Marker (RSM). Though DVSST is theoretically optimal, results shows that DVSST saves an average of 83% of the maximum possible theoretical power savings for that application on the Rabbit 2000 processor. Differences between theoretical and actual savings are due to the limited number of frequency levels supported by the Rabbit 2000 processor.

References

[1] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt. A. Veidenbaum, and A. Nicolau. Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints in the COPPER Framework. *Proceeding of Design, Automation and Test in Europe Conference (DATE)*, March 2002.

[2] M. Fleischmann. Crusoe Processor Products and Technology, LongRun Power Management - Dynamic Power Management for Crusoe Processors. http://www.transmeta.com/pdf/white_papers/paper_mfleischmann_17jan01.pdf, Transmeta Inc., January 17, 2001.

[3] S. Goddard and K. Jeffay. Analyzing the Real-Time Properties of a Data flow Execution Paradigm using a Synthetic Aperture Radar Application. *Proc. 3rd IEEE Real-Time Technology & Applications Symp*., Montreal, Canada, pp. 60–71, June 1997.

[4] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power Optimization of Variable-Voltage Core-Based Systems. *IEEE Trans. Computer-Aided Design*, vol. 18, no. 12, pp. 1702-1714, Dec. 1999.

[5] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors. *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 178–187, December 1998.

[6] Intel XScale microarchitecture, http://developer.intel.com/design/intelxscale.

[7] T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. *Proc. of ISLPED*, pp. 197–202, Aug. 1998.

[8] H. Kawaguchi, Y. Shin, and T. Sakurai. Experimental Evaluation of Cooperative Voltage Scaling (CVS): A Case Study. *Proceedings of IEEE Workshop on Power Management for Real-Time and Embedded Systems*, pp. 17-23, May 2001.

[9] W. Kim, J. Kim and S –L. Min. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. *Proceedings of Design Automation and Test in Europe (DATE'02)*, Paris, France, March 2002.

[10] J. Labrosse. *The Real Time Kernel MicroC/OS-II*, CMP Books, May 2002.

[11] C-M. Lee, Implementing Rate-Based Execution in MicroC/OS-II. Mater's Project, Dept. of CSE, University of Nebraska-Lincoln, November 27, 2002.

[12] Y.-H. Lee and C. M. Krishna. Voltage-Clock Scaling for Low Energy Consumption in Real-Time Embedded Systems. *Proceedings of the Sixth Int'l Conf. on Real Time Computing Systems and Applications*, pp. 272-279, 1999.

[13] Y-H. Lee, Y. Doh and C. M. Krishna. EDF Scheduling Using Two-Mode Voltage-Clock-Scaling for Hard Real-time Systems. *Proc. of CASES 2001*, pp. 221-228, 2001.

[14] C.L.Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, Vol.20, pp.46–61, 1973.

[15] J. Luo and N. K. Jha. Power-conscious Joint Scheduling of Periodic Task Graphs and Aperiodic Tasks in Distributed Real-time Embedded Systems. *Proceedings of ICCAD*, pages 357--364, Nov 2000.

[16] A. Manzak and C. Chakrabarti. Variable Voltage Task Scheduling for Minimizing Energy or Minimizing Power. *Proceedings IEEE Int. Conf. on Acoustic, Speech, and Signal Processing (ICASSP'00)*, pp. 3239–3242, June 2000.

[17] A.K.-L. Mok. Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment. Ph.D. Thesis, MIT, Dept. of EE and CS, MIT/LCS/TR-297, May 1983.

[18] D. Mosse, H. Aydin, B. Childers and R. Melhem, Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications. *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, Philadelphia, PA, Oct. 2000.

[19] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *Proc. of the 18th ACM Symp. on Operating Systems Principles*, 2001.

[20] G. Quan and X. Hu. Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors. *Proceedings of DAC'01: IEEE/ACM Design Automation Conference*, pp. 828-833, June 2001.

[21] Rabbit Semiconductors. Rabbit 2000 Microprocesser User's Manual, http://www.rabbitsemiconductor.com/documentation/docs/manuals/Rabbit2000/UsersManual

[22] D. Shin and J. Kim. A Profile-Based Energy-Efficient Intra-Task Voltage Scheduling Algorithm for Hard Real-Time Applications, *Proc. of ISLPED*, 2001.

[23] D. Shin, J. Kim, and S. Lee. Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications. *IEEE Design and Test Computers*, 18(2):20–30, 2001.

[24] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. *Proceedings of the Design Automation Conference*, pp. 134–139, June 1999.

[25] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. *Proceedings of the International Conference on Computer-Aided Design*, pp. 365–368, November 2000.

[26] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 13–23, November 1994.

[27] W. Wolf. *Modern VLSI Design*, Prentice Hall Modern Semiconductor Design Series, Third Edition 2002.

[28] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. *IEEE Symposium on Foundations Computer Science*, pp. 374–382, Oct. 1995.

[29] F. Zhang and S. T. Chanson, Processor Voltage Scheduling for Real-Time Tasks with Non-Preemptable Sections. *Proceedings of the 23rd IEEE International Real-Time Systems Symposium*, Austin, Texas, pp. 235–245, Dec. 2002.