Energy Aware Non-Preemptive Scheduling for Hard Real-Time Systems

Ravindra Jejurikar
Center of Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697
jezz@ics.uci.edu

Rajesh Gupta
Department of Computer Science
University of California, San Diego
La Jolla, CA 92093
gupta@cs.ucsd.edu

Abstract

Slowdown based on dynamic voltage scaling (DVS) provides the ability to perform an energy-delay tradeoff in the system. Non-preemptive scheduling becomes an integral part of systems where resource characteristics makes preemption undesirable or impossible. We address the problem of energy efficient scheduling of non-preemptive tasks based on the Earliest Deadline First (EDF) scheduling policy. We present the stack based slowdown *algorithm that builds upon the optimal feasibility test for non-preemptive systems. We also propose a dynamic slack reclamation policy to further enhance energy savings. Simulation results show on an average* 15% *energy savings using static slowdown factors and* 20% *savings with dynamic slowdown, over known slowdown techniques.*

1 Introduction

The concept of a task that is invoked periodically is central to a real-time system. Tasks are executed on a processor and must complete execution in a timely manner. Based on the task characteristics, priorities are assigned to tasks, which drive the scheduling decisions. Task scheduling can be classified into two broad categories: *preemptive* scheduling and *non-preemptive* scheduling. Under preemptive scheduling, the current task execution can be preempted by a higher priority task, whereas under non-preemptive scheduling, a higher priority task can be scheduled only after the completion of the current task. Though preemptive scheduling can guarantee a higher system utilization, there are scenarios where properties of hardware devices and software configuration make preemption either impossible or prohibitively expensive. Non-preemptive scheduling also has the advantages of accurate response time analysis, ease of implementation, no synchronization overhead and reduced stack memory requirements. Non-preemptive scheduling is used in light weight multi-tasking kernels and has been shown to be beneficial in multimedia applications [3]. This work focuses on energy efficient scheduling of non-preemptive real-time tasks.

The two major techniques of minimizing the processor energy consumption are: *shutdown* and *slowdown*. Slowdown through dynamic voltage and frequency scaling (referred to as DVS) is known to be effective in energy minimization [18, 1, 23]. A reduction in the supply voltage decreases the power consumption of the processor because of the quadratic relationship between power and voltage. The power consumption, P, is given by:

$$P = C_{eff} \cdot V_{dd}^2 \cdot f \tag{1}$$

where C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage and f is the operating frequency. However, the transistor gate delay (and hence frequency) depends on the voltage and a decrease in voltage has to be accompanied by a decrease in processor frequency. There is a linear dependence between frequency and voltage [20], resulting in a linear increase in the execution time of a task. Thus voltage scaling provides the ability to perform an energy-delay tradeoff in the system. Real-time systems have strict timing requirements and slowdown has to be performed judiciously in achieving our goal of minimizing energy.

Previous works on energy aware scheduling have mainly focussed on preemptive scheduling. Among the earliest works, Yao *et al.* [21] presented an optimal off-line algorithm to schedule a given set of jobs with arrival times and deadlines. For a similar task model, optimal algorithms have been proposed for fixed priority scheduling [15, 22] and scheduling over a fixed number of voltage levels [11], [7]. Energy efficient scheduling of periodic real-time task sets has also been addressed. Real-time feasibility analysis has been used in previous works to compute static slowdown factors for tasks [18], [4]. Aydin *et al.* [1] have addressed the problem of energy minimization considering the task power characteristics. When tasks complete earlier than the worst case, there is opportunity for additional (dynamic) slowdown which increases the energy savings

[14, 2, 10]. The problem of maximizing the system value for a specified energy budget, as opposed to minimizing the total energy, is addressed in [17, 16]. Note that these works assume a preemptive task system. Non-preemptive scheduling has been addressed primarily in the context of multiprocessor scheduling [5]. Zhang *et al.* [24] have given a framework for non-preemptive task scheduling and voltage assignment for dependent tasks on a multi-processor system. They have formulated the voltage scheduling problem as an integer programming problem. The problem of minimizing the energy consumption by performing a slowdown tradeoff in the computation and communication subsystems is addressed in [12].

In this work, we address non-preemptive scheduling of periodic tasks on a uni-processor system. Zhang and Chanson have addressed energy efficient scheduling for the same model and have presented the dual speed (DS) algorithm [23]. The DS algorithm computes two speeds, a low speed L based on an analysis for an independent task set and a high speed H taking into account the blocking time arising due to non-preemption. Note that the computation of the high speed, based on a sufficient feasibility test, is not optimal. This can result in using a H speed higher than required and consume more energy. Furthermore, the dual speed algorithm switches to the high speed whenever any task is blocked, which may not be needed during every task blocking. We propose a novel algorithm (the *stack based slowdown* algorithm) that minimizes the transitions to a higher speed by computing different slowdown factors based on the blocking task. The algorithm is based on the optimal feasibility test under non-preemptive scheduling [8]. We also enhance the stack based slowdown algorithm with dynamic slack reclamation for additional energy savings. While earlier works do not perform dynamic slowdown when tasks are blocking high priority tasks, we overcome this limitation in this work.

The rest of the paper is organized as follows: Sect. 2 describes the preliminaries and formulates the problem. In Sect. 3, we present the stack based slowdown algorithm for non-preemptive task scheduling. A dynamic slack reclamation algorithm follows in Sect. 4. The simulation results are given in Sect. 5 and we conclude in Sect. 6.

2 Preliminaries

2.1 System Model

The system consists of a task set of n periodic real time tasks, denoted as $\Gamma = \{\tau_1, \ldots, \tau_n\}$. Each task τ_i is a 3-tuple $\{T_i, D_i, C_i\}$, where T_i is the period of the task, D_i is the relative deadline and C_i is the worst case task execution time (WCET) at the maximum processor speed. We assume the relative task deadline is equal to the period ($D_i = T_i$, for each task τ_i). Each invocation of the task is called a *job* and the k^{th} invocation of task τ_i is denoted as $\tau_{i,k}$. Tasks are non-preemptively scheduled on a single processor system. A task set is said to be *feasible* if all tasks meet the deadlines. The processor utilization for the task set, $U = \sum_{i=1}^{n} C_i/T_i \leq 1$ is a necessary condition for the feasibility of any schedule [13]. We say that an executing task is *blocking* another task, if a higher priority task is waiting in the ready-queue for the completion of current task execution. Equivalently, the higher priority task is *blocked* in the system. Note that a blocking task would have been preempted under a preemptive scheduling policy.

2.2 Processor Model

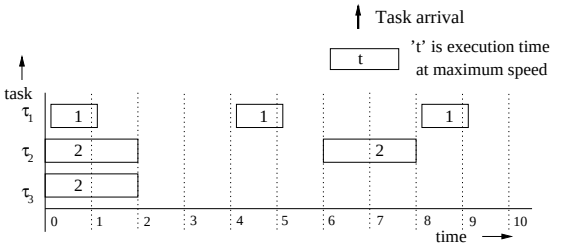
Recent processors such as the Intel XScale [6] and Transmeta Crusoe [19] support variable frequency and voltage levels, that can be varied at run-time. We define a *slowdown factor* as a normalized operating frequency. At a given instance, it is the ratio of the current frequency to the maximum processor frequency. Note that the voltage and frequency levels are tightly coupled, and a {frequency, voltage} pair is associated with each slowdown factor. We assume that processing speed can be varied over a discrete range, with f_{min} and f_{max} being the minimum and maximum operating frequency respectively. We normalize the speed to the maximum frequency to have discrete points in the interval $[\eta_{min}, 1]$, where $\eta_{min} = f_{min}/f_{max}$. The overhead incurred in changing the processor speed is assumed to be incorporated into the task execution time. Considering static and dynamic slowdown, a speed change can occur only when a task begins execution or when a higher priority task is blocked. This overhead is constant and can be incorporated in the worst case processing time of a task. Note that the same assumption is made in prior works [1],[2], [21],[23].

2.3 Motivating Example

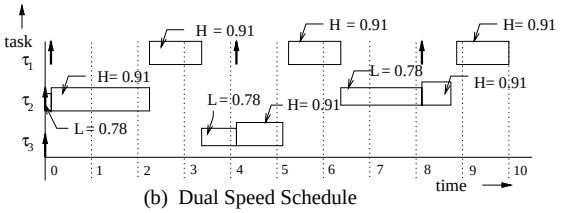
Consider a real time system comprised of 3 periodic tasks as described below,

$$\tau_1 = \{4, 4, 1\}, \tau_2 = \{6, 6, 2\}, \tau_3 = \{10, 10, 2\}$$

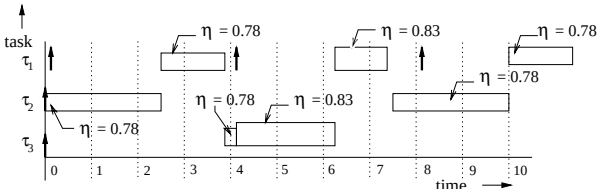
To generate a scenario where tasks are blocked, we assume that task $\tau_{2,1}$ (first instance of task τ_2) and $\tau_{3,1}$ arrive at time $t = 0$ and task $\tau_{1,1}$ arrives just after task $\tau_{2,1}$ begins execution. Figure 1(a) shows the jobs for each task at their arrival time and their workload at maximum speed. All task deadlines are equal to task period (arrival time of the next instance of the same task). The task schedule under the Dual Speed (DS) algorithm is shown in Fig. 1(b). The speeds computed by the DS algorithm [23] are $H = 0.916$ and $L = 0.78$. At time $t = 0$, the system begins execution



(a) Task description: Task arrival times and WCET at maximum speeed



(b) Dual Speed Schedule



(c) Improved Schedule

Figure 1. (a) Task arrival times and deadlines (NOT a task schedule), tasks τ_2 and τ_3 arrive at $t = 0$ and task τ_1 arrives slightly later than $t = 0$. (b) Task schedule based on the Dual speed (DS) algorithm. (c) Improved energy efficiency task schedule with fewer transitions to a higher speed.

at the L speed and schedules the highest priority ready task, $\tau_{2,1}$. Task $\tau_{1,1}$ arrives immediately and is blocked by task $\tau_{2,1}$. The processor speed is increased to H on blocking and jobs $\tau_{2,1}$ and $\tau_{1,1}$ execute at the H speed. We consider an improvement to the DS algorithm, whereby the system can switch back to the L speed, when a lower priority task than the blocking task begins execution. Thus the system changes to the L speed when task $\tau_{3,1}$ is scheduled. Job $\tau_{1,2}$ arrives during the execution of $\tau_{3,1}$ and the processor speed is again changed to H. The remaining portion of task $\tau_{3,1}$ and task $\tau_{1,2}$ are scheduled at the H speed. The complete task schedule with the speed transitions is shown in Fig. 1(b).

Note that the H speed is computed using a simple feasibility test in the DS algorithm. This can result in a higher H speed - thereby consuming more energy. The H speed can be computed using the optimal feasibility test in [8] (given by Theorem 1 in Section 3.1). For the given example, the optimal feasibility test computes a speed of $\eta = 0.833$ which suffices as the H speed. Furthermore, we show that switching to the H speed is not always required when a task is blocked. If a blocking task does not compromise the feasibility of higher priority tasks, the system can continue execution at the L speed even when tasks are blocked. An analysis of the task set (described later in Section 3) concludes that task τ_2 can execute at the L speed even when tasks are blocked. When task $\tau_{3,1}$ blocks a task, switching to a speed of $\eta_3 = 0.833$ ensures meeting all deadlines. Based on this analysis, an improved schedule is shown in Fig. 1(c). The system begins execution at the lower speed $L = 0.78$. When task $\tau_{2,1}$ blocks task $\tau_{1,1}$ at system start, the speed is unchanged and tasks $\tau_{2,1}$ and $\tau_{1,1}$ execute at the L speed as opposed to the H speed under DS algorithm. At $t = 3.85$, task $\tau_{3,1}$ begins execution at the L speed and task $\tau_{1,2}$ is blocked on arrival. The system switches to a speed of $\eta = 0.83$ (as opposed to a speed of $H = 0.91$) and all tasks with a deadline less than or equal to $t = 10$ are executed at this speed. All tasks meet the deadline and the schedule is shown in Fig. 1(c).

3 Static Slowdown Factors

3.1 Constant Static Slowdown

Feasibility conditions for non-preemptive scheduling are well studied [8]. The *optimal* feasibility condition based on the EDF scheduling policy (with no inserted idle intervals) is stated below.

Theorem 1 *[8] A periodic task set, sorted in non-decreasing order of the task period, can be feasibly scheduled under a non-preemptive EDF scheduling policy iff,*

$$\sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1 \tag{2}$$

$$\forall i, 1 < i \leq n; \forall t, T_1 \leq t \leq T_i : \quad C_i + \sum_{k=1}^{i-1} \lfloor \frac{t}{T_k} \rfloor C_k \leq t \tag{3}$$

Note that it suffices to check the feasibility at time instances corresponding to the deadline of higher and equal priority tasks, called the scheduling points. The scheduling points for each task τ_i are given by $S_i = \{kT_j | j = 1, \ldots, i; k = 1, \ldots, \lfloor \frac{T_i}{T_j} \rfloor\}$. Theorem 1 leads to the computation of the optimal constant static slowdown factor, as described next.

Corollary 2 *: A periodic task set, sorted in non-decreasing order of their period, can be feasibly scheduled under the non-preemptive EDF scheduling policy, at a constant slowdown of* η*, iff*

$$\frac{1}{\eta} \sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1 \tag{4}$$

$$\forall i, 1 < i \leq n; \forall t, T_1 \leq t \leq T_i : \quad \frac{1}{\eta} \left(C_i + \sum_{k=1}^{i-1} \lfloor \frac{t}{T_k} \rfloor C_k \right) \leq t \tag{5}$$

3.2 Stack Based Slowdown Algorithm

Though Corollary 2 can compute a constant slowdown, the system can be under-utilized at this slowdown and result in idle intervals. Similar to the dual speed algorithm[23], we can execute tasks at a lower processor speed $\overline{\eta}$, called the *base speed*, in the absence of task blocking. The base speed is the same as the L speed under DS algorithm and satisfies the following constraint:

$$\frac{1}{\overline{\eta}} \sum_{k=1}^{n} \frac{C_k}{T_k} \leq 1 \tag{6}$$

If a higher priority task is blocked due to the non-preemptive nature of the system, the execution speed may need to be increased to ensure all deadlines. The impact of the blocking arising from the current task execution can be computed. A slowdown factor of η_i, based on the blocking task τ_i, suffices if the following constraints are satisfied:

$$\forall t, T_1 \leq t \leq T_i : \quad \frac{1}{\eta_i} \left(C_i + \sum_{k=1}^{i-1} \lfloor \frac{t}{T_k} \rfloor C_k \right) \leq t \tag{7}$$

The above constraints compute a slowdown factor that ensures meeting the deadlines of tasks with a priority higher than the blocking task. Note that if η_i is lower than the current processor speed, no change in speed is required. If η_i is greater than the current speed and task τ_i blocks a task, then the processor speed is increased to η_i. The system switches back to the original speed (that before the system speed was increased to η_i) on executing a task with lower priority than that of task τ_i or if the system becomes idle. Thus we ensure that all tasks with a priority greater than or equal to that of task τ_i execute at a speed of at least η_i. The speed transitions resemble a *stack* operation and, indeed, we use a stack to implement the algorithm. Hence the proposed algorithm is referred to as the Stack Based Slowdown (SBS) algorithm.

3.2.1 Slowdown Algorithm

Algorithm 1 describes the proposed Stack Based Slowdown (SBS) algorithm. The algorithm maintains a stack S and each stack node, *sn*, has an associated slowdown (η) and a priority(P), represented as $sn(\eta, P)$. We use the notation $\eta(sn)$ and $P(sn)$ to represent the slowdown and priority of a stack node (sn) respectively. The stack is initialized with a slowdown factor equal to the *base speed* ($\overline{\eta}$) and a priority lower that the lowest priority that any job can achieve, which is represented by $-\infty$. This node is called the *base node* and is represented as $sn(\overline{\eta}, -\infty)$ (lines 4-5). At all times, let sn_t represent the node at the top of the stack and let J_c (an instance of task τ_c) be the current job executing in the system. The processor speed is always set to the slowdown factor at the top of the stack ($\eta(sn_t)$) and the system execution begins at the base speed (similar to the dual speed algorithm). While job J_c is executing, if job J_i with a priority higher than that of job J_c arrives, the higher priority job J_i is blocked due to non-preemption. If the current job is blocking a higher priority job and the slowdown factor η_c (of job J_c) is greater than the stack top slowdown factor $\eta(sn_t)$, then a new node is pushed on the stack with a slowdown and priority that of job J_c. In that case, a node $sn(\eta_c, P(J_c))$ is pushed on the stack (lines 6-12). A stack top node (sn_t) is popped off the stack when a job with a priority lower than the stack top priority ($P(sn_t)$) is executed (lines 13-17). This ensures that all jobs with priority higher than that of the blocking task (J_c) are executed at a slowdown of at least η_c, which guarantees all higher priority task deadlines. Since the stack is initialized with a base node with priority $-\infty$ (equivalent to an infinitely large task deadline), the base node is never popped off the stack. If the system becomes idle, all nodes except the base node are popped from the stack (line 19). We prove that the SBS algorithm guarantees the feasibility of the system.

Algorithm 1 Stack Based Slowdown (SBS) Algorithm

1: **Notation :**
2: J_c : the current job (instance of τ_c) executing in the system
3: sn_t : the node at the top of the stack
4: **Stack Initialization :**
5: Push base node $sn(\overline{\eta}, -\infty)$ on the (empty) stack;
6: **On arrival of job J_i in the system:**
7: **if** (processor IDLE before task arrival) **then**
8: $SetSpeed(\eta(sn_t))$;
9: **else if** ($P(J_i) > P(J_c)$ **and** $\eta_c > \eta(sn_t)$) **then**
10: Push $sn(\eta_c, P(J_c))$ on the stack;
11: $SetSpeed(\eta_c)$;
12: **end if**
13: **On execution of each job J_i :**
14: **while** ($P(J_i) < P(sn_t)$) **do**
15: Pop sn_t from the stack; {Pop the stack top node}
16: **end while**
17: $SetSpeed(\eta(sn_t))$;
18: **On system idle:**
19: Pop all nodes from stack, *except* the base node;

Theorem 3 *A task set, sorted in non-decreasing order of the relative task period, can be feasibly scheduled by the stack based slowdown algorithm at a base speed $\overline{\eta}$ and a*

slowdown factors η_i *for task* τ_i *if,*

$$\frac{1}{\bar{\eta}}\left(\sum_{k=1}^{n}\frac{C_k}{T_k}\right) \leq 1 \tag{8}$$

$$\forall i, 1 < i \leq n;\ \forall t, T_1 \leq t \leq T_i: \quad \frac{1}{\eta_i}\left(C_i + \sum_{k=1}^{i-1}\lfloor\frac{t}{T_k}\rfloor C_k\right) \leq t \tag{9}$$

Proof: Suppose the claim is false and let t be the first time that a task instance misses its deadline. Let t' be the the latest time before t such that there are no pending jobs with arrival times before t' and deadlines less than or equal to t. Since no requests can arrive before system start time $(time = 0)$, t' is well defined. Let A be the set of jobs that arrive no earlier than t' and have deadlines at or before t. By choice of t', the system is either idle before t' or executing a job with a deadline greater than t. We consider both these cases separately. Note that by the EDF priority assignment, only jobs in A are allowed to *start* execution in $[t', t]$. Also, there are pending requests of jobs in A at all times during the interval $[t', t]$ and the system is never idle in the interval.

Case I: If the system were idle at time t', then only the jobs in A execute in the interval $[t', t]$. Let $X = t - t'$. Since all the jobs are periodic in nature and the jobs in A arrive no earlier than t', the number of executions of each task τ_i in A in the interval X is bounded by $\lfloor\frac{X}{T_i}\rfloor$. By the stack based slowdown algorithm, the base node is never popped during the entire execution. Nodes pushed on the stack have a speed higher than the base speed, and all tasks execute at a speed greater than or equal to the base speed $\bar{\eta}$. Thus the execution time of each job is bounded by $C_i/\bar{\eta}$. Since a task misses its deadline at time t, the execution time for the jobs in A exceeds the interval length X.
Therefore,

$$\sum_{i=1}^{n}\lfloor\frac{X}{T_i}\rfloor C_i\frac{1}{\bar{\eta}} > X$$

which implies

$$\frac{1}{\bar{\eta}}\sum_{i=1}^{n}\frac{C_i}{T_i} > 1$$

which contradicts (8).

Case II: Let J_b be the job that blocks a job in A, executing at time t' with a deadline greater than t. Since J_b is executing at time t', with a deadline greater that t, $X < T_b$ and $A \subseteq \{\tau_1, \ldots \tau_k\}$, where $T_k < X$ and $k < b$. Only the task J_b and the tasks in A execute in the interval $[t', t]$. When the task τ_b blocks another task, if the stack top slowdown is smaller than η_b, then η_b is pushed on the stack. Since this stack node is not popped until all jobs with priority greater than τ_b execute, the speed of all jobs in this interval is at least η_b (Note that blocking of other jobs in the interval can only increase the speed to greater than η_b). Thus, the total execution time of these jobs is bounded by $\frac{1}{\eta_b}(C_b + \sum_{i=1}^{b-1}\lfloor\frac{X}{T_i}\rfloor C_i)$. Since a task misses its deadline at time t, the execution time for the jobs in A and that of job J_b exceeds X, the length of the interval. Therefore,

$$\frac{1}{\eta_b}(C_b + \sum_{i=1}^{b-1}\lfloor\frac{X}{T_i}\rfloor C_i) > X$$

Since $X < T_b$, this contradicts (9). Hence all tasks meet the deadline when scheduled by the stack based slowdown algorithm. ■

3.2.2 Computational Complexity

The computation of task slowdown factors depends on the number of scheduling points for each task. The number of scheduling points are determined by the maximum task period and the computation of slowdown factors is fast in practice. Theoretically, the number of scheduling points can be pseudo polynomial in the problem size. Note that the computation of slowdown factors is done off-line and it is beneficial to compute them with the optimal feasibility test.

4 Dynamic Slack Reclamation

Dynamic slack arises due to early task completions as well as when tasks execute at a higher speed than the base speed. This slack can be reclaimed to further reduce the processor speed, resulting in increased energy savings.

4.1 Motivation

One of the limitations of prior works is that they do not reclaim slack when task are blocked in the system (e.g. DSDR [23]). This can severely limit slack reclamation under non-preemptive scheduling and we overcome this limitation in our work. When no higher priority tasks are blocked, it is known that tasks can reclaim the higher priority (than the task priority) run-time while meeting all deadlines. However, these techniques cannot be directly applied to blocking tasks. We show that reclaiming higher priority run-time (slack) when (higher priority) tasks are blocked can lead to tasks missing the deadline. We illustrate this with an example, shown in Fig. 2. The task set is comprised of three tasks with the following parameters:

$$\tau_1 = (5, 5, 2),\ \tau_2 = (10, 10, 3),\ \tau_3 = (10, 10, 3)$$

The arrival times are as shown in Fig 2(a) with τ_1 arriving at $t = 1$ and tasks τ_2 and τ_3 arriving at $t = 0$. The system is feasible with $U = \bar{\eta} = 1.0$ and the stack top slowdown is always be $\eta(sn_t) = 1.0$. The task schedule, where tasks reclaim higher priority run-time is shown in Fig. 2(b). Task

τ_2 completes earlier at time $t = 0.5$, leaving an unused run-time of 2.5 time units. This run-time has the same priority as task τ_3 and can be reclaimed when no tasks are blocked. Using the available free run-time task τ_3 computes a slowdown factor to complete at time $t = 6.0$ (free run-time of $t = 2.5$ and its own budget of 3 time units). Task τ_1 arrives at time $t = 1$ and is blocked by task τ_3. If task τ_3 continues execution at the same speed, we see that task τ_1 misses its deadline of $t = 6.0$. It can be seen than the blocking time for task τ_1 cannot exceed 3 time units, else it can miss its deadline. Thus we need to limit the blocking time to 3 time units. This can require computing a different slowdown factor when a task is blocked. When task τ_1 arrives at time $t = 1$, a new slowdown factor is computed so that the remaining portion of task τ_3 completes in 3 time units. This bounds the blocking time of task τ_1 to meet its deadline as shown in Fig. 2(c).

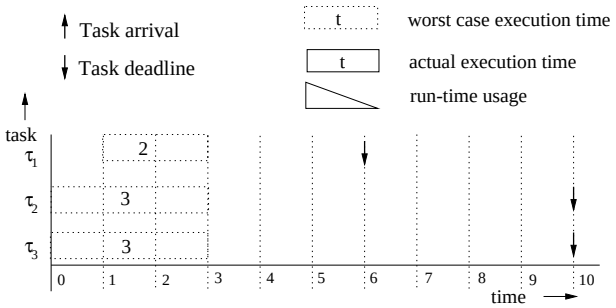
4.2 Computing Maximum Blocking Time (B^{max})

We compute the maximum blocking time (B_i^{max}) permissible to task τ_i while guaranteeing all higher priority task deadlines. B_i^{max} is used to bound the blocking time under slack reclamation. When task τ_i blocks another task, the stack top slowdown is at least $max(\eta_i, \overline{\eta})$ and higher priority tasks (than the blocking task) are executed at least at this slowdown factor. Given tasks are sorted by their period, we compute B_i^{max} for task τ_i such that the following condition is satisfied.

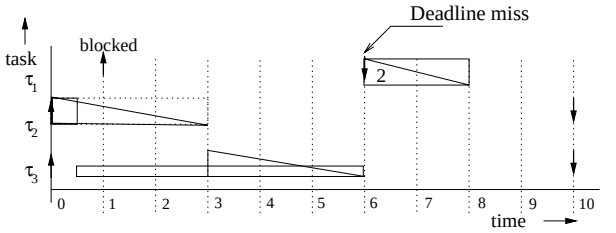
$$\forall t, T_1 \leq t \leq T_i : \quad B_i^{max} + \frac{1}{max(\eta_i, \overline{\eta})} \left(\sum_{k=1}^{i-1} \lfloor \frac{t}{T_k} \rfloor C_k \right) \leq t \tag{10}$$

4.3 Slack Reclamation Preliminaries

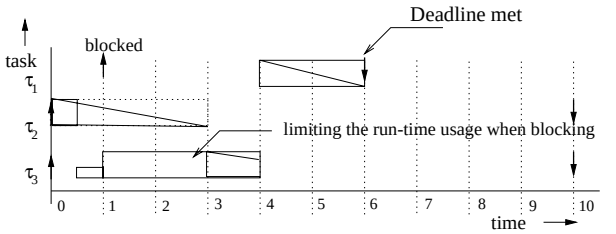
Dynamic slack reclamation algorithms manage the allocation of time budgets for tasks as well as the time budgets that can be reclaimed during execution. We define the *run time* of a job as the time budget allocated to the job, based on the task workload and the slowdown factor. The run-time of a job with a workload C (at maximum speed) and slowdown η, is C/η. Each run time has a time budget and a priority associated with it, and is represented by a pair (t, P). The priority of a run time associated with a job is the same as the job priority. A job consumes run time as it executes. The unused run time of jobs is maintained in a priority list called the *Free Run Time list (FRT-list)* [23]. A FRT-list is maintained sorted by priority of the run-time, with the highest priority at the head of the list. Run-time is always consumed from the head of the list. In addition to the run-time of a task, a *blocking run-time* is also maintained for each task τ_i. When a instance of task τ_i begins



(a) Task set description: Task arrival times and WCET



(b) Incorrect salck reclamation and deadline miss



(b) Correct Slack Reclamation under task blocking

Figure 2. Dynamic slack reclamation of blocking tasks: (a) Task arrival times and deadlines (NOT a task schedule). (b) Reclaiming higher priority run-time by a blocking task (τ_3) and task τ_1 missing the deadline. (c) Limiting the slack reclamation by the maximum blocking time for task τ_3 and all tasks meet the deadline.

execution, the blocking run-time for the job is initialized to B_i^{max}. The blocking run-time is used to limit the blocking time under slack reclamation.

Under the SBS algorithm, the stack top node, sn_t, is crucial and determines the time budget for each job execution. We say a stack node sn_i dominates sn_j if $\eta(sn_i) > \eta(sn_j)$ or equivalently sn_j is dominated by sn_i. Since only higher slowdown factors than the stack top slowdown factor are pushed on the stack, a node dominates all nodes below it in the stack. We use similar notation and definitions used in [23] to explain our algorithm.

- J_i : the current job of task τ_i.
- $R_i^r(t)$: the available run time of job J_i at time t.
- $B_i^r(t)$: the blocking run-time available to J_i at time t.

- $R_i^F(t)$: the free run time available for Job J_i - the run time from the FRT-list with priority $\geq P(J_i)$
- $C_i^r(t)$: the residual workload of job J_i.
- $R_i^M(D)$: The difference between the run-time computation based on the slowdown of node sn_D and that of its *immediately dominated* node (adjacent node below node sn_D in the stack), sn_d, on a stack. If η_D and η_d be the slowdown of nodes sn_D and sn_d respectively ($\eta_D > \eta_d$), then $R_i^M(D) = (\frac{C_i}{\eta_d} - \frac{C_i}{\eta_D})$, is the difference in run time at the two speeds.

Algorithm 2 Dynamic Slack Reclamation Algorithm

1: **Stack Initialization:**
2: Initialize stack with a base node $(\overline{\eta}, -\infty)$
3: FRT-list is initially empty.
4: **On arrival of job J_i in the system:**
5: **if** (J_c is running **and** $P(J_i) > P(J_c)$) **then**
6: **if** $(\eta_c > \eta(sn_t))$ **then**
7: Push sn$(\eta_c, P(J_c))$ on the stack;
8: **end if**
9: $setSpeed\left(\frac{C_c^r(t)}{min\{(R_c^r(t)+R_c^F(t)),B_i^r(t)\}}\right)$;
10: **end if**
11: **On execution of each job J_i :**
12: **while** $(P(J_i) < P(sn_t))$ **do**
13: Pop the stack top node;
14: **end while**
15: $R_i^r(t) = C_i/\eta(sn_t)$;
16: $B_i^r(t) = B_i^{max}$;
17: **for** (each stack node sn_D dominating the base node) **do**
18: Add $(R_i^M(D), P(sn_D))$ to FRT-list;
19: **end for**
20: $setSpeed(\frac{C_i^r(t)}{R_i^r(t)+R_i^F(t)})$;
21: **On Completion of job J_i:**
22: Add run-time $(R_i^r(t), P(J_i))$ to FRT-list;
23: **On System Idle:**
24: Pop all nodes except base node

4.4 Slack Reclamation Algorithm

We propose a slack reclamation scheme that works with the SBS algorithm and is called the Stack Based Slowdown with Dynamic Reclamation (SBS-DR) algorithm. The SBS-DR policy is described in Algorithm 2. The system initializes the stack with a base node and an empty FRT-list (line 2). The conditions under which stack nodes are pushed and popped are the same as the SBS algorithm (shown in lines 6-8 and lines 12-14 of Algorithm 2). The two sources of slack reclamation are (1) early completion of tasks and (2) execution at speed greater than the base speed ($\overline{\eta}$). Before the execution of each job, the algorithm reserves a run-time for the job based on the slowdown factor of the stack top node, sn_t. As shown in line 15, job J_i (an instance of task τ_i) is assigned a run-time of $C_i/\eta(sn_t)$. The blocking run-time of job J_i is initialized to B_i^{max}. If node sn_t is not the base node then tasks are executed faster than the base speed. The extra time budget that would be available if the task were executed at the base speed, is the slack in the system. For each stack node sn_D dominating the base node, the difference in the budget arising from a slowdown of $\eta(sn_D)$ and the slowdown factor of the (adjacent) *immediate dominated* node $\eta(sn_d)$, is added to the FRT-list with a priority of $P(sn_D)$ as shown in line 18 of the algorithm. The other source of dynamic slack is when a task completes before consuming its allocated time budget. On job completion, the unused run time is added to FRT-list with the same priority as the job priority. When no tasks are blocked, a job J_i can use its own run time as well higher priority run-time from FRT-list (i.e. $R_i^F(t)$. The task slowdown factor is computed to utilize the maximum time budget available (line 20). The dynamic slowdown factor is the ratio of the residual workload to the available runtime, as described in line 20 of Algorithm 2. When tasks are blocked, the algorithm ensures that the blocking task (τ_c) completes by its maximum blocking task B_c^{max}. This is achieved by limiting the run-time available for task τ_c to $min\{(R_i^F(t) + R_i^r(t)), B_i^r(t)\}$. The slowdown computation on blocking is shown in line 9 of Algorithm 2. Note that it suffices to recompute the slowdown when the first task is blocked during the execution of the blocking task (no re-computation is required for subsequent blocking by the same job (J_c)).

The following rules are used by the slack reclamation algorithm. Note that the rules need to be applied only on the arrival and completion of a task in the system.

- As job J_i executes, it consumes run time at the same speed as the wall clock (physical time) [23]. If $R_i^F(t) > 0$, the run time is used from the FRT-list, else $R_i^r(t)$ is used.
- When the system is idle, it uses the run time from the FRT-list if the list is non-empty.
- When a (high priority) task is blocked, the executing task consumes the blocking run-time at the same speed as the wall clock (physical time). If no task is blocked, then $B_i^r(t)$ is unchanged. Note that the blocking run-time management is performed in addition to the above mentioned run-time management.

We prove that tasks can reclaim the slack in this manner while guaranteeing all deadlines.

Theorem 4 *A task set, sorted in non-decreasing order of the relative task period, can be feasibly scheduled by the stack based slowdown with dynamic reclamation (SBS-DR) algorithm (Algorithm 2) at a base speed $\overline{\eta}$ and slowdown factor η_i for task τ_i if,*

$$\frac{1}{\overline{\eta}}(\sum_{k=1}^{n}\frac{C_k}{T_k}) \leq 1 \tag{11}$$

$$\forall t, T_1 \leq t \leq T_i: \quad \frac{1}{\eta_i}(C_i + \sum_{k=1}^{i-1}\lfloor\frac{t}{T_k}\rfloor C_k) \leq t \tag{12}$$

$$\forall t, T_1 \leq t \leq T_i: \quad B_i^{max} + \frac{1}{max(\eta_i, \overline{\eta})}\left(\sum_{k=1}^{i-1}\lfloor\frac{t}{T_k}\rfloor C_k\right) \leq t \tag{13}$$

The details of the proof are present in [9].

5 Experimental Setup

To evaluate the effectiveness of our proposed techniques, we perform simulations on randomly generated task-sets each containing 10 to 15 tasks. A mixed workload is used with task periods uniformly distributed in the following three ranges: [1000,4000], [5000,10000] and [15000,20000]. An initial processor utilization u_i of each task was uniformly assigned in the range [0.05, 0.10]. The worst case execution times (WCET) for each task was set to $u_i \cdot T_i$, at the maximum processor speed. The task execution times are scaled to ensure the feasibility of the task set under non-preemptive scheduling. The execution times are further reduced (uniformly scaled) to vary the processor utilization of the task set. We simulate several task sets in our experimentation and the average results are presented in the paper.

The processor power model is based on the dynamic power consumption in CMOS circuits, as indicated by (1). The operating voltage range for the processor is $0.6V$ to $1.8V$, which is the trend in current embedded processors. We normalize the operating speed and support discrete slowdown factors in steps of 0.05 in the normalized range.

5.1 Slowdown with no slack reclamation

We compare the energy savings of the following techniques based on statically computed slowdown factors (with each task assumed to execute up to its WCET) :

- Optimal Constant Slowdown (OCS) algorithm (Corollary 2)
- Dual Speed (DS) algorithm [23]
- Stack Based Slowdown (SBS) algorithm, proposed in this paper.

Note that, with no dynamic slack reclamation, the base speed ($\overline{\eta} = U$) is the lower bound on the task slowdown factor. However non-preemptive scheduling can demand a a higher slowdown factor (η_{max}) (as given by Corollary 2). The DS and the SBS algorithms exploit opportunities to execute at a speed lower than η_{max} to result in energy savings. The extent of energy savings are proportional to the difference in the slowdown factors given by $\overline{\eta}$ and η_{max}. The difference in the two speeds is captured by the gain factor (G_f), which is defined as $G_f = \frac{\eta_{max}}{\overline{\eta}} - 1$. For most of the generated task-sets, the gain factor was uniformly distributed in the range of 0.0 - 0.5 (for a few tasks with lower utilization, the gain factor was even larger). We have classified the task sets into groups that have a gain factor within a range of $G_f \pm 0.05$ and the results for a gain factor of $G_f = 0.1$ and $G_f = 0.3$ are shown in Figure 3.

Figure 3 compares the energy consumption of the DS and the SBS algorithm normalized to the OCS algorithm. The utilization of the task set is shown along the X-axis and the normalized energy consumption along the Y-axis. We show that the DS algorithm only perform marginally better than the OCS method. This is due to the fact that task blocking is frequent under non-preemptive scheduling, which results in tasks executing at the high speed H for most of the time. Note that the dual speed algorithm does not use the optimal feasibility test to compute the H speed. This can lead to a higher H speed than the OCS algorithm and result in higher energy consumption than OCS method, as seen at $U = 90\%$ ($G_f = 0.1$). On the other hand, the SBS algorithm uses the optimal feasibility test and always consumes less energy than the OCS algorithm. Furthermore, the system only switches to higher speeds when needed, as opposed to the DS algorithm which switches to the high speed (H) whenever a task is blocked. Thus the system manages to remain in the lower speed for a longer duration under the SBS algorithm, which leads to higher energy savings.

With a decrease in the utilization, the energy savings are seen to decrease with slight irregularities (rise and falls) in the energy savings. Note that the relative difference in energy consumption at $\overline{\eta}$ and η_{max} is lower at smaller slowdown factors (lower utilization). This is an inherent power consumption characteristic and leads to a decrease in energy savings at lower utilization. The irregularity arises from the grouping of tasks with a gain factor within a range of ± 0.5. Furthermore, the mapping of continuous slowdown factors to discrete levels is also another cause. We show that the SBS algorithm performs better than the DS algorithm and results on an average 15% energy savings over the DS algorithm. The energy savings are seen to increase with higher gain factors as shown for $G_f = 0.1$ and $G_f = 0.3$. The higher the gain factor, the higher is the difference in the base speed ($\overline{\eta}$) and the maximum speed, and the energy savings of executing a task at the base speed are higher.

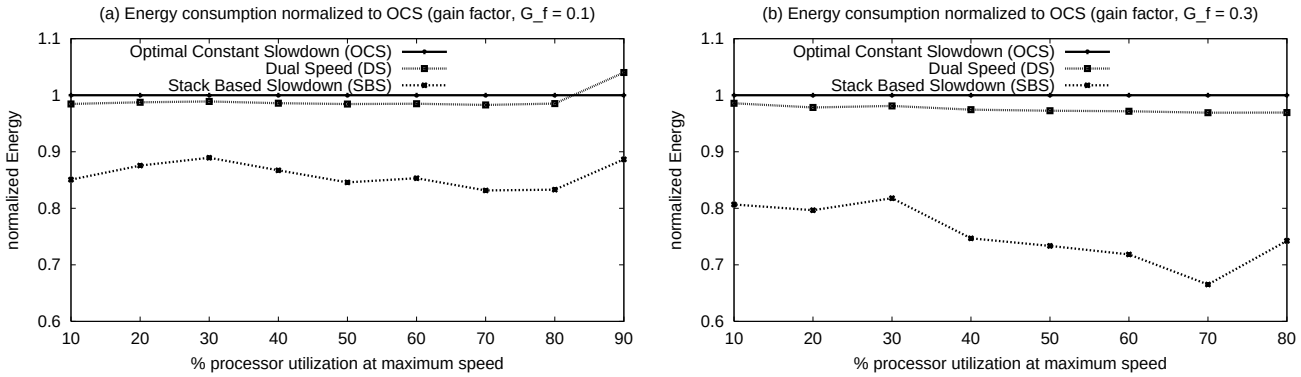


Figure 3. Energy consumption based on static slowdown factors, for gain factors, $G_f = 0.1$ and $G_f = 0.3$.

5.2 Dynamic slack reclamation

We now compare the additional energy gains achieved through dynamic slack reclamation techniques. To generate varying execution times, we vary the *best case execution time (BCET)* of a task as a percentage of its WCET. The execution times are generated by a Gaussian distribution with mean $\mu = (WCET + BCET)/2$ and a standard deviation, $\sigma = (WCET - BCET)/6$. The BCET of the task is varied from 100% to 10% in steps of 10%. Experiments were performed on various task sets and Fig. 4 shows the energy gains as BCET is varied at gain factors of $G_f = 0.1$ and $G_f = 0.2$ (with a task utilization of 70% - 80%). The variation of BCET is shown along the X-axis and the energy consumption is along the Y-axis. The energy consumption is normalized to the SBS algorithm (no slack reclamation). We compare the following schemes:

- Stack Based Slowdown (SBS) algorithm.
- Stack Based Slowdown with Dynamic Reclamation (SBS-DR) algorithm.
- Dual Speed Dynamic Reclamation (DSDR) [23] algorithm

Note that dynamic slack reclamation leads to energy savings even under worst case execution time, or BCET of 100%. This is because slack also arises from (1) executing tasks at a speed higher than the base speed and (2) mapping tasks to discrete voltage levels. A decrease in the BCET increases the slack in the system and we see a steady decrease in the energy consumption. SBS-DR performs slack reclamation (even under task blocking) to result on an average 20% energy gains over the SBS algorithm. On the other hand, the DSDR algorithm results in significantly higher energy consumption (even higher than SBS scheme - no slack reclamation). This is because of the frequent transitions to the higher speed and also due to the fact that no slack reclamation is performed when tasks are blocked in the system.

Comparing the energy savings at $G_f = 0.1$ and $G_f = 0.3$, we see that the relative energy savings reduce at higher values of gain factor. Larger gain factors result in higher η_{max} and executing tasks at higher speed results in relatively higher energy consumption. Though dynamic slowdown reduces the energy consumption, it can increase the number of transitions to higher speed and the relative energy savings are seen to reduce at higher values of gain factor (Fig. 4 (a) and 4 (b)).

6 Conclusions and Future Work

We have presented energy aware scheduling algorithms for non-preemptive systems. The techniques are important in systems where task preemption is impossible or prohibitively expensive (such as ultra-low power sensor network nodes). Compared to preemptive scheduling, a higher speed may be necessary under non-preemptive scheduling. Identifying the time intervals when a higher speed is necessary is important to reduce the energy consumption. The stack based slowdown (SBS) algorithm minimizes transitions to a higher speed and increases the energy efficiency of the system. Simulation results show on an average 15% savings in energy consumption when scheduling with static slowdown factors. Our dynamic slack reclamation technique enables slowdown of blocking tasks and result in an additional 20% increase in energy efficiency. We plan to apply these scheduling techniques to communication subsystems and extend it to a distributed scheduling policy.

Acknowledgments

We acknowledge support from National Science Foundation (Award CCR-0098335) and from Semiconductor Research Corporation (Contract 2001-HJ-899).

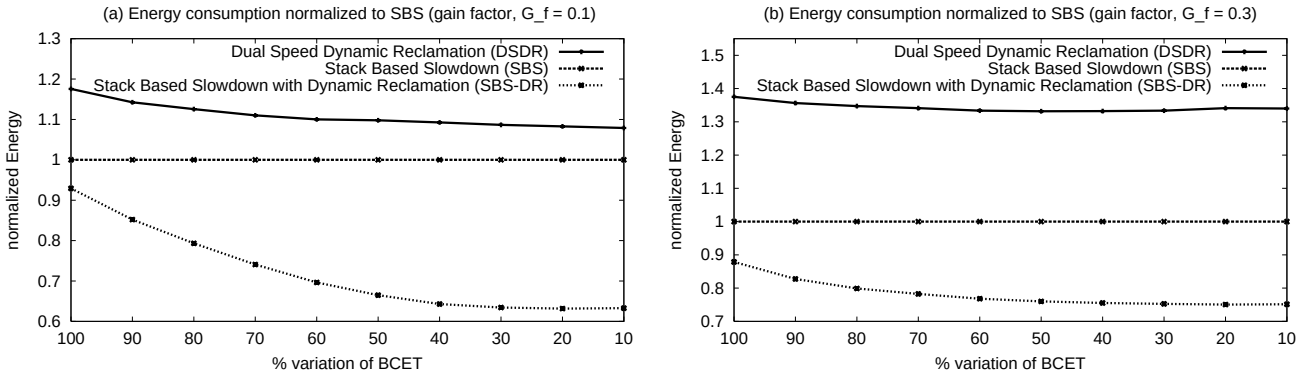


Figure 4. Energy consumption with dynamic slack reclamation, for gain factors, $G_f = 0.1$ and $G_f = 0.3$.

References

[1] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proc. of EuroMicro Conference on Real-Time Systems*, Jun. 2001.

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

[3] S. Dolev and A. Keizelman. Non-preemptive real-time scheduling of multimedia tasks. *Journal of Real-Time Systems*, 17(1):23–39, 1999.

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

[5] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable-voltage core-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1702–14, 1999.

[6] Intel XScale Processor. Intel Inc. *(http://developer.intel.com/design/intelxscale).*

[7] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processor. In *International Symposium on Low Power Eletronics and Design*, pages 197–202, Aug. 1998.

[8] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of IEEE Real-Time Systems Symposium*, pages 129–139, Dec. 1991.

[9] R. Jejurikar and R. Gupta. Energy aware non preemptive scheduling in hard real-time systems. In *CECS Technical Report #05-xx, UC Irvine*, Mar. 2005.

[10] W. Kim, J. Kim, and S. L. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of Design Automation and Test in Europe*, pages 788–794, Mar. 2002.

[11] W. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. In *Proceedings of the Design Automation Conference*, pages 125–130, 2003.

[12] J. Liu, P. H. Chou, and N. Bagherzadeh. Communication speed selection for embedded systems with networked voltage-scalable processors. In *Proceedings pf International Symposium on Hardware/Software Codesign*, Nov. 2002.

[13] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[14] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of Symposium on Operating Systems Principles*, 2001.

[15] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processors. In *Proc. of Design Automation and Test in Europe*, pages 782–87, 2002.

[16] C. Rusu, R. Melhem, and D. Mosse. Maximizing rewards for real-time applications with energy constraints. *ACM Transactions on Embedded Computer Systems*, 2(4):537–559, Nov. 2003.

[17] C. Rusu, R. Melhem, and D. Mosse. Multi-version scheduling in rechargeable energy-aware real-time systems. In *Proceedings of EuroMicro Conference on Real-Time Systems*, pages 95–104, 2003.

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

[19] Transmeta Crusoe Processor. Transmeta Inc. *(http://www.transmeta.com/technology).*

[20] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, 1993.

[21] F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proc. of IEEE Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[22] H. Yun and J. Kim. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *Trans. on Embedded Computing Sys.*, 2(3):393–430, 2003.

[23] F. Zhang and S. T. Chanson. Processor voltage scheduling for real-time tasks with non-preemptible sections. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 235–245, Dec. 2002.

[24] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the Design Automation Conference*, pages 183–188, 2002.