Dual Mode Algorithm for Energy Aware Fixed Priority Scheduling with Task Synchronization

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

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

Abstract

Slowdown factors determine the extent of slowdown a computing system can experience based on functional and performance requirements. Dynamic Voltage Scaling (DVS) of a processor based on slowdown factors can lead to considerable energy savings. We address the problem of computing static and dynamic slowdown factors in the presence of task synchronization. In this work, tasks are allowed to have different slowdown factors based on the task characteristics. This is increasingly important as the task set becomes diverse. We present the Dual Mode (DM) algorithm under the rate monotonic scheduling policy. We introduce the notion of frequency inheritance which bounds the blocking time to guarantee the task deadlines. Sufficient constraints have been identified for the feasibility of the task set under synchronization. We formulate this problem of computing the static slowdown factors for tasks as an optimization problem to minimize the total energy consumption of the system. Our simulation experiments show on an average 10% energy gains over the known slowdown techniques.

1 Introduction

Power is one of the important metrics for optimization in the design and operation of embedded systems. The processor consumes a significant portion of the total energy. There are two primary ways to reduce processor power consumption in embedded computing systems: processor shutdown and processor slowdown. Slowdown using frequency and voltage scaling is more effective in reducing the energy consumption [3, 4, 18, 23]. However, scaling the processor frequency leads to an increase in the execution time of a job. In real-time systems, we want to minimize energy while adhering to the task deadlines. Minimizing power and meeting deadlines are contradictory goals and we have to judiciously manage time and power to achieve our goal of minimizing energy.

In this paper, we focus on system level power management by the computation of static and dynamic slowdown factors for the tasks in the system. We assume a real-time system in which the tasks run periodically and have fixed deadlines. These tasks are scheduled on a single processor based system using rate monotonic scheduling. Tasks synchronize to enforce mutual exclusive access to the shared resources. We compute static slowdown factors in the presence of task synchronization to minimize the energy consumption of the system.

Most of the earlier work dealt with independent task sets. Shin *et al.* [18] have computed uniform slowdown factors for an independent periodic task set. Yao, Demers and Shanker [22] presented an optimal off-line speed schedule for a set of N jobs. An optimal schedule for tasks with different power consumption characteristics is considered by Aydin, Melhem and Mossé [3]. The same authors [4] prove that the processor utilization (at maximum speed) is the optimal slowdown factor when the deadline is equal to the period. Quan and Hu [15] [16] discuss off-line algorithms for the case of fixed priority scheduling.

Since the worst case execution time (WCET) of a task is not usually reached, there is dynamic slack in the system. Pillai and Shin [14] recalculate the slowdown to use the dynamic slack while meeting the deadlines. Low-power scheduling using slack reclamation heuristic is studied by Aydin *et al.* [4] and Kim *et al.* [10]. Scheduling of task graphs on multiple processors has also been considered. Luo and Jha [12] have considered scheduling of periodic and aperiodic task graphs in a distributed system. Non-preemptive scheduling of a task graph on a multi processor system is considered by Gruian and Kuchcinski [6]. Zhang *et al.* [24] have given a framework for task scheduling and voltage scaling of dependent tasks on a multi-processor system. They have formulated the voltage scaling problem as an integer programming problem.

In real life applications, tasks access the shared resources in the system. Low power scheduling in the presence of task synchronization [7] and non-preemptive sections [23] has been considered. Previous work assumes that all tasks have a constant static slowdown factor. In this paper, we present a dual mode algorithm which overcomes this limitation. We compute static slowdown factors by formulating the problem as a convex optimization problem. Furthermore, we present a dynamic slowdown algorithm, where we slowdown both the critical and non-critical sections of a task. We gain as much as 10% energy savings over the known techniques.

The rest of the paper is organized as follows: Section 2 formulates the problem. In Section 3, we present the dual mode algorithm under rate monotonic scheduling. In Section 4, we formulate the computation of slowdown factors as an optimization problem. The experimental results are given in Section 5. Finally, Section 6 concludes the paper with future directions.

2 Preliminaries

In this section, we introduce the necessary notation and formulate the problem. We first describe the system model followed by a motivating example.

2.1 System Model

A task set of n periodic real time tasks is represented as $\Gamma = \{\tau_1, ..., \tau_n\}$. A 3-tuple $\{T_i, D_i, C_i\}$ is used to represent each task τ_i, where T_i is the period of the task, D_i is the relative deadline with $D_i \leq T_i$, and C_i is the worst case execution time (WCET) of the task at maximum speed, given that it is the only task running in the system. Each invocation of the task is called a *job*. A priority function $\mathcal{P}(J)$ is associated with each job such that if a job J has a higher priority than J', then $\mathcal{P}(J) > \mathcal{P}(J')$. Under rate monotonic (deadline monotonic) scheduling, the shorter the period (deadline), the higher the priority. The techniques presented in this paper apply to both rate monotonic and deadline monotonic scheduling.

Each system usually has a set of shared resources. Access to the shared resources are mutually exclusive in nature. Common synchronization primitives include semaphores, locks and monitors [19]. We assume that semaphores are used for task synchronization. When a task has been granted access to a shared resource, it is said to be executing in its *critical section*. The k^{th} critical section of task τ_i is represented as $z_{i,k}$. We assume critical sections of a task are properly nested [17], wherein if two critical sections within a task overlap, then one critical section lies completely within the other. Due to the resource sharing, a task can be *blocked* by lower priority tasks. A critical section is called a *blocking section* if a higher priority job is blocked for the completion of the critical section. If no higher priority job is blocked when a job is executing, the job is said to be executing in its *non-blocking section*. Let B_i be the maximum blocking time for task τ_i under the given resource access protocol.

2.2 Variable Speed Processors

A wide range of processors like the Intel XScale [1] and Transmeta Crusoe [2] support variable voltage and frequency levels. Voltage and frequency levels are tightly coupled. When we change the speed of a processor we change its operating frequency. We proportionately change the voltage to a value which is supported at that operating frequency. The important point to note is when we perform a slowdown we change both the frequency and voltage of the processor. Given the minimum frequency f_{min} and the maximum supported frequency f_{max}, we normalize the speed to the maximum frequency to have discrete points in the operating range $[\eta_{min}, 1]$, where $\eta_{min} = f_{min}/f_{max}$.

The *slowdown factor* can be viewed as the normalized frequency. At a given instance, it is the ratio of the scheduled frequency to the maximum frequency of the processor. We assume that the speed of the processor can be varied over a discrete range. In this paper, we assume the overhead incurred in changing the processor speed is incorporated in the task execution time. A speed change can only occur at a context switch and when a task is blocked. This overhead, similar to context switch overhead, is constant and can be included in the worst case execution time of the task. All tasks are assumed to be preemptive, however access to the shared resources need to be serialized. Our aim is to schedule the given task set and the processor speed such that all tasks meet their deadlines and the energy consumption is minimized.

2.3 Motivating example

Consider a simple real time system with three periodic tasks having the following parameters :

$$\tau_1 = \{5, 5, 1\}, \tau_2 = \{10, 10, 4\}, \tau_3 = \{80, 80, 2\} \qquad (1)$$

Based on rate monotonic scheduling, task τ_1 has the highest priority and the task τ_3 has the lowest priority. All tasks access a shared resource through a semaphore S. The critical section for the tasks are the intervals $z_{1,1} = [0.5, 1]$, $z_{2,1} = [3.5, 4]$ and $z_{3,1} = [0, 1]$ within each task. Based on the stack resource protocol, the blocking times for the tasks are $B_1 = 1$, $B_2 = 1$ and $B_3 = 0$. The arrival times for

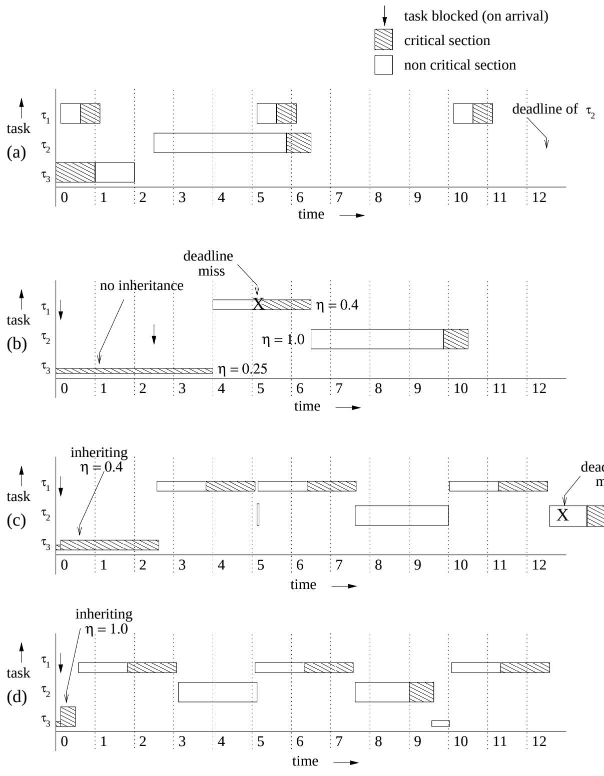

Figure 1. Motivation for frequency inheritance (a) Task arrival times and deadlines (period=deadline) with critical sections. (b) No frequency inheritance and job $\tau_{1,1}$ misses deadline. (c) Inheriting the frequency of the blocked task $\eta_1^S = 0.4$ and task $\tau_{2,1}$ misses deadline. (d) Inheriting a slowdown of $max(\eta_1^S, \eta_2^S) = 1.0$, meets all deadlines while observing blocking.

the tasks are $\phi_1 = 0.1$, $\phi_2 = 2.6$, $\phi_3 = 0$. This task set is shown in Figure 1(a). The jobs for each task are shown at their arrival time with their workload. The jobs are to be scheduled on a single processor by a rate monotonic scheduler. We cannot ignore the blocking times in the computation of slowdown factors [8]. The feasibility test in the presence of synchronization is given in [17] (Equation 4), which implies the task set is feasible for all task phases (the phase of a task is the arrival time of the first instance of the task). The given task set is schedulable at full speed. If we extend the schedulability test given in [17] to consider the task slowdown factors, we have the following constraints : $\forall_i, \frac{B_i}{\eta_i} + \sum_{k=1}^{i} \lceil \frac{T_i}{T_k} \rceil \frac{C_k}{\eta_k} \leq T_i$. Assume that based on the tasks power characteristics, we compute slowdown factors which satisfy the above constraints to have $\eta_1 = 0.4$, $\eta_2 = 1.0$, $\eta_3 = 0.25$. We consider the task schedule at this computed slowdown.

As shown in Figure 1(c), task τ_3 arrives at time $t = 0$ and enters the critical section. Task τ_1 arrives at time $t = 0.1$ and is blocked under SRP, since it eventually needs the shared resource that is locked by task τ_3. Note that the above mentioned constraints assume the same slowdown during the blocking interval as the task slowdown. Thus the blocking time for task τ_1 assumed in the constraint is $1/0.4 = 2.5$. However, the blocking time arises from task τ_3 which has a smaller slowdown factor, $\eta_3 = 0.25$. This increases the blocking time for the task to $1/0.25 = 4$ time units. Thus the actual blocking time is greater than 2.5 and results in task τ_2 missing its deadline as shown in Figure 1(b). We introduce the property of frequency inheritance to bound the blocking time under slowdown.

In Figure 1(c), we show the schedule where a blocking task inherits the maximum slowdown over all the blocked tasks. When the task τ_1 is blocked on arrival at time $t = 0.1$, the blocking task τ_3 inherits the slowdown of task τ_1, $\eta_1 = 0.4$. This bounds the blocking time to $1/0.4 = 2.5$ and task τ_1 meets its deadline. However, this does not guarantee all tasks meeting the deadline and task τ_2 misses its deadline as seen in the Figure 1(c).

To ensure that all tasks meet the deadline, we have to take into consideration the implicit blocking of other higher priority tasks. When task τ_1 is blocked, inheriting a slowdown of the maximum slowdown factor of all higher priority tasks up to task τ_1, guarantees meeting all task deadlines. As seen in Figure 1(d), inheriting a slowdown of $max(0.4, 1) = 1.0$ ensures meeting all deadlines. Thus we introduce a slowdown called the blocking slowdown factors, that a task needs to inherit, when it blocks another task.

These are the slowdown rules considering task synchronization, referred to as the synchronization mode. We also compute slowdown factors for tasks assuming they are independent, which is referred to as independent mode. We present the dual mode algorithm, which describes how the two modes can be combined at run-time to minimize the total energy consumption.

3 Static Slowdown Factors

We compute static slowdown factor for a system with an underlying rate monotonic (RM) scheduler. The problem of scheduling tasks in the presence of resource sharing is NP-hard [20]. Resource access protocols have been designed to bound the blocking times and *sufficient* schedulability tests have been given in the presence of maximum blocking times. The *Stack Resource Protocol (SRP)* [5] has the property of blocking a task only at its arrival in the system. When a new job J arrives in the system, it is blocked under SRP, if a lower priority job is holding a resource required by the job J or a higher priority job. Let B_i be the *maximum blocking time* for task τ_i based on the SRP.

3.1 Rate Monotonic Scheduling

Let $\Gamma = \{\tau_1, \cdots, \tau_n\}$ be the tasks in the system ordered in non-increasing order of their priority. Lehoczky et al. [11] showed that the schedulability analysis is needed only at discrete points, called the *scheduling points*. Given the tasks are sorted in non-increasing order of their priority, the set of scheduling points for task τ_i is defined by

$$S_i = \{kT_j | j = 1, ..., i; k = 1, ..., \lfloor \frac{T_i}{T_j} \rfloor\} \tag{2}$$

when the period is the same as the deadline, $T_i = D_i$. If D_i is less than T_i, we consider only the scheduling points up to the task deadline D_i. This set of scheduling point, S_i', is defined as follows:

$$S_i' = \{(t \in S_i) \wedge (t < D_i)\} \cup \{D_i\} \tag{3}$$

The schedulability test in the presence of blocking time is given by Sha et al. [17]. If the tasks are sorted in non-increasing order of priority and B_i is the blocking time for task τ_i, then τ_i can be feasibly scheduled if there *exists* at least one scheduling point $S_{ij} \in S_i$, which satisfies

$$B_i + \sum_{k=1}^{i} C_k \lceil \frac{S_{ij}}{T_k} \rceil \leq S_{ij} \tag{4}$$

3.1.1 Independent Mode (Indep)

Assuming the task are independent (in the absence of blocking), let η_i^I be the slowdown factors for task τ_i.

Theorem 1 *: A task set of n independent periodic tasks, sorted in non-increasing order of their priority, is feasible at a slowdown factor of η_i^I for task τ_i, if there exists one or more scheduling points S_{ij} per task that satisfy:*

$$\sum_{k=1}^{i} \frac{1}{\eta_k^I} C_k \lceil \frac{S_{ij}}{T_k} \rceil \leq S_{ij} \tag{5}$$

3.1.2 Synchronization Mode (Sync)

Let B_i be the maximum blocking encountered by each task τ_i, under a given resource access protocol and let η_i^S be the slowdown factors in the presence of blocking.

Theorem 2 *: A task set of n periodic tasks, sorted in non-increasing order of their priority, is feasible at a slowdown factor of η_i^S for each task τ_i, if there exists at least one scheduling point S_{ij} for each task τ_i which satisfies:*

$$\frac{1}{\eta_i^S} B_i + \sum_{k=1}^{i} \frac{1}{\eta_k^S} C_k \lceil \frac{S_{ij}}{T_k} \rceil \leq S_{ij} \tag{6}$$

and the critical section of every blocking task, τ_b, inherits a slowdown of $max_{j=m}^{b} \eta_j^S$, where τ_m is the blocked job with the highest priority level.

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

3.1.3 Blocking slowdown factor η_m^B

We show in Figure 1(c) that inheriting the maximum slowdown over all blocked tasks does not guarantee task deadlines and we define a new slowdown factor that must be inherited by a blocking task. We define the *blocking slowdown factor*, $\eta_m^B(b)$, for task τ_m, as the slowdown factor that the blocking critical section of task τ_b must inherit to guarantee deadlines of all higher priority tasks. As given by Theorem 2, $\eta_m^B(b) = max_{j=m}^{b} \eta_j^S$. Since the blocking slowdown factor of a task depends on the blocking task, a task may have to maintain blocking slowdown factors for each blocking task. We would like to have only one blocking slowdown factor per task and we use the maximum of all blocking slowdown factors for a task. This maximum is called the *blocking slowdown factor*, η_m^B, and is computed as the maximum synchronization slowdown of all the lower or equal priority tasks.

$$\eta_m^B = max_{j=m}^{n-1} \eta_j^S \tag{7}$$

3.2 Dual Mode Algorithm

We present a *Dual Mode (DM)* algorithm for energy aware scheduling with synchronization. The two modes of operation are *independent mode (Indep)* and *synchronization mode (Sync)*. The DM algorithm is based on the use of Stack Resource Protocol (SRP). The slowdown factors of η_i^I and η_i^S are used by task τ_i in the independent and synchronization mode respectively. The system starts in the independent mode. When a new job arrives in the system, the protocol checks if this job is blocked. If a lower priority task τ_b blocks the new task in the independent mode, then the system enters the synchronization mode. The blocking job *inherits* the *blocking slowdown factor* and the *priority* of the maximum priority blocked task. All tasks with a higher priority than task τ_b execute in the synchronization mode, at a slowdown of η_i^S. The dual mode algorithm is given in Figure 2. When the system enters the synchronization mode, the priority of the blocking job $\mathcal{P}(J_b)$ is marked as shown in line (5) of the algorithm. In lines 6 and 7, the blocked job inherits the priority and the slowdown factor of the highest priority blocked task. *setSpeed(η)* sets the CPU speed to the specified slowdown factor η. The system changes back to independent mode, when it executes a task with lower or equal priority than the marked priority. Thus, the non-blocking section of task τ_b executes in the independent mode. This is an improvement over previous work [23] where the mode is not changed until the completion of τ_b.

Since the slowdown in independent mode η_i^I is smaller than or equal to the slowdown in synchronization mode η_i^S, this leads to energy savings. We say a job executed in synchronization mode if it begins execution in the synchronization mode. Such a job executes to completion in the syn-

```
    On arrival of a new job $J_i$ :
(1) if ( $J_i$ is Blocked )
(2)     Let $J_b$ be the job blocking $J_i$;
(3)     if $(mode = Indep)$
(4)         $mode \leftarrow Sync$;
( )     endif
(5)     $MarkedPrio \leftarrow min(MarkedPrio, \mathcal{P}(J_b))$;
(6)     $J_b$ inherits priority;
(7)     $J_b$ inherits slowdown;
(8) endif
    On execution of each job $J_i$ :
(1) if ($mode = Sync$ and $\mathcal{P}(J_i) \leq MarkedPrio$)
(2)     $mode \leftarrow Indep$;
(3)     $MarkedPrio \leftarrow +\infty$;
(4) endif
(5) if $(mode = Indep)$ setSpeed ($\eta_i^I$);
(6) else setSpeed ($\eta_i^S$);
(7) endif
    On completion of blocking critical section of $J_i$ :
(1) Update speed of $J_i$;
(2) Update priority of $J_i$;
```

Figure 2. Dual Mode (DM) Algorithm

chronization mode. Otherwise, we say that the job executed in independent mode. Note that some critical sections of this job can execute in the synchronization mode.

Theorem 3 *: A task set of n periodic tasks, sorted in non-increasing order of their priority, can be feasibility scheduled with the Dual Mode algorithm with slowdown factors of η_i^I and η_i^S for each task τ_i if there exists scheduling points $S_{ij}^I \in S_i$ and $S_{ij}^S \in S_i$ for each task τ_i such that:*

$$\overset{\forall i}{i = 1, ..., n} \quad \sum_{k=1}^{i} \frac{1}{\eta_k^I} C_k \lceil \frac{S_{ij}^I}{T_k} \rceil \leq S_{ij}^I \tag{8}$$

$$\overset{\forall i}{i = 1, ..., n} \quad \frac{1}{\eta_i^S} B_i + \sum_{k=1}^{i} \frac{1}{\eta_k^S} C_k \lceil \frac{S_{ij}^S}{T_k} \rceil \leq S_{ij}^S \tag{9}$$

$$\forall i \quad \eta_i^S \geq \eta_i^I \tag{10}$$

Proof: We prove the above by contradiction. Suppose the claim is false and let t be the earliest time that a task misses its deadline. Let this task be τ_i and t' be the arrival time of the task instance. Let $\mathcal{A} \subseteq \{\tau_1, ...\tau_i\}$ be the set of jobs that execute in $[t', t]$ with priority greater than or equal to that of τ_i. Since τ_i is pending at all times during the interval $[t', t]$, the system in never idle in the interval. By the RM priority assignment, the only jobs that are allowed to *start* in $[t', t]$ are in $\mathcal{A}$. If a job not present in $\mathcal{A}$ executes in $[t', t]$, it must be holding some resource allocation at time t' that is blocking a job in $\mathcal{A}$. The stack resource protocol ensures that at most one job not in $\mathcal{A}$ executes in the interval $[t', t]$. Let this lower priority job that execute in $[t', t]$ be denoted by J_b. We consider both cases, with and without the blocking job J_b.

Case I: Only jobs in $\mathcal{A}$ are executed during $[t', t]$. Consider the sub-interval $[t', t' + S_{ij}^I]$ where S_{ij}^I is the scheduling point in Equation 8. The number of executions of each task $\tau_k \in \mathcal{A}$ in an interval of length S_{ij} is bounded by $\lceil \frac{S_{ij}^I}{T_k} \rceil$. Since the slowdown factors satisfy Equation 10, the slowdown of each task is at least η_i^I during the entire interval. Since a task misses its deadline at time t, the execution time for the jobs in $\mathcal{A}$ exceeds the interval length S_{ij}^I. Therefore,

$$\sum_{k=1}^{i} \frac{1}{\eta_k^I} (\lceil \frac{S_{ij}^I}{T_k} \rceil) C_k > S_{ij}^I$$

which contradicts with Equation 8.

Case II: Let J_b be the job blocking a job in $\mathcal{A}$. J_b is holding a resource allocation at time t' and the processor demand is larger than that in the first case due to the execution of the blocking job J_b. The total length of the time that J_b executes in $[t', t]$ is bounded by its outermost critical section. In particular, the maximum execution time of J_b in $[t', t]$ at full speed is bounded by B_i [5]. By the inheritance property, the blocking critical section inherits the maximum *blocking slowdown factor* over the jobs blocked in the system. The slowdown for the blocking critical section of J_b is at least η_i^S and the blocking time is bounded by $\frac{1}{\eta_i^S} B_i$. Only the outermost critical section of J_b and the tasks in $\mathcal{A}$ execute in the interval $[t', t]$. All tasks in the interval execute in the synchronization mode at a slowdown of η_k^S for task $\tau_k \in A$. Consider the subinterval $[t', t' + S_{ij}^S]$. The number of jobs of each task in $\mathcal{A}$ in this interval is bound by $\lceil \frac{S_{ij}^S}{T_j} \rceil$. Since τ_i does not complete by S_{ij}^S, the total workload of the jobs in $\mathcal{A}$ and J_b exceeds the interval S_{ij}^S. Thus

$$\frac{1}{\eta_i^S} B_i + \sum_{k=1}^{i} \frac{1}{\eta_k^S} (\lfloor \frac{S_{ij}^S}{T_k} \rfloor) C_k > S_{ij}^S$$

which contradicts with Equation 9.

3.3 Dynamic Slowdown

The static slowdown factors are computed assuming the worst case execution times for the tasks. Usually, tasks complete their execution earlier than their worst case execution time resulting in slack at run-time. This slack can be used to further reduce the processor speed to result in further energy gains. In this section, we present a dynamic

```
    On arrival of a new job $J_i$ :
     Perform the following in addition
     to the DM algorithm
(1)     $R_i^r(t) \leftarrow \frac{C_i}{\eta_i^I}$;
    On execution of each job $J_i$ :
(1) if ($mode = Sync$ and $\mathcal{P}(J_i) \leq MarkedPrio$)
(2)     $mode \leftarrow Indep$;
(3)     $MarkedPrio \leftarrow +\infty$;
( ) endif
(4) if ( $J_i$ is executed the first time
        and $mode = Sync$ )
(5)     Add to FRT-list($R_i^D, MarkedPrio$);
(6)     $R_i^r(t) \leftarrow \frac{C_i}{\eta_i^S}$;
( ) endif
(7) if ( Jobs blocked on $J_i$ )
(8)     setSpeed ($\frac{E_i^{CS}(t)}{R_i^{CS}(t)+R_i^F(t)}$);
( ) else
(9)     setSpeed ($\frac{E_i^r(t)}{R_i^r(t)+R_i^F(t)}$);
( ) endif
    On completion of $J_i$ :
(1) Add to FRT-list($R_i^r(t), \mathcal{P}(J_b)$);
```

Figure 3. Dual Mode Dynamic Reclamation (DMDR) Algorithm

slack reclamation scheme that works with task synchronization. The algorithm is an extension to the Dual Mode algorithm and is termed as the Dual Mode Dynamic Reclamation (DMDR) algorithm. It also leads to an enhancement of the DSDR algorithm [23], by applying dynamic slowdown to both non-blocking as well as blocking sections of a task.

We define *run time* of a job as the time budget assigned to the job. If e is the execution time of the job and the computed static slowdown is η, the run time of e/η is assigned to the job. Each run time also has a *priority* associated with it. The priority is used for reclaiming the unused run time of other jobs. The priority of the *run time* remains the same till the time budget is depleted.

When a job arrives in the system, each job is assigned a run time based on its static slowdown factor. The assigned run time for each job has a priority equal to the job priority. When a job completes earlier than the worst case execution time, the unused run time is free to be used by other jobs. Unused run time of jobs is maintained in a list called the *Free Run Time list (FRT-list)* [23]. When a job completes earlier than its WCET, the unused budget is added to the list. To ensure that the task meets its deadline, a task can use the free run time with a priority no smaller than its priority (free run time of a completed higher priority jobs). The dynamic slowdown factor is determined by the the total runtime that the job can utilize. Given the run time R and workload of e time units at maximum speed, a dynamic slowdown factor of e/R can be used to utilize the entire run time.

We use the same notation and definitions used in [23].

- J_i : the current job on 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 that can be utilized by J_i.
- $E_i^r(t)$: the residual execution time of job J_i.
- $E_i^{CS}(t)$: the residual execution time of the current critical section of J_i.
- $R_i^{CS}(t)$: the alloted run time for the residual critical section $(E_i^{CS}(t))$ at the inherited slowdown factor.
- $R_i^D = \frac{C_i}{\eta_i^I} - \frac{C_i}{\eta_i^S}$: the difference in run time in independent mode and synchronization mode

The DMDR algorithm is given in Figure 3. There are two types of dynamic reclamations similar to the DSDR algorithm. (1) If a job executing for the first time begins in the synchronization mode, the difference in the run times between the two mode R_i^D is added to the *FRT-list* with the priority of $MarkedPrio$. (2) When a job completes earlier than its available run time, its free run time is added to the *FRT-list* with the same priority as the job priority. Since a higher priority job is blocked at its arrival, the notion of priority inheritance is implicitly followed by the tasks. We need priority inheritance for dynamic slack reclamation. A blocking task can only use the free run-time having a priority no smaller than the inherited priority. We use this slack to slowdown the blocking critical section of the task and not the entire task.

The below rules are used in DMDR algorithm:

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

The rules may only be applied where tasks are blocked/unblocked and on a context switch.

We have enhanced the DSDR algorithm with the following changes. (1) On completion of the critical section in the *Sync* mode, if there are no blocked tasks in the system and there is free slack with priority greater or equal to the current inherited priority of the blocked task, the system can switch back to independent mode. (2) The blocking CS can use all the available free slack. However, we distribute the slack proportionately between the current critical section and the highest priority blocked job (i.e. we only use a portion of the free slack). This leaves some free run time for the blocked job for dynamic slowdown.

Theorem 4 *: Given a periodic task set is scheduled by the DSDR algorithm, all tasks complete their execution before their deadline.*

We prove the above by contradiction. The details of the proof are given in [9].

4 Computing Slowdown Factors

Computation of slowdown factors when tasks have identical power characteristics is considered in [7] and [23]. Computing slowdown factors based on the task power characteristics minimizes the total system energy consumption. In this section, we formulate the problem of computing the task slowdown factors as an optimization problem.

4.1 Power Delay Characteristics

The number of cycles, C_i that a task τ_i needs to complete is a constant during voltage scaling. The processor cycle time, the task delay and the dynamic power consumption of a task vary with the supply voltage V_{DD}. The power delay characteristics of the CMOS technology [21] are as given below.

$$P_{dynamic} = C_{eff} V_{DD}^2 f \tag{11}$$

$$Cycle\ Time\ (CT)\ \alpha \frac{1}{f} = k' \frac{V_{DD}}{(V_{DD} - V_{TH})^{\alpha}} \tag{12}$$

where k' is a device related parameter, V_{TH} is the threshold voltage, C_{eff} is the effective switching capacitance per cycle and α ranges from 2 to 1.2 depending on the device technology. The slowdown factor is the inverse of the cycle time and $\eta = 1/CT$.

4.2 Convex Minimization Problem

We formulate the energy minimization problem as an optimization problem. The voltage and slowdown factors are normalized to the maximum values. We compute normalized voltage levels for the tasks such that the conditions in Theorem 3 are satisfied. Let $\vec{v} \in \mathbb{R}^{2n}$ be a vector representing the normalized voltages V_i^S and V_i^I of task τ_i. V_i^S represents the task voltage in the synchronization mode and V_i^I represents the task voltage in independent mode. The optimization problem is to compute the optimal vector $\vec{v^*} \in \mathbb{R}^{2n}$ such that the system is feasible and the total energy consumption of the system is minimized. Let $f_i(V)$ be the normalized energy consumption of task τ_i as a function of the normalized voltage V (for the case of identical power characteristics $f_i(V) = V^2$). Since some jobs execute in synchronization mode and the rest in independent mode, the total energy consumption depends on the fraction of the jobs of each task that execute in synchronization mode. Let δ_i^S be the fraction of the total number of task instances of task τ_i that execute in synchronization mode, then $\delta_i^I = 1 - \delta_i^S$ is the fraction of the task instances executed in independent mode. The total energy consumption of the system E, a function of the voltage vector $\vec{v} \in \mathbb{R}^{2n}$, is given by Equation 13. Thus, we have the following optimization problem:

minimize :

$$E(v) = \sum_{i=1}^{n} \delta_i^I \cdot f_i(V_i^I) \cdot \frac{C_i}{\eta_i^I T_i} + \sum_{i=1}^{n} \delta_i^S \cdot f_i(V_i^S) \cdot \frac{C_i}{\eta_i^S T_i} \tag{13}$$

under the constraints :

$$\overset{\forall i}{i = 1, ..., n} \qquad \sum_{k=1}^{i} \frac{1}{\eta_k^I} C_k \lceil \frac{S_{ij}^*}{T_k} \rceil \leq S_{ij}^* \tag{14}$$

$$\overset{\forall i}{i = 1, ..., n} \qquad \frac{1}{\eta_i^S} B_i + \sum_{k=1}^{i} \frac{1}{\eta_k^S} C_k \lceil \frac{S_{ij}^*}{T_k} \rceil \leq S_{ij}^* \tag{15}$$

$$\forall i \qquad \eta_{min} \leq \eta_i^I \leq \eta_i^S \leq 1 \tag{16}$$

where η is a function of V given by $\eta = 1/CT$. Equation 14 ensures the feasibility when the tasks are independent. The scheduling point S_{ij}^I is the scheduling point with the minimum average workload in the interval $[0, S_{ij}]$ over all scheduling points. Similarly, the scheduling point S_{ij}^S is the interval with the minimum workload considering the blocking time. Since we are computing the scheduling point in this manner, we have $S_{ij}^I = S_{ij}^S = S_{ij}^*$ as seen in the problem formulation. Equation 15 enforces the feasibility of the task set in the presence of task synchronization. Equation 16 constraints the slowdown in the synchronization mode to be greater than or equal to the slowdown in the independent mode. The normalized slowdown factors are between the normalized minimum frequency η_{min} and 1. The cycle time at voltage V_i is given in Equation 12. The constraint given by Equations 14, 15, and 16 are convex [9]. The optimization function depends on the power characteristics $f_i(V)$ of the task. For all power characteristics, such that $\frac{f(V)}{\eta(V)}$ (note that η is a function of V) is convex, the optimization function is convex. Thus we have a convex minimization problem.

The number of tasks executed in each mode depends on the task slowdown factors. We do not know the initial value of δ_i^S to be used in the optimization function. Initially, we assume $\delta_i^S = 0.05$ and compute slowdown factors. We simulated the task set at the computed slowdown factors to get δ_i^S values from the simulation. The updated δ_i^S values did not change the task slowdown factors. In our experiments we assume $\delta_i^S = 0.05$ in the optimization function. We could use a iterative loop of computation of slowdown factors and updating the δ_i^S values obtained from simulation.

Given there are n tasks in the system, the number of variables is $2n$ and the number of constraints is $3n$. Thus the number of variables and constraints are linear in the problem size.

5 Experimental Setup

We have written a simulator in *parsec* [13], a C based discrete event simulation language, wherein we have implemented the scheduling policy and the slowdown algorithms. Simulation experiments were performed to evaluate our proposed technique with task sets of 10-15 tasks. We used a mixed workload with task periods belonging to one of the three period ranges [2000,5000], [500,2000] and [90,200]. The WCET's (worst case execution times) for the three ranges were [10,500], [10,100] and [10,20] respectively. The tasks were uniformly distributed in these categories with the period and WCET of a task randomly selected within the corresponding ranges. The number of semaphores (within 0 to 2) and the position of the critical sections within each task execution were selected randomly. The length of the critical sections were chosen to be $CSperc * WCET$, where $CSperc$ is the size of the critical section as a percentage of the WCET. We vary $CSperc$ up to 30% of the WCET in steps of 3%. Task sets are generated with a utilization between 50% to 75% at maximum speed.

Due to the diverse nature of task sets, tasks can have varying power characteristics [3], and it is energy efficient to compute slowdown factors based on the task characteristics. For experimental results we restrict power characteristics to be *linear*, where we vary the switching capacitance C_{eff} for the task. We say a task has a *power coefficient* k to represent a task with a switching capacitance k times the base case. However, the problem formulation in Section 4 works for all power characteristics such that $f(V)/\eta$ is convex and differential. We consider the following distributions similar to the ones presented by Aydin *et al.* [3]: (1) **Identical Distribution**: where all tasks have the same power coefficient. (2) **Bimodal Distribution**: represents the case where there are two types of tasks in the system, with 50% having a low power coefficient of 1 and the others having a high power coefficient k. (3) **Uniform Distribution**: where the coefficients of the power function of the tasks are uniformly distributed between 1 and k.

We vary k in the range $[1, 4]$ for experimental results. The energy and delay characteristics are given by Equations 11 and 12. We have used an operating voltage range of $0.6V$ and $1.8V$. The threshold voltage is assumed to be $0.36V$ and $\alpha = 1.5$. We have normalized the operating speed and support discrete voltage levels in steps of 0.05 in the normalized range. We compare the energy gains of the *Dual Mode (DM)* algorithm over the Dual Speed(DS) algorithm [23]. The dual speed algorithm has two speeds of H and L corresponding to the synchronization and independent mode respectively. We show the average gains of DM over DS, where the average is taken over 5 to 10 different task sets.

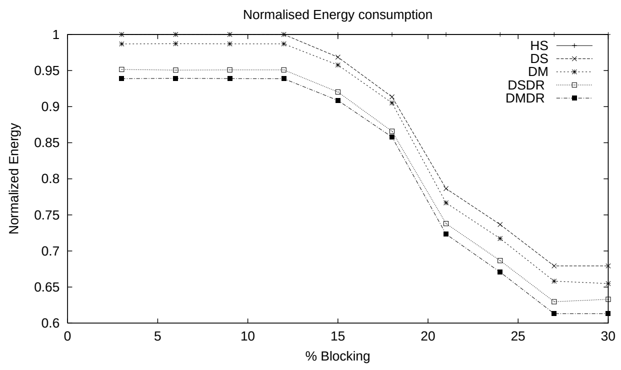

Figure 4. Average percentage energy savings of DM (DMDR) algorithm over the DS (DSDR) algorithm for identical power characteristics.

5.1 Identical power characteristics

In the first set of experiments, we generate tasks with identical power characteristics and vary the blocking percentage. Figure 4 shows the percentage gains of DM algorithm over the DS algorithm. The energy consumption of both algorithms are normalized to the High Speed (HS)[23] algorithm, where the tasks are always executed in the H speed of the dual speed algorithm. From the figure, it is seen that the DM algorithm performs better than the DS algorithm. Blocking is not significant up to 12% and the energy consumption is close to that of the HS algorithm. With increased blocking, DM and DS perform better than HS. DM assigns different slowdown factors for the tasks to have better energy gains. DM performs up to 4% better than DS at 30% of blocking. Since we compute continuous slowdown factors and assign them to the closest discrete voltage levels, there is run time slack even at worst case execution time. The figure also compares the energy gains of the dynamic reclamation schemes. Similar behavior is observed when comparing DMDR to DSDR. Comparing DM with DMDR, one sees that the dynamic reclamation adds up to 5% of energy savings.

It is beneficial to have different slowdown factors based on the power characteristics of the tasks. With tasks having different slowdown factors, the property of frequency (slowdown) inheritance is essential to guarantee task deadlines. A task may inherit a higher slowdown factor than its assigned slowdown factor. Executing parts of the tasks at high speed consumes additional energy. This leads to an additional energy overhead which is we refer to as the *inheritance overhead*. The inheritance overhead is not a part of the optimization function. However, is not clear how the optimization function can take this overhead into account. It is not clear which job instances will inherit a higher slowdown, what slowdown factor will be inherited and how long they will inherit a higher slowdown.

5.2 Varying power characteristics

In the second set of experiments, we vary the task power characteristics and compute slowdown factors. Figure 5 shows the percentage gains of the DM algorithm over the DS algorithm. The execution time of all the tasks are set to their WCET. It is seen that the energy gains over DS increase with the power coefficient k. Computing task slowdown factors considering the power characteristics, results in more energy gains. The energy gains also increase along with the blocking percentage. The gains steadily increase with blocking with some drops in gains due to inheritance overhead. From the graph of energy gains under bimodal distribution, it is seen that the gains are appreciable even when the blocking is not significant (3% to 9%). For larger blocking percentages, the gains are as high as 13% over the DS algorithm. Figure 5 also shows the energy gains for the case of uniform distribution of the task power coefficient. Since all tasks have a random power coefficient in the range $[1, k]$, there is relatively less variation in the task power characteristics compared to the bimodal distribution. This leads to less energy gains compared to bimodal distribution. The overall gains are reduced to a maximum of 8% as opposed to 13% with bimodal distribution. However, the DM algorithm presents significant energy gains over the DS algorithm. Note that the energy gains are over the DS algorithm, thus leading to large total energy savings.

5.3 Dynamic Slowdown

In this section, we compare the energy gains of the DMDR algorithm over the DSDR algorithm by varying the task execution times. We vary the *best case execution time (BCET)* of a task as a percentage of its WCET. Task execution times were 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%.

Figure 6 shows the energy gains of DMDR over DSDR as the BCET is varied. Task sets were generated using both bimodal and uniform distribution with a power coefficient of $k = 2.5$. It is seen that the gains increase as BCET decreases. The main source for energy gains is the computation of static slowdown factors based on the task power characteristics. Since we slowdown the blocking critical sections, we gain additional energy savings than the DSDR algorithm. This is also evident from the fact that the gains increase with an increase in the blocking percentage. Due to the inheritance overhead, the gains do not increase steadily with the increase in blocking percentage. Energy savings are as high as 7%, for the case of bimodal distribution. Figure 6 also shows the gains for the case of uniform distribution. Similar behavior as that for the bimodal distribution is observed. Since the energy gains are over the DSDR algorithm, the total energy gains are large.

6 Conclusions and Future Work

In this paper, we present algorithms to compute static and dynamic slowdown in the presence of task synchronization. Our framework allows tasks to have different slowdown factors depending on the task characteristics. Similar to priority inheritance, we introduce the notion of *frequency inheritance* to guarantee task deadlines. We present the dual mode algorithm under the RM scheduling policy. We prove that it is sufficient to execute in the synchronization mode for a shorter interval than that presented in the previous work [23] to further reduce the energy consumption. We formulate the computation of slowdown factors for tasks with different power characteristics as an optimization problem. Experimental results show that the computed slowdown factors save on an average 5-10% energy over the known techniques. The techniques are energy efficient and can be easily implemented in an RTOS. This will have a great impact on the energy utilization of portable systems.

We plan to further exploit the static and dynamic slack in the system to make the system more energy efficient. As a future work, we plan to compute discrete slowdown factors for the tasks.

Acknowledgments

The authors acknowledge support from National Science Foundation (Award CCR-0098335) and from Semiconductor Research Corporation (Contract 2001-HJ-899). We would like to thank Cristiano Pereira for the helpful discussions and the reviewers for their useful comments.

References

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

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

[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 *Euromicro Conference on Real-Time Systems*, June 2001.

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

[5] T. P. Baker. Stack-based scheduling of realtime processes. In *RealTime Systems Journal*, pages 67–99, 1991.

[6] F. Gruian and K. Kuchcinski. Lenes: task scheduling for low-energy systems using variable supply voltage processors. In *ASP-DAC*, 2001.

[7] R. Jejurikar and R. Gupta. Energy aware edf scheduling with task synchronization for embedded real time operating systems. In *COLP*, 2002.

[8] R. Jejurikar and R. Gupta. Energy aware task scheduling with task synchronization for embedded real time systems. In *International Conference on Compilers Architecture and Synthesis for Embedded Systems*, 2002.

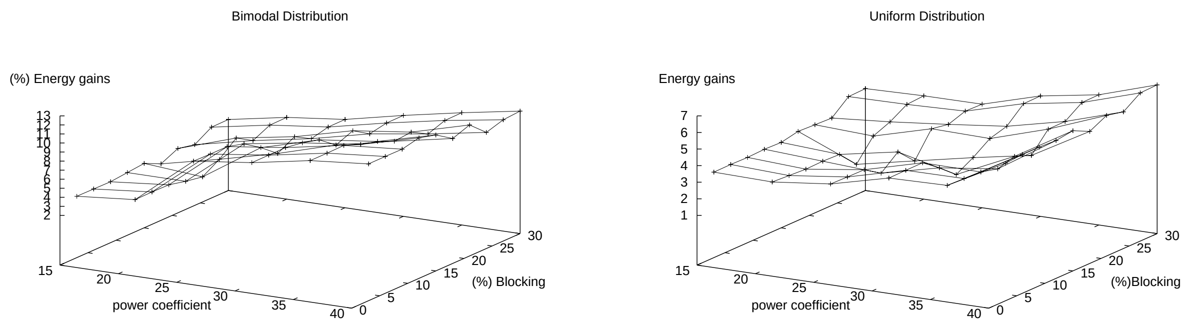

Figure 5. Average percentage gains of DM algorithm over DS algorithm.

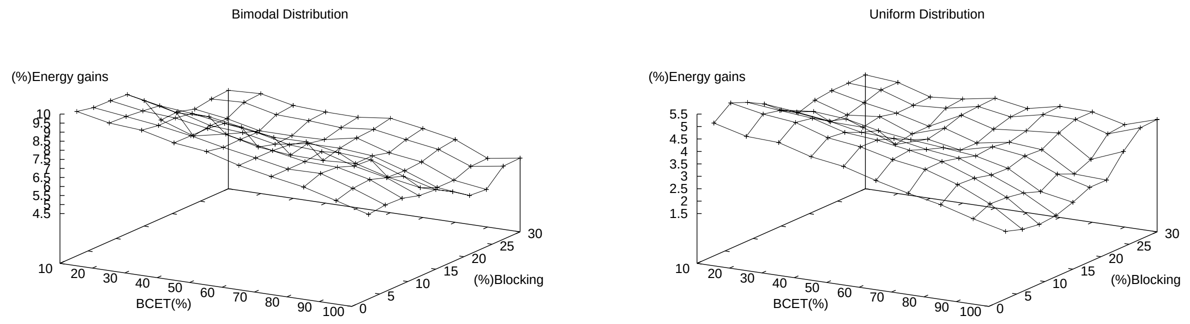

Figure 6. Percentage gains of DMDR algorithm over DSDR algorithm for power coefficient $k = 2.5$

[9] R. Jejurikar and R. Gupta. Dual mode algorithm. In *CECS Technical Report #02-36, University of California Irvine*, Aug. 2003.

[10] W. Kim, J. Kim, and S. L. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *DATE*, 2002.

[11] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behaviour. In *Real-Time Systems Symposium*, pages 166–171, 1989.

[12] J. Luo and N. Jha. Power-conscious joint scheduling of periodic task graphs and a periodic tasks in distributed real-time embedded systems. In *ICCAD*, 2000.

[13] Parallel Computing Laboratory. Parsec: A c-based simulation language. University of Califronia Los Angeles. http://pcl.cs.ucla.edu/projects/parsec.

[14] 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.

[15] G. Quan and X. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the Design Automation Conference*, pages 828–833, June 2001.

[16] 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.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on Computers*, pages 1175–85, 1990.

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

[19] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., 2001.

[20] J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. In *IEEE Transactions on Computers*, 1994.

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

[22] F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *FOCS*, pages 374–382, 1995.

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

[24] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *DAC*, 2002.