Dynamic Slack Reclamation with Procrastination Scheduling in Real-Time Embedded Systems

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

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

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 energy 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 extends the average sleep interval by 25% which reduces the idle energy consumption by 15%, while meeting all timing requirements.

Categories and Subject Descriptors: D.4.1 [Operating System]: Process Management – scheduling.

General Terms: Algorithms.

Keywords: dynamic slack reclamation, task procrastication, leakage power, critical speed, low power scheduling, real-time systems.

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, based on an off-line analysis of the task set characteristics (such as worst case task execution times and task periods); and (2) *dynamic* slowdown, where the slowdown decisions are made on-line based on the slack arising from varying task execution times. The extent of slowdown is usually referred to as a *slowdown factor*, which represented the operating speed normalized to the maximum processor speed. Among the earliest works on this problem, Yao *et. al.* [12] presented an off-line algorithm to compute the optimal static slowdown (speed) schedule for a set of N jobs. Dynamic voltage scaling techniques for real-time periodic task systems have also been the focus of many works in the literature [11, 3, 1]. These works have extended known feasibility test to compute static task slowdown factors. 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 that can be exploited for further energy savings. Slack reclamation heuristics have been proposed in [2, 7] to increase the extent of task slowdown. However, these techniques are primarily targeted for dynamic energy minimization.

With the shrinking device dimension, leakage current is 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.* [4] consider the combined problem of slowdown and shutdown and propose *competitive* off-line and on-line scheduling algorithm (for non-periodic task set). Lee *et. al.* [8] have extended procrastination scheduling to periodic real-time systems. Our earlier work [6] proposes an improved procrastination algorithm which works in conjunction with processor slowdown. The work in [6] is based on statically computed slowdown factors and pre-computed (static) task procrastination intervals (based on the worst case execution time). In this paper, we propose dynamic slack reclamation techniques that work

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC'05, June 13–17, 2005, Anaheim, California, USA.
Copyright 2005 ACM 1-59593-058-2/05/0006 ...$5.00.

in conjunction with procrastination scheduling. We show that prior works on dynamic slowdown can be used in conjunction with procrastination scheduling, while ensuring all task deadlines. While dynamic slack has been primarily used for task slowdown, slack can also be utilized to (dynamically) extend task procrastination intervals 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 formulates the problem. In Section 3, we present dynamic slack reclamation algorithms under procrastination scheduling. The experimental results are discussed in Section 4 and Section 5 concludes the paper with future directions.

2. PRELIMINARIES

2.1 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 independent and 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, $U = \sum_{i=1}^{n} C_i/T_i \leq 1$, is a necessary condition for the feasibility of any schedule [9]. 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 [9]. Each invocation of the task is called a *job* and the k^{th} instance of task τ_i is referred to as $\tau_{i,k}$. We use the notation of a task (τ_i) and its instance (job $J_\alpha = \tau_{i,k}$) interchangeably, when the meaning is clear from the context. A priority function $\mathcal{P}(J)$ is associated with each invocation of a task such that if a job J has a higher priority than J', then $\mathcal{P}(J) > \mathcal{P}(J')$.

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} [6]. The critical speed determines the lower bound on the processor speed, since execution at a speed lower than η_{crit} requires more time as well as consumes more energy. 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.

2.2 Static Task Procrastination

Procrastination scheduling [8, 6], 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. In our earlier work [6], we have presented a procrastination scheme where the procrastination intervals are based on static analysis (based on WCET of tasks). Under this algorithm, a maximum procrastinated interval, Z_i, is pre-computed for each task τ_i. The details of the procrastination algorithm and the computation of Z_i (for each task τ_i) can be found in [6]. 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.) We have shown that all deadlines are guaranteed if each task τ_i is procrastinated by no more than Z_i time units.

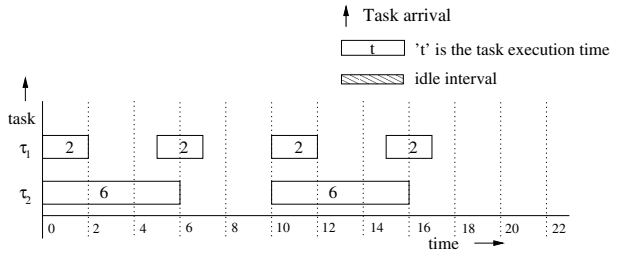
2.3 Dynamic Task Procrastination

Prior slack reclamation techniques primarily utilize the available slack to further slowdown task execution. While excessive slowdown can increase the static energy contribution, it can be beneficial to reclaim slack for extended task procrastination intervals, thereby minimizing leakage drain. We illustrate with an example how dynamic procrastination can extend the idle intervals in the system. Consider a task set consisting of the following two tasks:

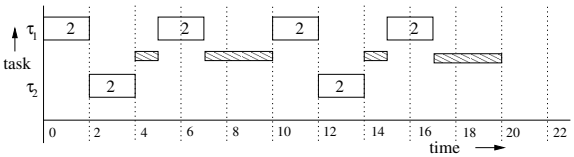
$$\tau_1 = \{5, 5, 2\} \text{ and } \tau_2 = \{10, 10, 6\}$$

The task arrival times and deadlines are shown in Fig. 1(a). The processor utilization (U) for the task set is 100% and no task execution can be procrastinated based on a static analysis, resulting in $Z_1 = Z_2 = 0$. The task schedule based on the EDF policy is shown in Fig. 1(b). At time zero, task $\tau_{1,1}$ is the highest priority ready task and it is scheduled for execution. The task completes execution at time $t = 2$ and task $\tau_{2,1}$ begins execution. The task instance $\tau_{2,1}$ has a shorter execution time than its worst case and completes in two time units, resulting in a slack of $6 - 2 = 4$ time units. The processor is idle at time $t = 4$ and task $\tau_{1,2}$ arrives at time $t = 5$ with a deadline of $t = 10$. Since $Z_1 = 0$, the task execution is not delayed and and task $\tau_{1,2}$ begins execution as soon as it arrives. The slack generated by task $\tau_{2,1}$ has the same priority (deadline) as task $\tau_{1,2}$ and can be reclaim by task $\tau_{1,2}$ for slowdown. Assuming the task is executing at the critical speed, it is not energy efficient to reclaim this slack for dynamic slowdown. Thus task $\tau_{1,2}$ executes at the assigned slowdown factor to complete execution at time $t = 7$ and the processor is again idle upto time $t = 10$. During the time interval [10,20], tasks have a similar schedule and is shown in Fig. 1(b).

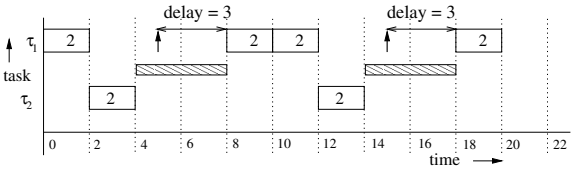
We show how task executions can be dynamically procrastinated to extend idle intervals, based on the available slack in the system. We later prove that a task execution can be delayed by the available slack with higher or equal priority than the task priority. The schedule with dynamic procrastination is shown in Fig. 1(c). Task $\tau_{1,1}$ begins execution at time zero which is followed by the execution of task $\tau_{2,1}$. The slack generated by the early completion of task $\tau_{2,1}$ has a deadline of $t = 10$. This slack can be reclaimed to dynamically procrastinate the execution of task $\tau_{1,2}$, which arrives at time $t = 5$. Since 4 units of slack is available at time $t = 4$, the execution of task $\tau_{1,2}$ can be delayed upto time $t = 8$. Task $\tau_{1,2}$ begins execution at $t = 8$ to complete execution by its deadline. Thus dynamic task procrastination can extend the idle interval to 4 time units, as opposed two idle intervals of 1 and 3 time units. Assuming a shutdown threshold to be 2 time units, procrastination can enable a shutdown throughout the idle intervals in Fig. 1(c), which is not possible in Fig. 1(b). When tasks have statically computed procrastination intervals, the maximum of the available slack and the static procrastination interval can be used to procrastinate task execution. It is important to note that the available slack cannot be added to the static procrastination interval of a task (an example illustrating the same is presented in [5]).



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



(b) Task schedule (without dynamic task procrastination).



(c) Extended idle intervals with dynamic task procrastination

Figure 1: Slack reclamation for dynamic procrastination. (a) Task set description: task arrival times with worst case execution times. (b) Task schedule under statically computed task procrastination intervals and many idle intervals. (c) Reclaiming higher priority slack to dynamically extend task procrastination intervals. Tasks can be delayed by the maximum of the static procrastination interval and the (higher priority) slack available on task arrival, while meeting all task deadlines.

3. DYNAMIC SLACK RECLAMATION

Dynamic slack reclamation schemes build upon static task slowdown factors for further energy savings. Prior works [2, 13] do not address slack reclamation in the presence of procrastination, which is the focus of this work.

3.1 Slack Reclamation Algorithm

The static slowdown factors determine the time budget, referred to as the task *run-time*, alloted to each task. The run-time for a task with workload (execution time at maximum speed) C and a static slowdown factor of η is C/η. 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)*. 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 from the FRT-list is always consumed from the head of the list (the highest priority run-time). Similar to known techniques [2], a task can reclaim run-time with a priority higher than or equal to its own priority while guaranteeing all deadlines. We use the following notation in the slack reclamation algorithm (similar notation is used in [13]).

- 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 $\geq \mathcal{P}(J_i)$).
- $C_i^r(t)$: 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 1 describes the slack reclamation scheme which can perform both dynamic slowdown and dynamic procrastination. When a task arrives in the system, it is assigned a time budget based on the static slowdown factor (line 2). Each task is eligible to use its own run-time as well as the higher and equal priority run-time from the FRT-list ($R_i^F(t)$ for task τ_i). The dynamic task slowdown factor is set to 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 at a speed lower than the critical speed (line 13). 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 [6], a timer is maintained to ensure that no task (τ_i) is delayed by more than its computed procrastination interval (Z_i^D).

Algorithm 1 Slack Reclamation Algorithm

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: Set Z_i^D to any number in the range $[0, max(Z_i, R_i^F(t))]$;
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$(R_i^r(t), \mathcal{P}(J_i))$;

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}

The following rules are used in consuming the 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.

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

Note that Algorithm 1 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: (a) the limit on dynamic task procrastination (line 5); and (b) the limit on dynamic task slowdown (line 13).

THEOREM 1. *All tasks meet the deadline when scheduled by the dynamic slack reclamation algorithm (Algorithm 1) with procrastination scheduling.*

The details of the proof are given in [5].

3.2 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 influences that (slack) available for slowdown and vice versa. Given the system is idle, using the entire slack for dynamic procrastination is not 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 turned on and executing jobs, each task reclaims the slack to execute at the lowest permissible speed greater than or equal to the critical speed (this minimizes the energy consumed in executing the task).

Algorithm 2 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 the task at the critical speed. If the entire slack would be consumed on executing the task at the critical speed, then the algorithm does not perform dynamic procrastination (line 4). Given that slack would be available even after 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 2). 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 1. 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 2. *All tasks meet the deadline when scheduled by the dynamic slack reclamation algorithm according to the slack distribution policy described in Algorithm 2.*

The details of the proof are present in [5].

Algorithm 2 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 1**

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 [2, 8, 11]. Based on real life task sets [10], tasks are 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 maintains the system feasibility. If this slowdown factor is smaller than the critical speed, η_{crit}, then the slowdown factor is increased 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, μ = (WCET+BCET)/2 and a standard deviation, σ = (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 70*nm* technology, which accounts for both static and dynamic power consumption [6]. 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. Procrastination can guarantee a minimum procrastination interval (minimum Z_i over all tasks) and this information is used to estimate the minimum idle interval with procrastination.

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 with statically computed task procrastination intervals.

- **Dynamic Slack Reclamation with Static Procrastination (DSR-SP)**: where slack is reclaimed 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 slack is reclaimed for both dynamic slowdown and dynamic procrastination (combined slowdown and procrastination given by Algorithm 2).

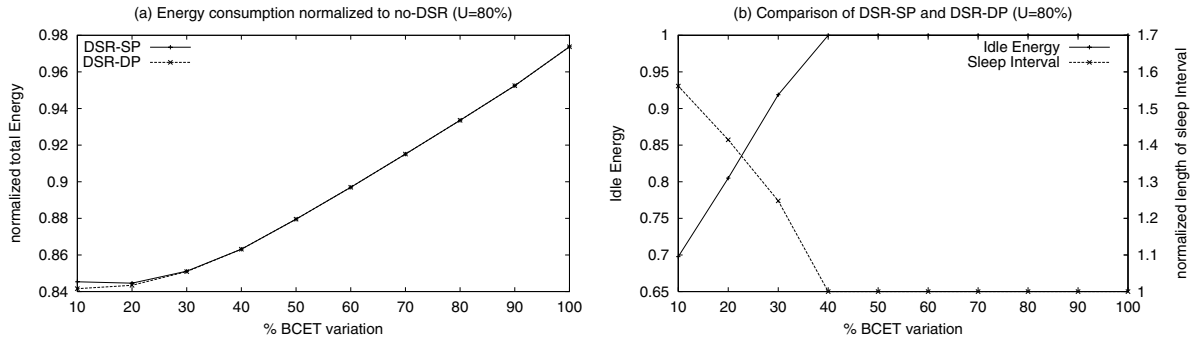


Figure 2: Utilization, U=80% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

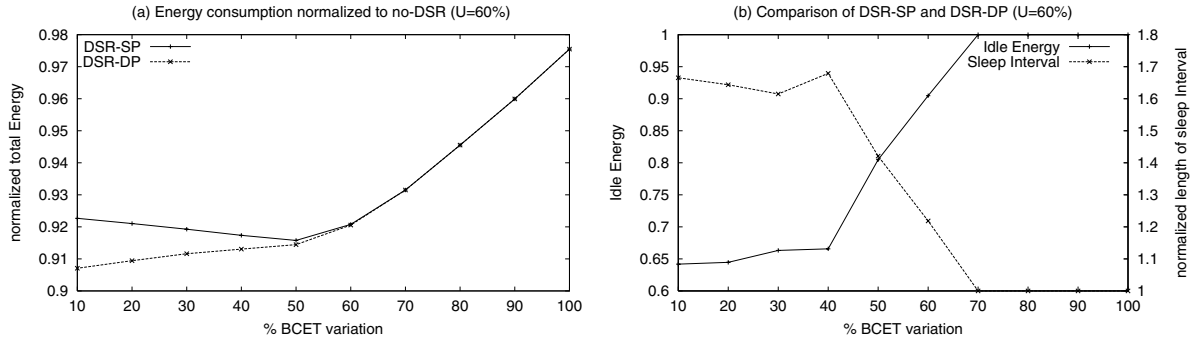


Figure 3: Utilization, U=60% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

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.

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) and higher energy savings are achieved by lowering these slowdown factors through slack reclamation. 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 to result in additional energy savings. This occurs at lower values of BCET for higher values of U and vice versa.

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 up to 70% compared to DSR-SP.

For smaller values of U, dynamic slowdown reaches the critical speed for higher values of BCET. A comparison of the Fig. 2 - Fig. 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 are computed for the tasks, which dominate over the dynamic slack available to extend procrastination intervals. Task execution time is 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.

Note that the total energy gains of DSR-DP over DSR-SP are not high. This is because 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. However, when the static procrastination intervals are not long enough for shutdown to be energy efficient, dynamic procrastination will extend the idle intervals to result in significant energy savings.

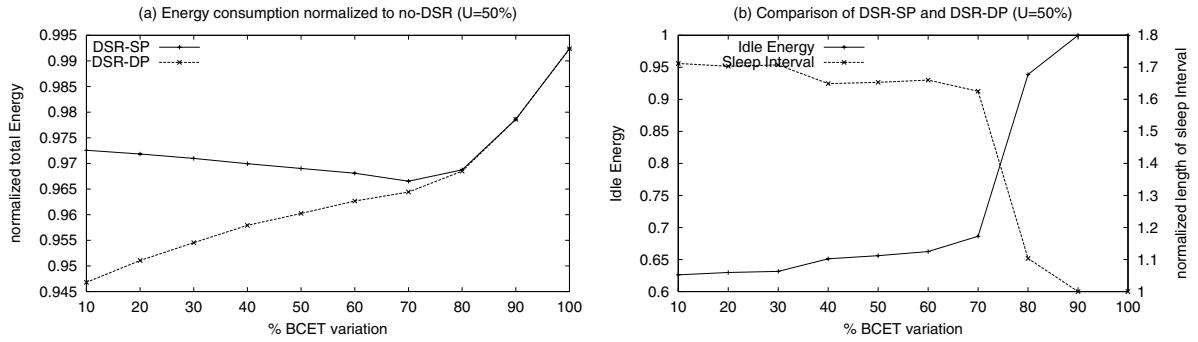


Figure 4: Utilization, U=50% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

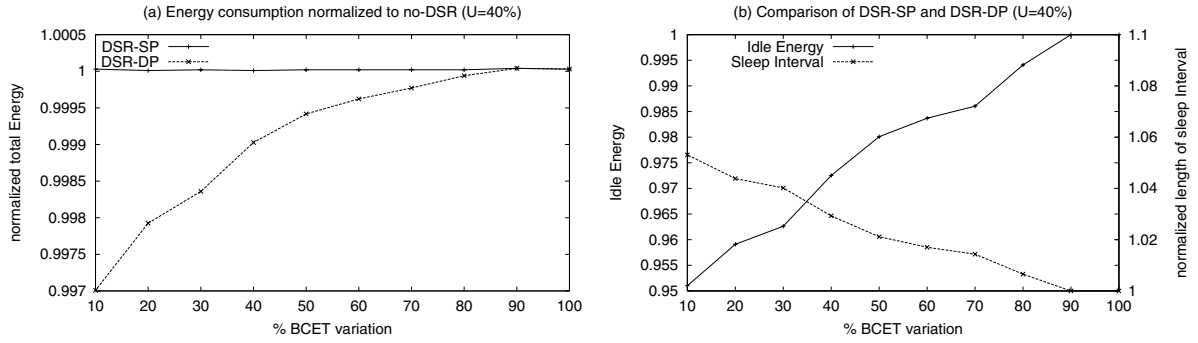


Figure 5: Utilization, U=40% (a) Comparison of total energy of DSR-SP and DSR-DP normalized to no-DSR (b) Comparison of average idle energy and average sleep interval of DSR-DP normalized to DSR-SP

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. We enhance slack reclamation to enable both dynamic processor slowdown and dynamic task procrastination. Reclaiming slack for dynamic slowdown results on an average 10% energy savings compared to no slack reclamation. Dynamic procrastination further increases the energy savings by reducing the idle energy consumption up to 70% (through longer sleep intervals). Such task slowdown techniques along with combined static and dynamic task 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.

6. 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 *Proceedings 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*, Dec. 2001.

[3] 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.

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

[5] R. Jejurikar and R. Gupta. Leakage aware dynamic slack reclamation in real-time embedded systems. In *CECS Technical Report #04-31, UC Irvine*, Nov. 2004.

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

[7] 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*, Mar. 2002.

[8] 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*, 2003.

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

[10] 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.

[11] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proc. of ICCAD*, pages 365–368, Nov. 2000.

[12] 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.

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