Leakage Aware Dynamic Slack Reclamation in Real-Time Embedded 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 #04-31

November 2004

Abstract

Leakage energy consumption is an increasing concern in current and future CMOS technology generations. Procrastination scheduling, where task execution can be delayed to maximize the duration of idle intervals, has been proposed to minimize leakage energy drain. We address dynamic slack reclamation techniques under procrastination scheduling to minimize the static and dynamic energy consumption. In addition to dynamic task slowdown, we propose dynamic procrastination which seeks to extend idle intervals through slack reclamation. While using the entire slack for either slowdown or procrastination need not be the most efficient approach, we distribute the slack between slowdown and procrastination to exploit maximum energy savings. Our simulation experiments show that dynamic slowdown results on an average 10% energy gains over static slowdown. Dynamic procrastination can extend the average sleep intervals by up to 70%, *while meeting all timing requirements.*

Contents

List of Figures

1 Introduction

Portable embedded systems are pervasive with applications in multimedia, telecommunications and consumer electronics. These systems are usually battery operated and power management is important in the design and operation of these systems. A processor is central to an embedded system and contributes to a significant portion of the total power consumption of the system. The two primary ways of reducing the processor power consumption are *shutdown* and *slowdown*. To understand the benefits of each technique, one needs to consider the two distinct contributors to device power consumption: (1) *dynamic* power consumption arising due to switching activity in a circuit and (2) *static* power consumption which is present even when no logic operations are performed. Slowdown (through dynamic voltage and frequency scaling) is known to reduce the dynamic power consumption at the cost of increased execution time for a given computation task. However, the increased computation time arising from slowdown results in increasing the static energy consumption. Shutdown, on the other hand, eliminates the static energy drain. With the increasing static power consumption (a result of increasing leakage currents), a combination of slowdown and shutdown techniques are important to minimize the total energy consumption of the system.

Most of the prior works have addressed processor slowdown to minimize the dynamic power consumption. Slowdown computation techniques can be broadly classified into: (1) *static* slowdown factors, computed using static analysis based on worst case task execution times and (2) *dynamic* slowdown factors, which utilize slack arising from varying task execution times to extend slowdown. Among the earliest works on this problem, Yao *et. al.* [14] presented an off-line algorithm to compute the optimal static slowdown (speed) schedule for a set of N jobs. This work has been later extended to consider realistic processor models and different scheduling policies [12, 10, 6]. Dynamic voltage scaling techniques for real-time periodic task systems has been the focus of many works, where known feasibility test have been extended to compute static slowdown factors [13, 2]. While static slowdown factors are computed based on the worst case execution time (WCET) of tasks, variations in task execution times result in dynamic slack which can be exploited for further energy savings. Slack reclamation heuristics have been proposed in [1, 5] to increase the extent of task slowdown. However, these techniques are primarily targeted for dynamic energy minimization.

With the shrinking device dimension, leakage currents are rapidly increasing. A five-fold increase in leakage current is predicted with each technology generation. Recently, leakage abatement has been an important focus on the work on power and energy minimization. Procrastination scheduling has been shown to minimize the leakage energy consumption by seeking to maximize idle intervals through delayed task execution. Irani *et. al.* [3] consider the combined problem of slowdown and shutdown and propose *competitive* off-line and on-line scheduling algorithm (for non-periodic task set). Procrastination scheduling has also been extended to periodic real-time systems. Lee *et. al.* [7] have proposed Leakage Control EDF (LC-EDF) algorithm and later works have enhanced the procrastination scheme [4, 11]. These techniques are based on statically computed slowdown factors and pre-computed procrastination intervals (based on the worst case execution times). In this paper, we propose dynamic slack reclamation techniques that work in conjunction with procrastination scheduling. We show that prior works on dynamic slowdown can be used in conjunction with procrastination scheduling, while ensuring task deadlines. While dynamic slack can be used for task slowdown, it can also be beneficial to use the dynamic slack for extended (dynamic) procrastination for leakage reduction. We propose slack reclamation techniques, which enable both dynamic slowdown and dynamic procrastination in a system. We achieve energy efficiency by wisely distributing the slack between slowdown and procrastination. We show that dynamic procrastination can increase the idle intervals to up to 1.7 times over static procrastination, which reduces the idle energy consumption by up to 60%.

The rest of the paper is organized as follows: Section 2 introduces the preliminaries and the system model. In Section 3, we present dynamic slack reclamation algorithms under procrastination scheduling. The experimental results are given in Section 4 and Section 5 concludes the paper with future directions.

2 System Model

The system consists of a task set of n periodic real time tasks, represented as $\Gamma = \{\tau_1, ..., \tau_n\}$. A 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 execution time (WCET) of the task at the maximum processor speed. The tasks are scheduled on a single processor system based on a preemptive scheduling policy. A task set is said to be *feasible* if all tasks meet the deadline. The processor utilization, $U = \sum_{i=1}^{n} C_i/T_i$, of less than or equal one is a necessary condition for the feasibility of any schedule [8]. In this work, we assume task deadlines are equal to the period (i.e. $D_i = T_i$, for each task τ_i) and tasks are scheduled by the Earliest Deadline First (EDF) scheduling policy [8]. All tasks are assumed to be independent and preemptive. Each invocation of the task is called a *job*. The notation of a task and the task instance (job) is used interchangeably, when the meaning is clear from the context. A priority function $P(J)$ is associated with each invocation of a task such that if a job J has a higher priority than J', then $P(J) > P(J')$.

A wide range of current embedded processors support variable voltage and frequency levels. We consider a uni-processor system with support for Dynamic Voltage Scaling (DVS). A *slowdown factor* (η) is defined as the normalized operating frequency, i.e., the ratio of the current frequency to the maximum frequency of the processor. Processors support discrete frequency levels and slowdown factors are discrete points in the range [0,1]. A static slowdown factor (η_i) is assigned to each task τ_i which ensures feasibility of the system. With the increasing dominance of leakage drain, performing the maximum possible slowdown need not be the most energy efficient operating point. Considering the static and dynamic energy consumption, the processor speed that minimizes the total energy per processor cycle is called the critical speed, denoted by η_{crit}. The speed η_{crit} can be computed for a given processor and is used as a lower bound on the static and dynamic task slowdown factors.

Procrastination Scheduling

Procrastination scheduling [7, 4], whereby task executions can be delayed to extend processor sleep intervals, is a part of our scheduling policy. We say a task is *procrastinated* (or delayed) if on task arrival, the processor is in a shutdown state and continues to remain in the shutdown state, despite the task being ready for execution. Procrastination scheme presented in [4] is an efficient scheduling algorithm and forms the basis of our work. Under this algorithm, a maximum procrastinated interval, Z_i, is pre-computed for each task τ_i, based on static slowdown factors. The details of the procrastination algorithm and the computation of Z_i (for each task τ_i) can be found in [4]. It has been shown that all deadlines are guaranteed if each task τ_i is procrastinated by no more than Z_i time units. We extend the work in [4] by proposing energy efficient slack reclamation algorithms. Note that the processor is shutdown only when the processor ready queue is empty and tasks are procrastinated only when the processor is shutdown. (Procrastination is handled by an additional controller, which takes over on processor shutdown.) Procrastination is also proposed in [11], however the algorithm has a worst case exponential time complexity, making it impractical.

Algorithm 1 Static Procrastination Algorithm

```
1: On arrival of a new job $J_i$:
2: if (processor is in sleep state) then
3:     if (Timer is not active) then
4:         $timer \leftarrow Z_i$; {Initialize timer}
5:     else
6:         $timer \leftarrow min(timer, Z_i)$;
7:     end if
8: end if

9: On expiration of Timer $(timer = 0)$:
10: Wakeup Processor;
11: Scheduler schedules highest priority task;
12: Deactivate timer;

13: Timer Operation:
14: timer – –; {Counts down every clock cycle}
```

The following results describes the computation of task procrastination intervals for a period task-set.

Theorem 1 *[4] Given tasks are ordered in non-decreasing order of their period, the procrastination algorithm guarantees all task deadlines if the procrastination interval Z_i of each task τ_i satisfies:*

$$\forall i,\ 1 \leq i \leq n: \qquad \frac{Z_i}{T_i} + \sum_{k=1}^{i} \frac{1}{\eta_k} \frac{C_k}{T_k} \leq 1 \tag{1}$$

$$\forall_{k<i}\ \ Z_k \leq Z_i \tag{2}$$

3 Dynamic Slack Reclamation

Dynamic slack reclamation schemes build upon static task slowdown factors for further energy savings. Prior works do not address slack reclamation in the presence of procrastination, which is the focus of this work. We begin with an overview of slack reclamation and define the following terms (same terms are used in prior work [15]). A *run-time* of a job is the time budget assigned to the job based on the static slowdown factor. The run-time for a task τ_i with workload (execution time at maximum speed) C_i and a static slowdown factor of η_i is C_i/η_i. Each run-time has an associated priority, which is the same as the job (task instance) priority. A job consumes run-time as it executes and early task completion results in dynamic slack (run-time). The unused run-time of a job is maintained in a priority list called the *Free Run Time list (FRT-list)*. It has been shown that a task can reclaim run-time with a priority higher than and equal to its own priority while guaranteeing all deadlines. The run-time that can be reclaimed by task τ_i at time t is denoted by $R_i^F(t)$. The list is maintained sorted by the priority of the run-time with the highest priority run-time at the head of the list. Run-time consumed form the FRT-list is always consumed from the head of the list (the highest priority run-time). The following rules are used in consuming the available run-time.

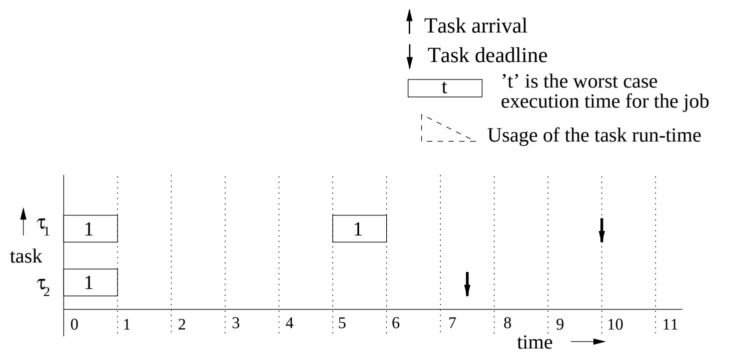
- As task τ_i executes, it consumes run-time at the same speed as the wall clock (physical time). If $R_i^F(t) > 0$, the run-time is used from the FRT-list, else the task uses its own run-time.
- When the system is idle (includes shutdown), it uses the run-time from the FRT-list if the list is non-empty.

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

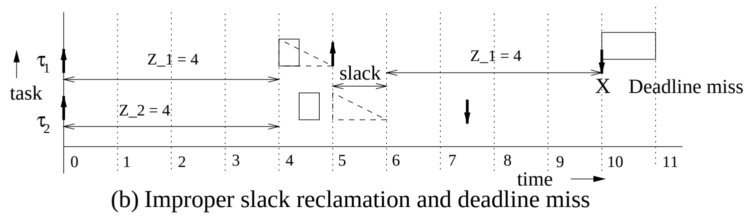
3.1 Motivation

Prior slack reclamation techniques use additional slack to further slowdown task executions. Excessive slowdown can increase the static energy contribution and it can be beneficial to reclaim slack for extended task procrastination interval, thereby minimizing leakage drain. We extend slack reclamation techniques to enable both dynamic slowdown and dynamic task procrastination. Prior works have shown that a task τ_i can reclaim available higher priority run-time ($R_i^F(t)$) for dynamic slowdown. We show that, even under procrastination scheduling, tasks can reclaim higher priority run-time (than the reclaiming task) to perform dynamic slowdown. Furthermore, we also propose algorithms to perform dynamic procrastination.

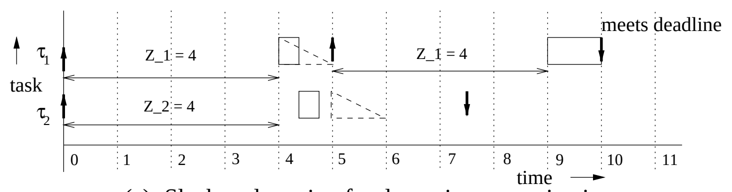
An intuitive solution for dynamic procrastination is to use free run-time ($R_i^F(t)$), available on task arrival time t, to extend the procrastination interval. We illustrate with an example that this approach can result in tasks missing the deadline. Consider a task set with two tasks $\tau_1 = \{5, 5, 1\}$ and $\tau_2 = \{7.5, 7.5, 1\}$. The task arrival times and deadlines are shown in Figure 1(a). Based on the procrastination algorithm in [4], the statically computed procrastination intervals for the tasks are $Z_1 = Z_2 = 4$. Consider that the processor is idle prior to time $t = 0$, when tasks τ_1 and τ_2 arrive. Since the processor is idle, the tasks can be procrastinated up to time $t = 4$ (by the procrastination algorithm). The processor wakes up at time $t = 4$ and executes the highest priority task. Both tasks have shorter execution times than their WCET and complete earlier, leaving the processor in the idle state prior to time $t = 5$. The next instance of task τ_1 arrives at time 5 with a deadline of $t = 10$. Since the processor is idle, the execution of τ_1 can be procrastinated. According to the run-time usage rules, run-time of task τ_1 is consumed in the interval $[4, 5]$. The processor is idle at $t = 5$, with run-time of task τ_2 (1 time unit with a deadline of 7.5) still available. The available run-time (1 unit) has a higher priority and can be reclaimed by task τ_1. However if this run-time is used to extend the procrastination interval by 1 time unit beyond the static procrastination interval of $Z_1 = 4$ (total of $4 + 1 = 5$ time units), then the processor remains idle up to time $t = 10$. This results in task τ_1 missing its deadline as shown in Fig. 1(b).



(a) Task set description: Task arrival times and WCET under static slowdown



(b) Improper slack reclamation and deadline miss



(c) Slack reclamation for dynamic procrastination

Figure 1. Slack reclamation for dynamic procrastination. (a) Task set description: task arrival times with worst case execution times. (b) Using the available (higher priority) run-time on task arrival to extend the static procrastination result in task τ_1 missing its deadline. (c) Procrastination by the maximum of the static procrastination interval and the (higher priority) slack available on task arrival meets all task deadlines.

We show that tasks can reclaim higher priority run-time, available at the end of the static procrastination interval (and not on task arrival), to further extend the idle interval. Since run-time is consumed when the processor is idle, slack can be exhausted by the end of the static procrastination interval. Thus on arrival of a task τ_i at time t, the task procrastination interval can be set to the maximum of the static procrastination interval (Z_i) and the free run-time ($R_i^F(t)$). In the above example, this results in task τ_1 (arriving at time $t = 5$) to be procrastinated by $max(1, 4) = 4$ time units. This rule results in a feasible schedule as shown in Fig. 1(c). When the reclaimable slack is greater than the static task procrastination interval, the slack can be reclaimed for extended procrastination.

Algorithm 2 Dynamic Slack Reclamation (DSR)

```
1:  **On arrival of a new job $J_i$:** $\{J_i$ is an instance of task $\tau_i\}$
2:  $R_i^r(t) \leftarrow \frac{C_i}{\eta_i}$;
3:  Add job $J_i$ to scheduler Ready Queue;
4:  **if** (processor is in sleep state) **then**
5:      $Z_i^D \leq max(Z_i, R_i^F(t))$; {set procrastination interval}
6:      **if** (Timer is not active) **then**
7:          $timer \leftarrow Z_i^D$ {Initialize timer}
8:      **else**
9:          $timer \leftarrow min(timer, Z_i^D)$;
10:     **end if**
11: **end if**

12: **On execution of each job $J_i$ :**
13: setSpeed $(max(\eta_{crit}, \frac{C_i^r(t)}{R_i^r(t)+R_i^F(t)}))$;

14: **On completion of job $J_i$ :**
15: Add to FRT-list($R_i^r(t), P(J_i)$);

16: **On expiration of Timer** $(timer = 0)$**:**
17: Wakeup Processor;
18: Scheduler schedules highest priority task;
19: Deactivate timer;

20: **Timer Operation:**
21: timer – –; {Counts down every clock cycle}
```

3.2 Dynamic Slack Reclamation

We now present a dynamic slack reclamation algorithm that works in conjunction with procrastination scheduling. First, we describe the notation used in the algorithm.

- J_i : the current job of task τ_i.
- $R_i^r(t)$: the available run-time of the current instance of task τ_i (i.e. J_i) at time t.
- $R_i^F(t)$: the free run-time (slack) available to job J_i at time t (i.e. run-time from the FRT-list with priority $\geq P(J_i)$).
- $C_i^r(t)$: the residual workload of job J_i.
- $R_i^{crit}(t)$: the run-time required to execute the residual portion of job J_i at the critical speed η_{crit}.

Algorithm 2 describes the slack reclamation scheme which can perform both dynamic slowdown and dynamic procrastination. The task run-time and dynamic slowdown is managed as follows. When a task arrives in the system, it is assigned a time budget based on the static slowdown factor and added to the ready-queue (line 2). On execution of a task, the available run-time for the task is its own run-time as well as the higher and equal priority run-time from the FRT-list ($R_i^F(t)$). The dynamic task slowdown factor is set to be the ratio of the residual workload to the available run-time. The algorithm ensures that the slowdown is never set below the critical speed, since it is not energy efficient to execute lower than the critical speed (line 13). On completion of the job, the unused run-time is added to the FRT-list. The algorithm also states how the dynamic slack can be used to extend task procrastination intervals. Let Z_i be the statically computed procrastination interval for each task τ_i and $R_i^F(t)$ be the available run-time on task arrival time t. The dynamic procrastination interval (Z_i^D) of each task τ_i is limited by $max(R_i^F(t), Z_i)$, which guarantees all task deadlines (line 5). Similar to the procrastination algorithm in [4], a timer is maintained to ensure that no task (τ_i) is delayed by more than its computed procrastination interval (Z_i^D). Note that Algorithm 2 does not explicitly determine the distribution of slack among slowdown and procrastination, but describes how slack can be utilized in either case. The two key points of this algorithm are: (1) the limit on dynamic task procrastination (line 5); and (2) the limit on dynamic task slowdown (line 13).

Lemma 2 *Given tasks are scheduled by the dynamic slack reclamation policy along with the dynamic procrastination algorithm, the run-time of each job is depleted at or before the job deadline.*

Proof: Suppose the claim is false. Let t be the first time that a run-time of a job or that in a FRT-list is not depleted by its deadline. In the rest of the proof, we use the term *high priority* run-time (job) to represent run-time (job) with a deadline less than or equal to t. Let t' be the the latest time before t such that the following two conditions are satisfied:
(1) the FRT-list does not contain any high priority run-time before t';
(2) either (a) no high priority jobs with arrival times before t' are pending ; or (b) the processor is idle before time t' (with or without high priority tasks pending).

Since no requests can arrive before system start time $(time = 0)$, t' is well defined. Constructed in this manner, the processor never stops consuming high priority run-time during the entire interval $[t', t]$. We consider the following two cases depending on whether high priority jobs are pending before time t'.

Case I : where no high priority jobs are pending prior to time t'. In this case, the run-time consumed in the interval is that generated by the high priority jobs arriving in the interval $[t', t]$. The run-time generated by the high priority jobs in the interval $[t', t]$ is bounded by $\sum_{k=1}^{n} \lfloor \frac{X}{T_k} \rfloor \frac{C_k}{\eta_k}$. Since the run-time is not depleted at time t, the generated run-time must be greater than the run-time consumed in the interval $[t', t]$, which is X. Therefore,

$$\sum_{k=1}^{n} \lfloor \frac{X}{T_k} \rfloor \frac{C_k}{\eta_k} > X$$

Since $\frac{X}{T_k} \geq \lfloor \frac{X}{T_k} \rfloor$, we have

$$\sum_{k=1}^{n} \frac{1}{\eta_k} \frac{C_k}{T_k} > 1$$

which contradicts with Equation 1 (assuming $Z_n = 0$).

Case II: the processor is idle prior to time t' with pending high priority jobs that arise due to task procrastination. Note that in this case, the high priority run-time generated in the interval $[t', t]$ can be larger than Case I due to the pending tasks. Let t_2 be the latest time before t when there are no pending high priority task prior to t_2. The run-time consumed in the interval X is generated by the high priority jobs arriving in the interval $[t_2, t]$. Let $Y = t - t_2$ be the length of the interval $[t_2, t]$, then the high priority run-time generated in the interval of length Y is bounded by $\sum_{k=1}^{j} \lfloor \frac{Y}{T_k} \rfloor \frac{C_k}{\eta_k}$, where j is the maximum task index with $T_j \leq Y$. The FRT-list has no high priority run-time prior to t' and the run-time generated in the interval $[t_2, t]$ (length Y) is consumed in the interval $[t', t]$ (length $X = t - t'$). Since the run-time is not depleted at time t, the run-time generated in interval of length Y must be greater than the run-time consumed in the interval $[t', t]$, which is X.

Therefore,

$$\sum_{k=1}^{i} \lfloor \frac{Y}{T_k} \rfloor \frac{C_k}{\eta_k} > X$$

Note that the high priority job arriving at time t_2, say τ_h, is not procrastinated by more than Z_h time units (cannot be procrastinated by $R_h^F(t) > Z_h$). If the task were procrastinated by $R_h^F(t) > Z_h$ then there should be high priority run-time throughout the sleep interval. Thus there would be high priority run-time prior to t', contradicting the definition of t'. Thus the job arriving at time t_2 ensures that the procrastination from time t_2 is bounded by Z_h. Note that j is the maximum task index such that $T_j < Y$. Since the task τ_h has a deadline less than t, $T_h \leq T_j$ and by Equation 2, the procrastination interval Z_h is bounded by Z_j. Thus it is true that $Y < X + Z_j$.

Therefore,

$$\sum_{k=1}^{i} \lfloor \frac{Y}{T_k} \rfloor \frac{C_k}{\eta_k} > Y - Z_j$$

Since $\frac{Y}{T_k} \geq \lfloor \frac{Y}{T_k} \rfloor$, we have

$$\frac{Z_j}{Y} + \sum_{k=1}^{j} \frac{1}{\eta_k} \frac{C_k}{T_k} > 1$$

Since all high priority jobs that contribute to the run-time in interval of length Y have their arrival time and deadline in the interval $[t_2, t]$, we have $T_j \leq Y$, and

$$\frac{Z_j}{T_j} + \sum_{k=1}^{j} \frac{1}{\eta_k} \frac{C_k}{T_k} > 1$$

which contradicts with Equation 1. Thus the run-time of each job is depleted no later than its deadline. ■

Theorem 3 *All tasks meet the deadline when scheduled by the dynamic slack reclamation algorithm (Algorithm 2) with procrastination scheduling.*

Proof: Note that a run-time is reserved for each job and the unused run-time is not added to the FRT-list until the job completes. Each run-time has the same deadline as the job deadline, and thus by Lemma 2 it follows that all jobs complete by their deadline. ■

3.3 Slack Distribution Policy

Given additional run-time (slack) for a job, using the entire slack for either dynamic slowdown or dynamic procrastination need not be an energy efficient solution. Slack reclamation should be wisely performed since the slack used for procrastination influence that (slack) available for slowdown and vice versa. Given the system is idle, using the entire slack for dynamic procrastination would not be energy efficient, if the incoming task has a static slowdown factor greater than the critical speed. On the other hand, leaving the slack entirely for dynamic slowdown need not be beneficial since the task might not be able to utilize the entire slack. Execution below the critical speed is not energy efficient and the extra slack available can result in many small idle intervals and increase leakage energy consumption. Once the processor is on and executing jobs, each task reclaims the slack to attain a speed as close as possible to the critical speed (this minimizes the energy consumed in executing the task).

Algorithm 3 describes a policy for distributing the slack between slowdown and procrastination. Determining the extent of dynamic procrastination for a task, when the processor is in the shutdown state, is crucial. We use the slack available on task arrival and the static task slowdown factor in computing the procrastination interval. Line 3 checks if the slack is sufficient to execute at the critical speed. If the entire slack would be consumed on execute the task at critical speed, then the algorithm does not perform dynamic procrastination (line 4). If extra slack is available even on executing the task at the critical speed, then the extra slack (Z_i^E) is used for dynamic procrastination (line 6). The dynamic procrastination interval Z_i^D is the maximum of the static procrastination interval (Z_i) and Z_i^E (shown in line 7 of Algorithm 3). The timer maintained for procrastination is updated based on the value of Z_i^D. The rest of the algorithm is the same as that of Algorithm 2. When the processor is woken up it uses the available slack for dynamic slowdown, with the critical speed being the lower bound on dynamic slowdown. We distribute slack between slowdown and procrastination in this manner to maximize energy efficiency.

Theorem 4 *All tasks meet the deadline when scheduled by the dynamic slack reclamation algorithm according to the slack distribution policy described in Algorithm 3.*

Proof: The task procrastination interval under the combined procrastination algorithm is always less than or equal to the available free run-time, $R_i^F(t)$. Since the static slowdown factors are greater than or equal to the critical speed, as seen in line 6 of the Algorithm 3 the procrastination interval will be smaller than $R_i^F(t)$. Thus the correctness of the algorithm follows from Theorem 3. ■

Algorithm 3 Slack Distribution Policy

```
1:  **On arrival of a new job $J_i$:**
2:  if (processor is in sleep state) then
3:      if ($R_i^F(t) + R_i^r(t) < R_i^{crit}(t)$ ) then
4:          $Z_i^E \leftarrow 0$;
5:      else
6:          $Z_i^E \leftarrow R_i^F(t) + R_i^r(t) - R_i^{crit}(t)$; {Note that $Z_i^E \leq R_i^F(t)$}
7:      end if
8:      $Z_i^D \leftarrow max(Z_i, Z_i^E)$;
9:      if (Timer is not active) then
10:         $timer \leftarrow Z_i^D$; {Initialize timer}
11:     else
12:         $timer \leftarrow min(timer, Z_i^D)$;
13:     end if
14: end if
15: Rest of the algorithm is same as Algorithm 2
```

4 Experimental Setup

We have implemented the proposed scheduling techniques in a discrete event simulator. To evaluate the effectiveness of our scheduling techniques, we consider several task sets, each containing up to 20 randomly generated tasks. We note that such randomly generated tasks is a common validation methodology in previous works [1, 7, 13]. Based on real life task sets [9], tasks were assigned a random period and WCET in the range [10 ms,125 ms] and [0.5 ms, 10 ms] respectively. Each task is assigned a static slowdown factor equal to the utilization at maximum speed, which is the optimal slowdown under EDF scheduling policy, to minimize the dynamic energy consumption [1]. If this slowdown factor is smaller than the critical speed, η_{crit}, then the slowdown factor is set to the critical speed. We generate varying execution times by varying 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 task sets with varying processor utilization (U) at maximum speed.

We use the power model for the Transmeta processor, based on the $70nm$ technology, consisting of both static and dynamic power consumption [4]. As described in the model, the critical speed of execution is $\eta_{crit} = 0.41$, the processor shutdown overhead is $483\mu J$ and the threshold idle interval for shutdown is 2.01 msec. We assume that the processor supports discrete voltage levels in steps of $0.05V$ in the range $0.5V$ to $1.0V$. These voltage levels correspond to discrete slowdown factors and each computed slowdown factor is mapped to the smallest discrete level greater than or equal to it. The upcoming idle interval is assumed to be the time period before the next task arrival in the system. The minimum guaranteed static procrastination interval is used to estimate the minimum idle interval.

We compare the energy consumption of the following techniques:

- **No Dynamic Slack Reclamation (no-DSR)**: where all tasks are executed at the static slowdown factor.
- **Dynamic Slack Reclamation with Static Procrastination (DSR-SP)** : where the reclaimed slack is used only for dynamic slowdown of the processor. Procrastination is based on statically computed task procrastination intervals (Z_i).
- **Dynamic Slack Reclamation with Dynamic Procrastination (DSR-DP)** : where the reclaimed slack is used for both dynamic slowdown and dynamic procrastination (combined slowdown and procrastination given by Algorithm 3).

4.1 Experimental Results

Figures 2 to 5 compare the energy savings of dynamic slack reclamation for different processor utilization (at maximum speed), U . For each value of U, we compare the following :

- sub-figure (a) (in Figs. 2, 3, 4, and 5) compares the total energy consumption of DSR-SP and DSR-DP normalized to the no-DSR policy. The variation of the BCET is along the X-axis and the normalized total energy along the Y-axis.
- sub-figure (b) (in Figs. 2, 3, 4, and 5) compares the average sleep interval and the average idle energy consumption of the DSR-DP normalized to DSR-SP policy. The increase in the sleep interval and the decrease in the idle energy is shown through two separate Y-axis for the same.

The energy gains for $U = 80\%$ are shown in Fig. 2(a). Reducing the BCET generates additional dynamic slack that can be reclaimed for additional energy savings. At high values of BCET, dynamic slowdown rarely reaches beyond the critical speed and it is energy efficient to utilize the entire slack for dynamic slowdown. From Fig. 2(b), we see that DSR-DP further reduces the idle energy consumption as BCET falls below 40%. A comparison of DSR-SP and DSR-DP shows that the average sleep interval under DSR-DP increases to up to 1.6 times that of the DSR-SP policy. The average idle energy is seen to reduce to up to 70% compared to DSR-SP.

We study the energy gains of dynamic slack reclamation for different values of processor utilization, U. We observe that the energy gains are greater at higher values of utilization (U) and decrease with lower utilization. Higher values of U result in higher static slowdown factors which consume more energy. Dynamic slack reclamation results in lowering the slowdown factors to result in higher energy

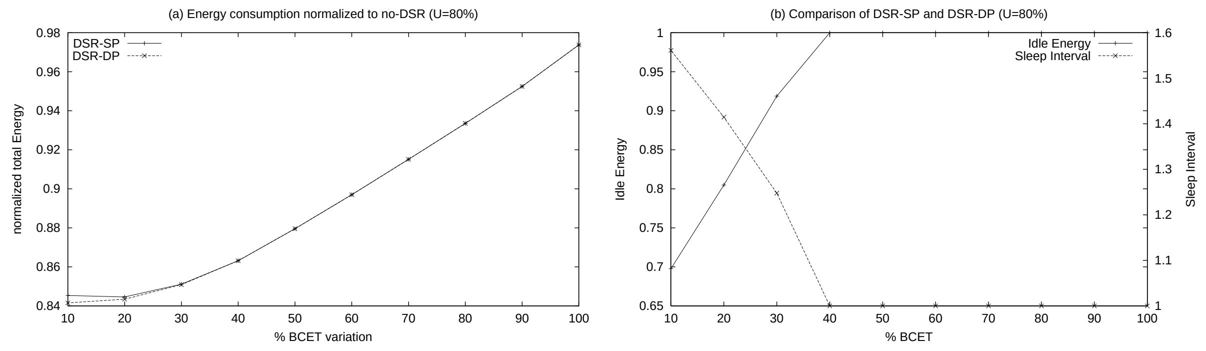


Figure 2. Utilization, U=80% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

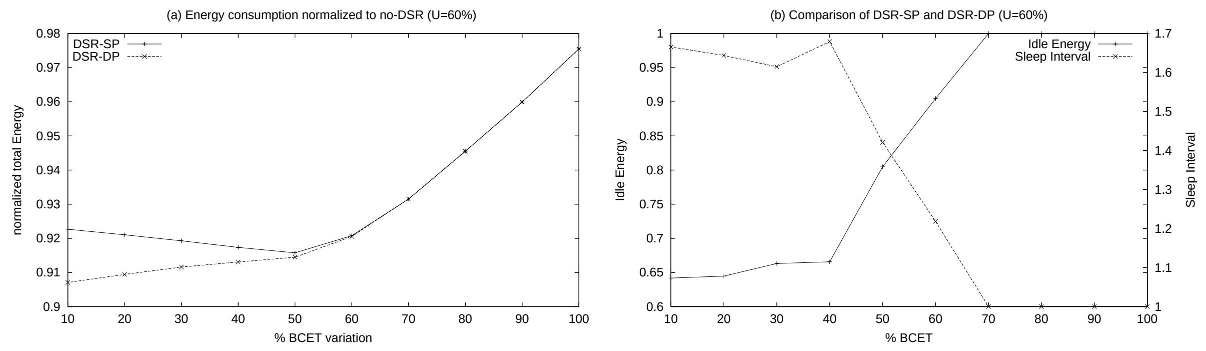


Figure 3. Utilization, U=60% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

gains. As the utilization decreases, the difference in the energy consumption between the static slowdown factors and the critical speed decrease and the relative gains are lower. DSR-DP improves the procrastination intervals as the dynamic slowdown factors fall below the critical speed. This occurs at lower values of BCET for higher values of U and vice versa. We also see that the energy gains of DSR-DP over DP-SP increase at lower values of U. Note that the total energy gains of DSR-DP over DSR-SP are not high. Majority of the short idle intervals that result in leakage are already avoided through static procrastination intervals which accounts for the bulk of the savings. Thus even though DSR-DP increases the average sleep interval duration, we do not see significant energy savings. Dynamic procrastination will result in significant energy gains when the static procrastination intervals are not long enough to perform shutdown (less than $t_{threshold}$). Additional procrastination will enable shutdown and reduce the leakage energy consumption.

For smaller values of U, dynamic slowdown reaches the critical speed for higher values of BCET. A comparison of the Figs. 2-5 shows that DSR-DP results in additional gains below BCET of 60% at $U = 60\%$ and below BCET of 80% at $U = 50\%$. We see that the sleep intervals under DSR-DP are increased up to 1.7 times and the idle energy is reduced to up to 60%. At a utilization $U = 40\%$ and lower, all tasks are executed at the critical speed ($\eta_{crit} = 0.41$). Static procrastination intervals

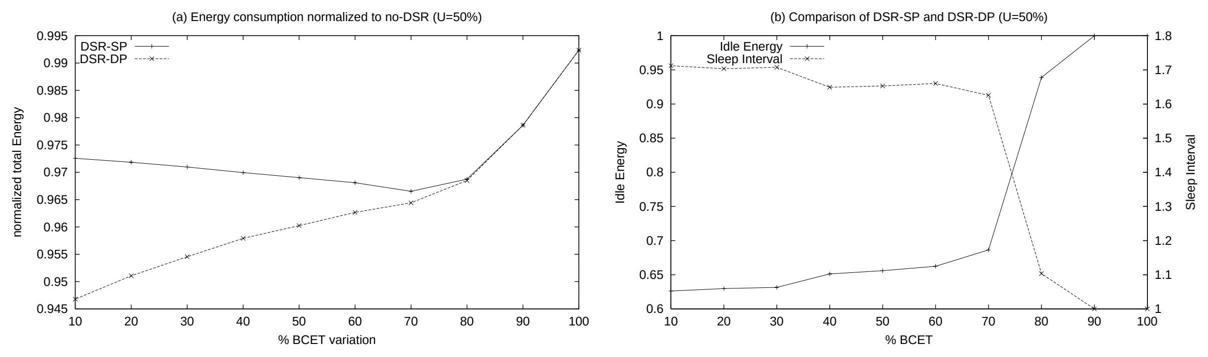


Figure 4. Utilization, U=50% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

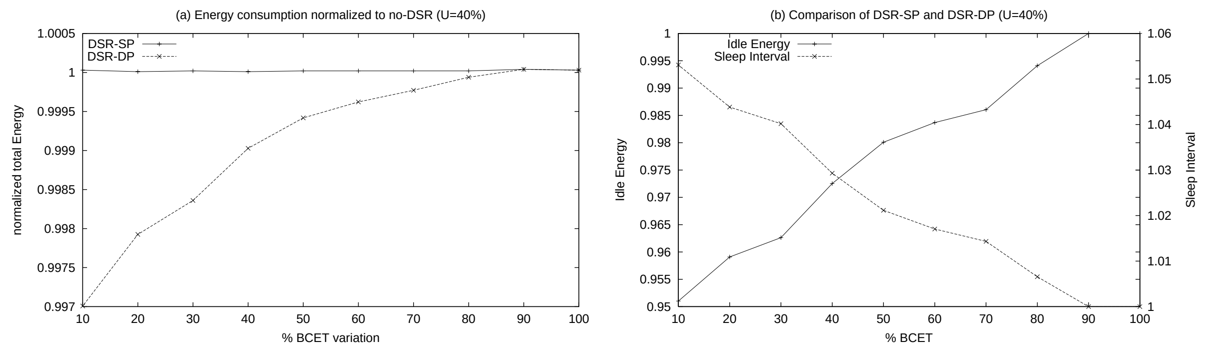


Figure 5. Utilization, U=40% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

are computed for the tasks, which dominates over the dynamic slack used to extend procrastination. Task execution are usually small and the accumulated free run-time slack rarely outperforms the static procrastination intervals. Thus DSR-DP does not result in significant energy savings at utilization lower than the critical speed.

5 Conclusions and Future Work

We present dynamic slack reclamation techniques that work in conjunction with procrastination scheduling to minimize the total static and dynamic energy consumption in a system. Reclaiming slack for dynamic slowdown results on an average 10% energy savings compared to no slack reclamation. We further enhance slack reclamation to enable both dynamic processor slowdown and dynamic task procrastination. Dynamic procrastination further decreases the idle energy consumption up to 70% while extending the average sleep interval to up to 1.7 times. Such task slowdown techniques along with combined static and dynamic procrastination are important as leakage drain continues to increase. The proposed techniques are simple and result in an energy efficient operation of the system. We plan to extend these techniques for energy efficient scheduling of all system resources.

References

[1] 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*, Dec. 2001.

[2] 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.

[3] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proceedings of Symposium on Discrete Algorithms*, Jan. 2003.

[4] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the Design Automation Conference*, pages 275–280, Jun. 2004.

[5] 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*, Mar. 2002.

[6] 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.

[7] Y. Lee, K. P. Reddy, and C. M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *EcuroMicro Conf. on Real Time Systems*, Jun. 2003.

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

[9] C. Locke, D. Vogel, and T. Mesler. Building a predictable avionics platform in Ada: a case study. In *Proceedings IEEE Real-Time Systems Symposium*, 1991.

[10] B. Mochocki, X. S. Hu, and G. Quan. A realistic variable voltage scheduling model for real-time applications. In *Proceedings of International Conference on Computer Aided Design*, Nov. 2002.

[11] L. Niu and G.Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 140–148, Sept. 2004.

[12] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processors. In *Proceedings of Design Automation and Test in Europe*, Mar. 2002.

[13] 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.

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

[15] 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*, Dec. 2002.