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

Ravindra Jejurikar Rajesh K. Gupta

Center for Embedded Computer Systems,
Department of Information and Computer Science,
University of California at Irvine,
Irvine, CA 92697
E-mail: `jezz@ics.uci.edu, gupta@cs.ucsd.edu`

CECS Technical Report #04-01
Jan 04, 2004

Abstract

Techniques like dynamic voltage scaling (DVS) and modulation scaling provide the ability to perform an energy-delay tradeoff in the computation and communications subsystems. Slowdown based on performance requirements has shown to be energy efficient while meeting timing requirements. We address the problem of computing slowdown factors for a non-preemptive task system based on the Earliest Deadline First scheduling policy. We present a stack based slowdown *algorithm based on the optimal feasibility test for non-preemptive systems. We also propose a dynamic slack reclamation policy to further enhance the energy savings. The algorithms are practically fast, and have the same time complexity as the feasibility test for non-preemptive systems. The simulation results for our test examples show on an average* 15% *energy gains using static slowdown factors and* 20% *gains with dynamic slowdown over the known slowdown techniques.*

Contents

List of Figures

List of Tables

1 Introduction

The concept of a task that is invoked periodically is central to a real time system. 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. In preemptive scheduling, the currently running task is preempted on the arrival of a higher priority task, whereas in non-preemptive scheduling, a new task is scheduled only on the completion of the current task execution. Though preemptive scheduling can guarantee a higher system utilization, there are scenarios where the properties of the device hardware and software make preemption either impossible or prohibitively expensive. For example, in packet-switched communication systems, a preemption requires the retransmission of the preempted packet. Non-preemptive scheduling is employed in such packet-switched networks, both wired [10, 11, 40] and wireless [30]. Scheduling over a shared media such as LAN, WLAN and buses is inherently non-preemptive, because each node in the network has to ensure that the shared channel is free before it can begin transmission. To further underline the importance of non-preemptive systems, scheduling of information flow for process control systems [2], [7], known as Fieldbuses [9], and automotive industry communication standards such as CAN [6], are based on non-preemptive scheduling. Along with its extensive use in communications system, non-preemptive processor scheduling is also used in light weight multi-tasking kernels and beneficial in multimedia applications [8]. Non-preemptive scheduling for real-time embedded systems has its benefits of accurate response time analysis, ease of implementation, no synchronization overhead and reduced stack memory requirements.

With the increase in computation and communication in portable devices, that operate on a limited battery supply, power is an increasingly important aspect in the design and operation of a system. Energy efficient scheduling techniques have shown to minimize the run-time energy consumption of the system [32], [4], [38], [31], [30]. To explain how energy efficiency can be achieved, we consider the example of a processor and then generalize it to other resources. The dynamic power consumption, P, for CMOS circuits [35], depends on the operating voltage and frequency of the processor and 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. Equation 1 shows the quadratic relationship between power and voltage. Thus a decrease in the supply voltage decreases the power consumption of the processor. 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 the frequency and voltage [35], resulting in a linear increases in the execution time of a task. Due to the quadratic decrease in power with voltage, and only a linear decrease in frequency, the energy consumption per unit work decreases with voltage at the cost of increased execution time. This technique of lowering the operating voltage, termed as *Dynamic Voltage Scaling (DVS)*, can be used to exploit energy-delay tradeoff for energy minimization. Recent processors such as the Intel XScale [14] and Transmeta Crusoe [33] support variable frequency and voltage levels, which can be varied at run-time.

Similar to DVS of a processor, wired communication links also support DVS which enables a energy-delay tradeoff in communications system. *DVS capability* has been shown for both serial I/O communication [17] and parallel I/O communication [34]. A lower voltage and frequency decreases the energy per bit at the cost of lowering the data throughput. Energy minimization using DVS techniques over communication links has been shown in the context of interconnect networks [31]. For the wireless communication systems, *modulation scaling* has been proposed by Schurgers *et al.* [29] which enables a similar energy-delay tradeoff. Modulation schemes are used in wireless communication where symbols are transmitted at a particular rate and each symbol encodes a particular number of bits, b. Decreasing the number the bits per symbol, decreases the transmission power per symbol at the cost of transmitting more symbols to send the same amount of data. This enables saving energy while increasing the data transmission time. Modulation scaling techniques have also been shown to minimize the run-time energy of communication systems [30].

In this paper, we focus on operating system level scheduling of non-preemptive tasks for hard real time systems with the goal of minimizing the energy consumption while meeting all task deadlines. In real-time systems, timing guarantees are of foremost importance and all timing requirements have to be ensured while performing a energy-delay tradeoff. Thus we have to judiciously decide the extent of slowdown while achieving our goal of minimizing energy. We propose a stack based slowdown algorithm to minimize the energy consumption, while guaranteeing all task deadlines. We also present a dynamic slack reclamation policy that works in conjunction with the static slowdown for additional energy gains. The scheduling techniques are general enough so that they can be used for non-preemptive scheduling of any resource that has the capability of slowdown. In our system model, tasks arrive periodically and have to complete by a specified deadline. Our scheduling policy is implemented as a centralized power manager, which receives all task requests and schedules them in a non-preemptive manner. Our simulation experiments show on an average 15% gains over the known slowdown technique, using static slowdown factors and the dynamic reclamation increases the gains to 20%.

The rest of the paper is organized as follows: The related work and the problem formulation is discussed in Section 2 and Section **??**. In Section 3, we present algorithms to compute optimum static slowdown factors for minimum energy consumption under the EDF scheduling policy. This is followed by an algorithm for dynamic slack reclamation in Section 4. The experimental results are given in Section 5. Finally, Section 6 concludes the paper with future directions.

2 Related Work

Previous work on energy aware scheduling mainly focuses on preemptive scheduling, which is commonly used in processor scheduling. Among the earliest works, Yao *et al.* [36] presented a 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 [25, 37] and scheduling over a fixed number of voltage levels [19]. The problem of scheduling for real-time task sets for energy efficiency has also been addressed. Real-time schedulability analysis has been used in previous works to compute static slowdown factors for the tasks [32], [12]. Aydin *et al.* [4] address the problem of minimizing energy, considering the task power characteristics. Dynamic slowdown techniques have been proposed to further increase the energy gains achieved by static slowdown [24], [5], [18]. The problem of maximizing the system value for a specified energy budget, as opposed to minimizing the total energy, is addressed in [26, 27, 28], [1].

Non-preemptive scheduling has been addressed in the context of multi-processor scheduling. Scheduling periodic and aperiodic task graphs which capture the computation and communication in a system is considered in [23],[13]. Zhang *et al.* [39] 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 [21], [20].

Energy aware scheduling has not been addressed in the context of non-preemptive uniprocessor scheduling. Prior work on scheduling with task synchronization [16] and non-preemptive sections [38] can be extended to handle non-preemptive scheduling. Zhang *et al.* present a dual speed (DS) algorithm [38] where the system executes at two speeds, a low speed, L, and a high speed, H. The authors also propose a dual speed dynamic reclamation (DSDR) algorithm which reclaims the run-time slack for further energy gains. The dual speed algorithm uses two speed to better exploit the slack based on run-time conditions, however it has some limitations. The high speed, H, is computed based on a sufficient feasibility test and is not optimal. This can result in using a H speed, greater than required and in certain cases a feasible task set can be declared as infeasible. Also, the dual speed algorithm uses the high speed whenever any task is blocked, which may not be needed during all task blockings. Feasibility conditions for non-preemptive scheduling of real time task sets have been studied and optimal tests have been proposed [15, 40]. However, using the constant speed computed by an optimal feasibility test, as the high speed (H) in the dual speed algorithm, need not imply feasibility. To overcome these limitations, we propose a static slowdown algorithm called the *stack based slowdown* algorithm for energy efficient scheduling of non-preemptive tasks. The algorithm can compute more than two speeds as opposed to the dual speed algorithm. We show that dynamic slack reclamation techniques presented in previous work [5, 38] cannot be used for scheduling of non-preemptive tasks. We enhance our stack based slowdown algorithm with dynamic reclamation technique for added energy gains. We compare our slowdown algorithm to the dual speed and DSDR algorithm, which are the best known slowdown algorithms that can be applied to non-preemptive scheduling.

3 Static Slowdown Factors

In this section, we present the computation of static slowdown factors for tasks that are non-preemptively scheduled by the earliest deadline first (EDF) scheduling policy.

3.1 Constant Static Slowdown

Feasibility condition for non-preemptive tasks are well studied [15]. The *optimal* feasibility condition for non-preemptive tasks, with no inserted idle intervals, is as given by Theorem 1.

Theorem 1 *[15] 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 the feasibility need not be checked for all time instances up to a task deadline. It suffices to check the feasibility at the scheduling points for each task τ_i, which are given by $S_i = \{kT_j | j = 1, \ldots, i; k = 1, \ldots, \lfloor \frac{T_i}{T_j} \rfloor\}$. The feasibility test leads to a constant static slowdown factor, which is given by Theorem 2.

Theorem 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, *if*

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

$$\forall_i, 1 < i \leq n; S_{ij} \in S_i : \quad \frac{1}{\eta}(C_i + \sum_{k=1}^{i-1} \lfloor \frac{S_{ij}}{T_k} \rfloor C_k) \leq S_{ij} \tag{5}$$

The correctness of the theorem follows directly from Theorem 1. Theorem 2 gives an algorithm to compute a constant slowdown for all tasks in the system while maintaining feasibility.

Before we present the algorithm, we present the results for preemptive scheduling policy.

Theorem 3 *For a preemptive task model based on EDF scheduling policy, a task set is feasible at a slowdown of* η, *if*

$$\frac{1}{\eta} \sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1 \tag{6}$$

EDF scheduling is optimal, and as can be seen from Theorem 3, the system utilization is a lower bound on the constant static slowdown.

3.2 Dynamic Slowdown

Theorem 2 computed a constant slowdown and the system can be under-utilized at this slowdown, resulting in idle intervals. Based on the run time conditions, we can further exploit the idle intervals to result in further energy gains. It is known that the preemptive EDF scheduling is the optimal scheduling policy [22] and can lead to a higher system utilization. This is also evident from the fact that the feasibility conditions for non-preemptive scheduling (Theorem 1) has additional constraints than that for preemptive scheduling (Theorem 3). The additional constraints in non-preemptive scheduling can result in a lower utilization (higher constant slowdown) than a preemptive system. A minimum slowdown satisfying Theorem 3 is the optimal slowdown for preemptive task system [3] and is a lower bound on the constant slowdown. In our dynamic slowdown algorithm, we set this slowdown as the base speed and tasks do not execute at a speed lower than this speed (in the absence of dynamic slack reclamation). We call this slowdown as the *base speed* $\bar{\eta}$ and it satisfies the following constraint.

$$\frac{1}{\bar{\eta}} \sum_{k=1}^{n} \frac{C_k}{T_k} \leq 1 \tag{7}$$

If tasks do not block other higher priority tasks, they can execute at a slowdown equal to the base speed without missing any deadlines. If a higher priority task is blocked due to the non-preemptive nature of the system, the system speed is increased if necessary to ensure deadlines of all higher priority tasks in the system. We calculate a speed η_i for each task τ_i which guarantees meeting all higher priority task deadlines. For each task, a slowdown η_i is computed which satisfies the following conditions.

$$\forall_j S_{ij} \in S_i; \frac{1}{\eta_i}(C_i + \sum_{k=1}^{i-1} \lfloor \frac{S_{ij}}{T_k} \rfloor C_k) \leq S_{ij} \tag{8}$$

where $S_i = \{kT_j | j = 1, ..., i; k = 1, ..., \lfloor \frac{T_i}{T_j} \rfloor\}$ is the set of scheduling points for task τ_i. If a task τ_i blocks a higher priority task, the current system slowdown is compared to η_i. If η_i is greater than the current speed, then we set the processor speed to η_i. All tasks with a priority higher than job τ_i execute at a speed of at least 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 τ_i or if the system becomes idle. This resembles a *stack* operation and, indeed, we use a stack to implement the algorithm.

3.3 Stack Based Slowdown (SBS) Algorithm

The SBS algorithm makes use of variable processor speeds. The current processor speed is maintained in a stack, S. Each stack node, sn, has a slowdown(η) and a priority(P) associated with it and is 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 algorithm is given in Figure 1. The stack is initialized with a slowdown factor equal to the *base speed* ($\bar{\eta}$) 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 never popped off the stack. The processor speed is always set to the slowdown factor at the top of the stack. Thus the system begins executing jobs at the base speed, similar to a preemptive scheduling policy. At all times, let sn_t represent the node at the top of the stack and let τ_c be the current job executing in the system. While task τ_c is executing, if a task τ_i with a priority higher than that of task τ_c arrives, the higher priority task τ_i is blocked due to non-preemption. If a current task is blocking a higher priority task and the slowdown factor η_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 τ_c. In that case, a node $sn(\eta_c, P(\tau_c))$ is pushed on the stack. All jobs with priority lower than that of τ_c are executed at a minimum slowdown of η_c to guarantee the feasibility of higher priority tasks. A stack top node (sn_t) is popped off the stack when a job with a priority lower that the stack top priority ($P(sn_t)$) is executed. Since the 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. We prove that the stack based slowdown algorithm ensures all task 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 algorithm at a base speed $\bar{\eta}$ and task slowdown factors η_i iff,*

$$\frac{1}{\bar{\eta}}(\sum_{k=1}^{n} \frac{C_k}{T_k}) \leq 1 \tag{9}$$

$$\forall_j S_{ij} \in S_i; \frac{1}{\eta_i}(C_i + \sum_{k=1}^{i-1} \lfloor \frac{S_{ij}}{T_k} \rfloor C_k) \leq S_{ij} \tag{10}$$

where S_i represents the set of scheduling points for task τ_i.

```
    Notation :
        τ_c  :  the current task executing in the system
        sn_t :  the node at the top of the stack

    Stack Initialization :
(1)  Given an initially empty stack,
     Push base node $sn(\eta, -\infty)$ on the stack;

    On arrival of task $\tau_i$ in the system:
(1) if (processor NOT idle before task arrival and
              $P(\tau_i) > P(\tau_c)$ and $\eta_c > \eta(sn_t)$)
(2)    Push $sn(\eta_c, P(\tau_c))$ on the stack;
(3)    $SetSpeed(\eta_c)$;
(4) endif

    On execution of each task $\tau_i$ :
(1) if ($P(\tau_i) < P(sn_t)$)
(2)    while ($P(\tau_i) < P(sn_t)$)
(3)        Pop $sn_t$ from the stack; // Pop the stack top node;
(4)    end // End while loop
(5)    $SetSpeed(\eta(sn_t))$;
(6) endif

    On system idle:
(1) Pop all nodes from stack, except base node;
```

Figure 1. Stack Based Algorithm for non-preemptive scheduling

Proof: Suppose the claim is false and a task instance misses its deadline. Let t be the first time that a job 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, the only jobs that are allowed to *start* execution in $[t', t]$ are in A. Also, there are pending requests of jobs in A at all times during the interval $[t', t]$ and the system in 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 Equation 9.

Case II: Let J_b be the job executing at time t' with a deadline greater than t, that blocks a job in A. 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 $D_k < X$ 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 $\frac{1}{\eta_b}(C_b + \sum_{i=1}^{k} \lfloor \frac{X}{T_i} \rfloor C_i)$. 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,

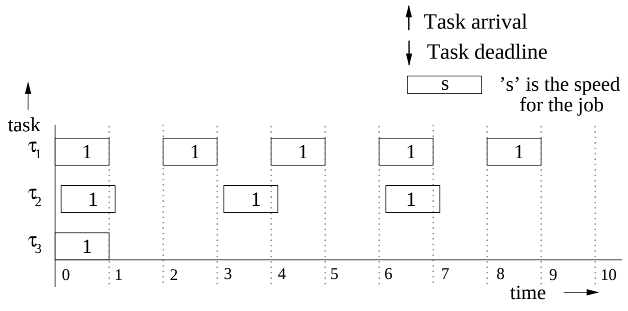
$$\frac{1}{\eta_k}(C_k + \sum_{i=1}^{k} \lfloor \frac{X}{T_i} \rfloor C_i) > X$$

Since $X < T_b$, this contradicts Equation 10. ■

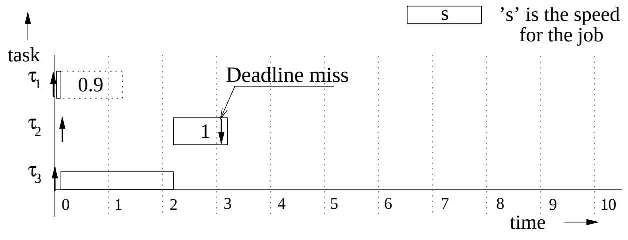
4 Dynamic Slack Reclamation

Dynamic slack arises due to early task completions and due to execution intervals at speed higher than the base speed. This slack can be reclaimed to further reduce the processor speed, resulting in increased energy gains. In this section, we present a dynamic slack reclamation scheme that that works in conjunction with the stack based slowdown algorithm and is referred to as the Stack Based Slowdown with Dynamic Reclamation (SBS-DR) algorithm. Before we describe our algorithm, we show that traditional dynamic slack reclamation approaches cannot ensure meeting all deadlines.

In traditional dynamic slack reclamation techniques, reclaiming the unused time budget (slack) of the higher priority tasks, ensures task deadlines. However, we show that performing the same with the stack based slowdown algorithm for non-preemptive task scheduling, can result in deadline miss. To illustrate this, we consider the same task set as shown in Section **??**, with tasks $\tau_1 = \{1,2,2\}, \tau_2 = \{1,3,3\}$ and $\tau_3 = \{1,15,15\}$. As described in Section 3, the computed task slowdown factors are $\eta_1 = 0.5$, $\eta_2 = 1.0$ and $\eta_3 = 1.0$, with the base speed, $\overline{\eta} = 0.9$. The time budget for each task is 1.11 at a slowdown $\eta = 0.9$ and at a slowdown $\eta = 1.0$ the time budget is 1.



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



(b) Deadline miss due to salck reclamation

Figure 2. (a) Task arrival times and deadlines (NOT a task schedule). (b) Feasible schedule, even if task τ_3 arrives just before other tasks and blocks the higher priority tasks.

We consider the case where task τ_1 and τ_3 arrive at time $t = 0$ and task τ_2 arriving at time $t = 0.1$. The task arrivals times are shown in Figure 2(a). The system begins execution at the base speed, $\overline{\eta} = 0.9$ and a budget of 1.11 is assigned to task τ_1. Task τ_1 completes in almost negligible time, at $t = 0.05$, leaving a free budget of 1.06. Since τ_3 is the only ready task, it begins execution. If tasks are allowed to reclaim the higher priority task budgets, then τ_3 reclaims the slack of τ_1. Task τ_2 arrives immediately when τ_3 begins execution, say at time $t = 0.06$, and has a deadline of $t = 3.06$. Since task τ_2 is blocked, $\eta_3 = 1.0$ is pushed on the stack and the budget of task τ_3 is set to 1.0. However, by reclaiming the free budget of task τ_1 along with its own budget, τ_3 completes at time $t = 1.11 + 1 = 2.11$. Task τ_2 executes at a speed of $\eta = 1$, using its own budget of 1 time unit, and finishes at time 3.11. However it has already missed its deadline of $t = 3.06$. Thus, we see that slack reclamation techniques from previous work cannot be used with the stack based slowdown for non-preemptive scheduling. We present slack reclamation techniques that work with the stack based slowdown algorithm.

4.1 Stack Based Slowdown with Dynamic Reclamation (SBS-DR)

The possible speeds (slowdown factors) of the nodes on the stack are the task slowdown factors greater than the base speed, $\{\eta_i | \eta_i > \bar{\eta}\}$. This is because a slowdown factor smaller than the base speed is never pushed on the stack. We say node sm_i dominates sm_j if $\eta(sm_i) > \eta(sm_j)$ or equivalently sm_j is dominated by sm_i. Since only higher slowdown factors are pushed on the stack, a node dominates all nodes below it in the stack. The slowdown factor of the node on the stack top determines the time budget for each task execution and hence the processor slowdown. We define *run time* of a job as the time budget assigned to the job considering the slowdown of the stack top node. The run time of a job with a workload C and slowdown η, is C/η. Each run time has a time value 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)* [38]. All FRT-lists are maintained sorted by priority of the run-times, with the highest priority at the head of the list and the lowest priority at the tail. Run-time is always consumed from the head of the list.

The dynamic slowdown works in conjunction with the stack based slowdown algorithm explained in the Section 3. For dynamic slack reclamation, in addition to a slowdown and a priority, each stack node also has a FRT-list associated with it, and a stack node sn is represented as $(\eta, P$, FRT-list). FRT-list(sn) is used to reference the FRT-list of the stack node sn. The dynamic slack arises from two sources: (1) If a job executes when the slowdown of the stack top node (sn_t) is greater than the base node slowdown factor, the job is assigned a budget of $\frac{C_i}{\eta(sn_t)}$. The remaining time budget that would be assigned if the task were executing at the base speed, is distributed among the nodes that are dominated by sn_t (stack nodes having a smaller slowdown than sn_t). (2) On job completion, its unused run time is added to the *FRT-list* of stack top node (sn_t) with the same priority as the job priority. The SBS-DR algorithm is given in Figure 3. Before the execution of a job, the algorithm reserves a run-time for the job based on the stack top slowdown factor when it begins execution. As shown in line (5) an instance of task τ_i is assigned a run-time of $C_i/\eta(sn_t)$, where sn_t is the stack top node. Furthermore, if the current node dominates other nodes in the stack, then for each dominated node sn_d, the difference in the budget arising from a slowdown of $\eta(sn_d)$ and the (adjacent) immediate dominating node slowdown $\eta(sn_D)$, is added to the FRT-list of node sn_d as shown in line 6 of the algorithm. A job can use its own run time as well as the free run-time from the stack top FRT-list, with a priority (the priority of the run-time) no smaller than the task priority. We prove that tasks can use this slack while guaranteeing all deadlines. The available budget for a task decides the processor speed.

While executing a task τ_c, if a new task arrives, with a higher priority than the current job and η_c is greater than the stack top slowdown, the speed is increased to η_c and the job completes execution at this speed. When a node $(\eta, P$, FRT-list) is pushed on the stack, the FRT-list of this newly added node is initially *empty*. This stack node is popped from the stack on the execution of a job whose priority is lower than that of task τ_c. When a stack node is popped, the FRT-list of this popped node is concatenated to the FRT-list of the new (current) stack top node. When a job completes execution, the unused slack (run-time) of the task is added to the FRT-list of the current node at the stack top . If the system becomes idle, all nodes except the base node are popped and the FRT-lists of all the popped nodes are added to the FRT-list of the base node.

We use similar notation and definitions used in [38, 16] 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.
- $R_i^F(t)$: 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 in run-time between node sm_d and its *immediately dominating* (adjacent) node, sm_D, on a stack. If η_D and η_d be the slowdown in modes sm_D and sm_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 between the two modes.

The dynamic slowdown factor is the ratio of the residual workload to the available runtime.
The following rules are used in SBS-DR algorithm.

- As job J_i executes, it consumes run time at the same speed as the wall clock (physical time) [38]. If $R_i^F(t) > 0$, the run time is used from the stack top FRT-list, else $R_i^r(t)$ is used.
- When the system is idle, it uses the run time from the base node FRT-list if the list is non-empty.

Note that the rules need to be applied only on the arrival of a task in the system and on task completion.

Theorem 5 *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 at a base speed $\bar{\eta}$ and task slowdown factors η_i iff,*

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

$$\forall_j S_{ij} \in S_i; \frac{1}{\eta_i}\left(C_i + \sum_{k=1}^{i-1} \left\lfloor \frac{S_{ij}}{T_k} \right\rfloor C_k\right) \leq S_{ij} \tag{12}$$

where S_i represents the set of scheduling points for task τ_i.

Proof: Suppose the claim is false and a task instance misses its deadline. Let t be the first time that a job misses its deadline and sn_c be the operating mode at time t. Let t' be the the latest time before t such that (1) there are no pending jobs with arrival times before t' and deadlines less than or equal to t, (2) the system is operating in mode sn_c at time t', and (3) mode sn_c slack is zero at time t'. Note that, t' is well defined since at system start ($time = 0$), there are no job requests, the operating mode with $\eta(sn) = \bar{\eta}$ and all slack lists are empty. Let A be the set of jobs that arrive in $[t', t]$ and have deadlines in $[t', t]$. Two cases arrive based on whether the system is operating in (1) the base mode with a slowdown of $\bar{\eta}$, or (2) a mode dominating the base mode at time t. We consider both cases separately.

Case I: The operating mode at time t is the base mode sn_b with a slowdown of $\eta(sn_b) = \bar{\eta}$. Let $Y = t - t'$. By definition of t', slack consumed during the interval Y is generated in this interval. The slack generated in the interval Y with deadline less than or equal to t is bounded by $\sum_{i=1}^{n} \lfloor \frac{Y}{T_i} \rfloor \frac{C_i}{\bar{\eta}}$. The slack consumed is bounded by Y. Since the task misses its deadline, the budget generated is greater than that consumed in the interval Y. Hence,

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

```
    Stack Initialization :
(1) Initialize stack with a base node ($U, -\infty$, FRT-list)

    On arrival of task $\tau_i$ in the system:
(1) if ( processor NOT idle before task arrival and
            $P(\tau_i) > P(\tau_c)$ and $\eta_c > \eta(sn_t)$)
(2)     Push sn($\eta_c, P(\tau_c)$, FRT-list) on the stack;
(3)     $SetSpeed(\eta_c)$;
(4) endif

    On execution of each task $\tau_i$ :
(1) while ($P(\tau_i) < P(sn_t)$)
(2)     $old\_sn_t \leftarrow$ Pop the stack top node;
(3)     Concatenate FRT-list($old\_sn_t$) to the FRT-list($sn_t$)
(4) end
(5) $R_i^r(t) = C_i/\eta(sn_t)$;
(6) For each dominated mode $sn_d$ on the stack :
        Add $R_i^M(d)$ to FRT-list($sn_d$);
(7) $setSpeed(\frac{C_i^r(t)}{R_i^r(t)+R_i^F(t)})$;

    On Completion of $\tau_i$:
(1) Add run-time $(R_i^r(t), P(\tau_i))$ to FRT-list($sn_t$);

    On System Idle:
(1) while ($stack.size > 1$) // Pop all nodes except base node
(2)     Concatenate FRT-list($sn_t$) to base node FRT-list;
(3)     Pop Stack Top;
(4) end
```

Figure 3. Stack Based Dynamic Reclamation Algorithm for non-preemptive scheduling

which implies

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

which contradicts Equation 11

Case II: The system is operating in a mode sn_c that dominates the base mode ($\eta(sn_c) > \bar{\eta}$). Consider the latest time before t when the system enter this mode. When the system enters the mode sn_c, it is executing a job with deadline greater than t. Furthermore, on entering the mode sn_c, the slack in the mode is zero. Hence all three properties are satisfied when the system enters mode sm_c, after which requests of jobs with deadline less than t are always pending and so t' is the time when the system enter the mode. The system is in a mode sn_c or a dominating mode, for the entire duration $[t', t]$. Let J_b be the job executing at time t' with a deadline greater than t. This job blocks the higher priority job that arrives at time t'. Let $Y = t' - t$. J_b executes at time t' with deadline greater than t and $Y < D_b$. If $A \subseteq \{\tau_1, ...\tau_k\}$, then $D_k < Y$ and $k < b$. The number of cycles of execution of job J_b in $[t', t]$ is bounded by its execution time C_i. Since the current job slowdown is pushed on the stack on blocking, $\eta(sn_c) = \eta_b$. Job J_b consumes a lower mode slack and this slack is bounded by $\frac{C_b}{\eta_b}$. Since only sn_c and higher mode slack is consumed during the entire interval, the total sn_c or higher mode budget generated in the interval Y is bounded by $\sum_{k=1}^{i-1}\lfloor\frac{Y}{T_i}\rfloor\frac{C_k}{\eta_b}$. The budget available to be consumed is bounded by $\frac{1}{\eta_b}(C_b + \sum_{k=1}^{i-1}(\lfloor\frac{Y}{T_k}\rfloor)C_k$. The budget consumed during this time period is Y. Since a task misses its deadline at time t, the budget generated after t' for the jobs in A and that of job J_b exceeds Y, the length of the interval. Therefore,

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

Since $Y < T_b$, this contradicts Equation 12. Hence all tasks meet the deadline. ■

5 Experimental Setup

To evaluate the effectiveness of our energy aware non-preemptive scheduling algorithms, we consider several task sets with randomly generated tasks. A mixed workload with task periods belonging to one of the two period ranges [1000,2000] and [4000,5000]. The Worst Case Execution Times (WCET) for the corresponding period ranges were [200,400] and [200,800]. The tasks were uniformly distributed in these categories with the period and WCET of a task randomly selected within the corresponding range.

Note that our scheduling techniques can be applied to both real-time computation as well as communication systems [40, 30]. For experimental purposes, we consider the case of processor scheduling. We use the power model for CMOS circuits as given in Equation 1. The details of the power model are given in [35]. The operating voltage range for the processor is $0.6V$ and $1.8V$, which is the trend in current embedded processors. We have normalized 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 gains of the following techniques:

- Optimal Constant Slowdown (OCS) algorithm (Theorem 2)

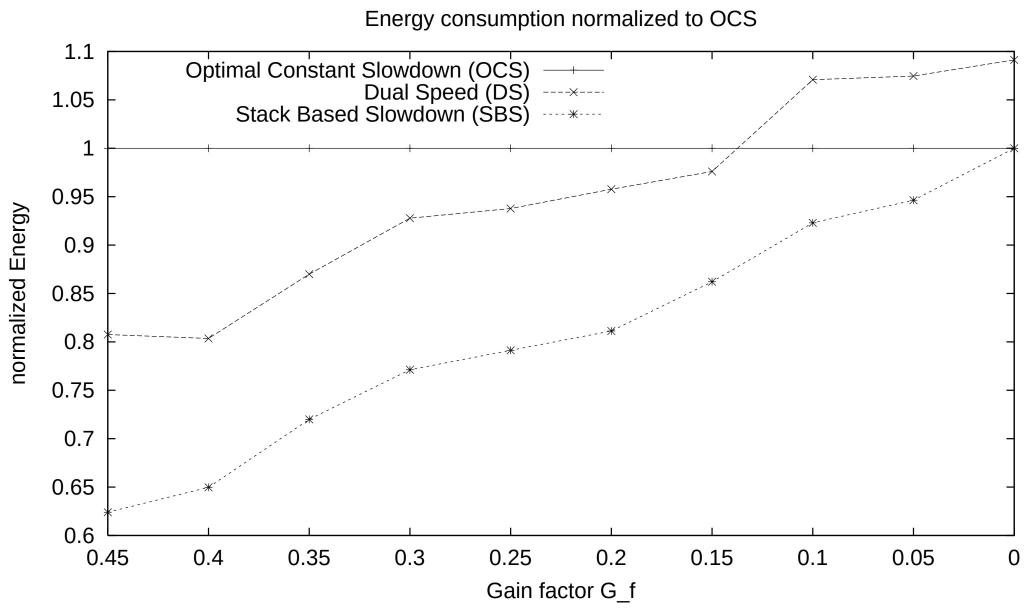


Figure 4. Energy comparison of the static slowdown algorithms as a function of the task gain factor (G_f).

- Dual Speed (DS) algorithm [38]
- Stack Based Slowdown (SBS) algorithm, proposed here.

Note that, with no dynamic slack reclamation, the base speed (equal to the system utilization U), $\bar{\eta} = U$, is the lower bound on the task slowdown factor. The slowdown computed by the OCS algorithm, say η_{max}, guarantees feasibility and this is the upper bound on the task slowdown factor. SBS algorithm uses the optimal feasibility test and ensures that the slowdown is always less than or equal to η_{max}. The difference in the speeds between $\bar{\eta}$ and η_{max} is the reason for the energy gains in DS and SBS, compared to OCS algorithm. To capture this relationship, we define the *gain factor (G_f)* of a task set as the difference between the ratio of the lower ($\bar{\eta}$) and upper (η_{max}) bounds on the constant slowdown and unity, $G_f = 1 - \frac{\bar{\eta}}{\eta_{max}}$. The lower the ratio of lower and upper bounds, higher is the gain factor which is a representative of the energy gains. The gain factor represents a fraction by which the energy consumption can be lowered. Figure 4 compares the energy consumption of the DS and the SBS methods, normalized to the OCS algorithm. The execution time for each task is its WCET at maximum speed. The energy gains for the various task-sets are shown with the gain factor along the X-axis and the normalized energy consumption along the Y-axis. It is seen that the energy gains of DS and SBS is proportional to the gain factor. Since the gains are proportional to the gain factor, we have averaged the energy gains over task-sets with a gain factor within a range of 0.05. The larger the gain factor, the larger is the difference in the base speed and the maximum speed, and the energy gains by executing a task at the base speed ($\bar{\eta}$) are larger. The energy consumption decreases steadily as the gain factor increases, with both DS and SBS having increased energy gains over the OCS algorithm.

For gain factors close to 0, it is seen that the DS algorithm consumes more energy than OCS. Since the DS algorithm does not use the optimal feasibility test, the H speed in the dual speed algorithm can be greater than η_{max}. This result in more energy consumption as opposed to the the savings achieved by using a lower speed (base speed) during certain time intervals. The SBS algorithm uses the optimal feasibility test and always consumes less energy than the OCS algorithm. Since SBS uses the optimal feasibility test and only switches to higher speeds when needed, it results in energy gains over the DS algorithm. It is seen that the SBS algorithm on an average has 15% energy gains over the DS algorithm.

5.2 Dynamic slack reclamation

We now compare the added energy gains that are achieved by dynamic slack reclamation techniques. The dynamic slack reclamation techniques compared are as follows:

- Optimal Constant Slowdown with Dynamic Reclamation (OCS-DR) algorithm, where we use the same the dynamic slack reclamation technique discussed in Section 4, over the slowdown factors computed by the OCS algorithm.
- Dual Speed Dynamic Reclamation (DSDR) algorithm [38]
- Stack Based Slowdown with Dynamic Reclamation (SBS-DR) algorithm, proposed here.

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 Figure 5 shows the energy gains as BCET is varied, at two gain factors, $G_f = 0.1$ and $G_f = 0.4$.

Dynamic reclamation leads to energy gains even at worst case execution times, or BCET of 100%. This is because the slack arising due to executing at speeds higher than the base speed are reclaimed during reclamation techniques. Secondly, mapping tasks to discrete voltage levels also adds to the system slack, resulting in energy gains even at BCET of 100%. A steady decrease in the energy consumption is seen with a decrease in the BCET. As the task execution time is decreased, there is shorter blocking intervals and fewer transitions to higher speeds, leading in increased energy gains compared to OCS-DR. The DSDR and SBS-DR follow the same trend with the variation of BCET. However, SBS-DR uses better static slowdown factors by using a optimal feasibility test and results in more energy savings. SBS-DR results on an average 20% energy gains over the DSDR algorithm. The same is true in both graphs in Figure 5. Comparing the energy savings at $G_f = 0.1$ and $G_f = 0.4$, it is seen that the larger the gain factor, the higher are the energy gains. This is because the slack reclamation techniques also reclaim the slack that arises by the tasks execution in a higher speed than the base speed. The energy consumption is seen to decrease up to 65% for $G_f = 0.1$ and as low as 58% for $G_f = 0.4$.

6 Conclusions and Future Work

We have presented energy aware scheduling techniques for non-preemptive task sets. These task sets are important in systems where task preemption is impossible (e.g., modulation scaling applied to wireless communications) or prohibitively expensive (such as ultra-low power sensor network nodes). We propose the stack based slowdown algorithm based on the optimal feasibility test and present a dynamic slack reclamation algorithm that works in conjunction with it. We see that designing slowdown algorithms based on optimal feasibility tests results in higher energy efficiency. Experimental results show on

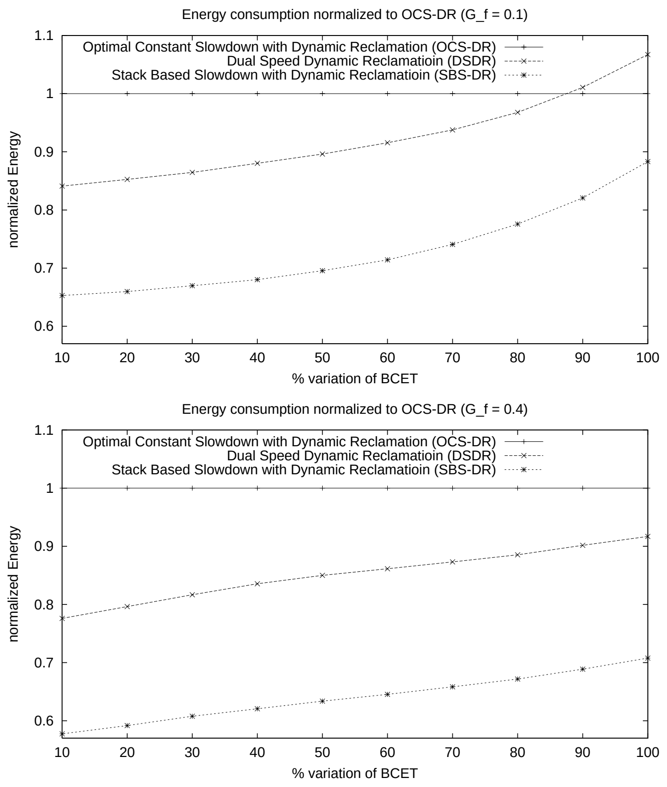


Figure 5. Energy Consumption with Dynamic Slack Reclamation, for gain factors, $G_f = 0.1$ and $G_f = 0.4$.

an average 15% gains when scheduling with static slowdown factors and 20% gains with dynamic slack reclamation. These techniques are energy efficient and can be applied to energy efficient scheduling of communications systems as well. This will lead to energy efficient computation and communication systems and will have a great impact on the energy utilization of portable devices.

Our algorithm is based on a centralized scheduling policy. We plan to extend it to a distributed scheduling policy which will extend the applicability to more systems.

References

[1] T. A. AlEnawy and H. Aydin. Energy-constrained performance optimizations for real-time operating systems. In *Proceedings of the Workshop on Compilers and Operating System for Low Power*, 2003.

[2] L. Almeida and J. A. Fonseca. Analysis of a simple model for non-preemptive blocking-free scheduling. In *EuroMicro Conference on Real-Time Systems*, pages 233–, 2001.

[3] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Real-Time Systems Symposium*, Phoenix, AZ, Dec 1999.

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

[5] H. Aydin, R. Melhem, D. Mossé, and P. M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Real-Time Systems Symposium*, December 2001.

[6] CAN-CIA. CAN specification 2.0 Part B, 1992. http://www.can-cia.org/downloads/ciaspecifications.

[7] S. Cavalieri, A. Corsaro, O. Mirabella, and G. Scapellato. Scheduling periodic information flow in fieldbus and multi-fieldbus environments. In *The International Conference on Automation 1998 Proceeding, Milano, Italy*, 1998.

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

[9] EN 50170. General purpose field communication system. In *European Standard, CENELEC*, July 1996.

[10] D. Ferrari and D. Verma. Real-time communication in a packet-switching network. In *Proc. Second IFIP WG6.1/WG6.4 Intl. Workshop on Protocols for High-Speed Networks*, Palo Alto, Calif., 1990.

[11] D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, 1990.

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

[13] F. Gruian and K. Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the Asia South Pacific Design Automation Conference*, 2001.

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

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

[16] R. Jejurikar and R. Gupta. Dual mode algorithm for energy aware fixed priority scheduling with task synchronization. In *Proceedings of the Workshop on Compilers and Operating System for Low Power*, 2003.

[17] J. Kim and M. Horowitz. Adaptive supply serial links with sub-1v operation and per-pin clock recovery. In *Proceedings of International Solid-State Circuits Conference*, Feb 2002.

[18] 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 *DATE*, 2002.

[19] W. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. In *Proceedings of the 40th conference on Design automation*, pages 125–130, 2003.

[20] J. Liu and P. H. Chou. Energy optimization of distributed embedded processors by combined data compression and functional partitioning. In *International Conference on Computer Aided Design*, Nov. 2003.

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

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

[23] J. Luo and N. Jha. Power-conscious joint scheduling of periodic task graphs and a periodic tasks in distributed real-time embedded systems. In *International Conference on Computer Aided Design*, 2000.

[24] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of 18th Symposium on Operating Systems Principles*, 2001.

[25] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processors. In *Design Automation and Test in Europe*, pages 782–787, March 2002.

[26] C. Rusu, R. Melhem, and D. Mosse. Maximizing the system value while satisfying time and energy constraints. In *Real-Time Systems Symposium*, 2002.

[27] C. Rusu, R. Melhem, and D. Mosse. Multi-version scheduling in rechargeable energy-aware real-time systems. In *EuroMicro Conference on Real-Time Systems*, 2003.

[28] C. Rusu, R. Melhem, and D. Mosse. Maximizing rewards for real-time applications with energy constraints. In *ACM Transactions on Embedded Computer Systems*, accepted.

[29] C. Schurgers, O. Aberthorne, and M. B. Srivastava. Modulation scaling for energy aware communication systems. In *International Symposium on Low Power Electronics and Design*, pages 96–99, August 6-7, 2001.

[30] C. Schurgers, V. Raghunathan, and M. B. Srivastava. Modulation scaling for real-time energy aware packet scheduling. In *Global Communications Conference (GlobeCom'01), San Antonio, Texas*, pages 3653–3657, November 25-29, 2001.

[31] L. Shang, L. S. Peh, and N. K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, January 2003.

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

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

[34] G. Wei, J. Kim, D. Liu, and M. Horowitz. A variable frequency parallel io interface with variable frequency parallel i/o interface with adaptive power-supply regulation. *Journal of Solid-State Circuits*, 35(11):1600–1610, Nov 2000.

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

[36] F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the Foundations of Computer Science*, pages 374–382, 1995.

[37] 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.

[38] F. Zhang and S. T. Chanson. Processor voltage scheduling for real-time tasks with non-preemptible sections. In *Real-Time Systems Symposium*, 2002.

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

[40] Q. Zheng and K. G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, 42(2/3/4):1096–1105, Feb/Mar/Apr. 1994.