Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems *

Hakan Aydin† Rami Melhem‡ Daniel Mossé§ Pedro Mejía-Alvarez¶

Abstract

In this paper, we address power-aware scheduling of periodic hard real-time tasks using dynamic voltage scaling. Our solution includes three parts: (a) a static *(off-line) solution to compute the optimal speed, assuming* worst-case *workload for each arrival, (b) an on-line speed reduction mechanism to* reclaim *energy by adapting to the* actual *workload, and (c) an on-line, adaptive and* speculative *speed adjustment mechanism to anticipate early completions of future executions by using the* average-case *workload information. All these solutions still guarantee that all deadlines are met. Our simulation results show that the reclaiming algorithm saves a striking 50% of the energy over the static algorithm. Further, our speculative techniques allow for an additional approximately 20% savings over the reclaiming algorithm. In this study, we also establish that solving an instance of the* static *power-aware scheduling problem is equivalent to solving an instance of the reward-based scheduling problem [1, 4] with concave reward functions.*

1 Introduction

In the last decade, the research community has addressed the low power system design problems with a multi-dimensional effort [7, 18]. Such on-going research has important implications for real-time systems design, simply because most of the applications running on power-limited systems inherently impose temporal constraints on the response time (such as real-time communication in satellites).

The *variable voltage scheduling* (VVS) framework, which involves dynamically adjusting the voltage and frequency (hence the CPU speed), has recently become a major research area for power-aware computer systems. For real-time systems, the proposed VVS schemes focus on minimizing energy consumption in the system, while still meeting the deadlines. Yao et al.[23] provided a static off-line scheduling algorithm, assuming aperiodic tasks and worst-case execution times. Heuristics for on-line scheduling of aperiodic tasks while not hurting the feasibility of periodic requests are proposed in [9]. Non-preemptive power aware scheduling is investigated in [8]. Concentrating on periodic tasks with identical periods, the effects of having an upper bound on the voltage change rate are examined in [10], along with a heuristic to solve the problem. Slowing down the processor whenever there is a single task eligible for execution was explored in [21]. Lorch and Smith addressed the variable voltage scheduling of tasks with *soft* deadlines in [14]. The static solution for the general periodic model where tasks have potentially different power consumption characteristics is provided in [2].

However, most of the scheduling schemes presented in these studies, while using exclusively worst-case execution time (WCET) to guarantee the timeliness of the system, lack the ability to dynamically take advantage of unused computation time. In fact, real-time applications usually exhibit a large variation in *actual* execution times; for example [5] reports that the ratio of the worst-case execution time to the best-case execution time can be as high as 10 in typical applications.

Consequently, *dynamically monitoring and reclaiming the 'unused' computation time* can be (and, as we show later in this paper, is in fact) a powerful approach to obtain considerable power savings and to minimize the effects of designing the system with WCET information, which is usually a very conservative prediction of the *actual* execution time. Additional improvements are possible thanks to the *statistical* workload information; in this paper, we investigate also *aggressive* schemes where we *anticipate* the early completions of *future* executions and *speculatively* reduce the CPU speed. This approach immediately raises two intertwined questions, namely, (a) the *level* of aggressiveness under a given probability distribution of actual workload; and (b) the issue of guaranteeing the timing constraints even in aggressive modes. It is obvious that the solutions to these problems should be simultaneously practical and efficient, in order to be applicable on-line.

*This work has been supported by the Defense Advanced Research Projects Agency through the PARTS project (Contract F33615-00-C-1736).

†Computer Science Department, George Mason University, Fairfax VA 22030. E-mail: aydin@cs.gmu.edu. Work done while the author was with the University of Pittsburgh.

‡Computer Science Department, University of Pittsburgh, Pittsburgh PA 15260. E-mail: melhem@cs.pitt.edu.

§Computer Science Department, University of Pittsburgh, Pittsburgh PA 15260. E-mail: mosse@cs.pitt.edu.

¶CINVESTAV-IPN. Sección de Computación, Av. I.P.N. 2508, Zacatenco, México, DF 07300. E-mail: pmejia@computacion.cs.cinvestav.mx. Work done while the author was visiting the University of Pittsburgh.

0-7695-1420-0/01 $17.00 © 2001 IEEE

It goes without saying that dynamic reclaiming and/or aggressive techniques should preserve the feasibility of the task system (i.e., no deadline should be missed), even under a worst-case scenario that may take place after *any* speed adjustment decision.

We must note that a recent study [11] addressed dynamic energy reclaiming issues (without speculation) in power-aware scheduling for cyclic and periodic task models, in the context of systems with two (discrete) voltage levels. However, systems which are able to operate on a (more or less) continuous voltage spectrum are rapidly becoming a reality thanks to advances in power-supply electronics and CPU design [6, 17]. For example, the Crusoe processor is able to dynamically adjust clock frequency in 33 MHz steps [22]. To the best of our knowledge, the concept of "speculative speed reduction" was first introduced by the authors in [16]; however, only tasks sharing a common deadline were considered.

Paper Organization

In this paper, we identify and address three dimensions of power-aware scheduling for real-time systems and develop efficient algorithms for the periodic task model. Effectiveness in reducing the energy consumption can be improved only by a simultaneous consideration of these three dimensions, since they complement each other. Thus, we present:

1. A static (off-line) solution to compute the optimal speed at the task level, assuming worst-case workload for each arrival[1] (Section 3). In the same section, we also show that solving an instance of the the *static* power-aware real-time scheduling problem is equivalent to solving an instance of the reward-based scheduling problem [1].
2. An on-line speed adjustment mechanism to dynamically reclaim energy not used by tasks that complete without consuming their worst-case workload (Section 4).
3. An on-line, adaptive and speculative speed adjustment mechanism to anticipate and compensate probable early completions of future executions (Section 5).

We emphasize once again that, in the context of real-time systems, all these components should be designed not to cause any deadlines to be missed even under the worst-case scenario: the aim is to **meet the timing constraints while simultaneously and dynamically reducing power consumption as much as possible**.

2 System Model and Notation

The ready time and deadline of each real-time task T_i will be denoted by r_i and d_i, respectively. The indicator of the worst-case workload in variable voltage/speed settings, that is, the worst-case number of processor cycles required by T_i, will be denoted by C_i. Note that, under a constant speed S (given in cycles per second), the execution time of the task T_i is $t_i = \frac{C_i}{S}$. A schedule of real-time tasks is said to be **feasible** if each task T_i receives at least AC_i CPU cycles before its deadline, where $AC_i \leq C_i$ is the *actual* number of CPU cycles (actual workload) of T_i.

We assume that the CPU speed can be changed between a minimum speed S_{min} (minimum supply voltage necessary to keep the system functional) and a maximum speed S_{max}, and that $0 \leq S_{min} \leq S_{max} = 1$; that is, we normalize the speed values with respect to S_{max}. In our framework, the voltage/speed changes take place only at context switch time and while state saving instructions execute. Pouwelse et al. report in [19] that the voltage/speed change can be performed in less than 140 μs in Strong ARM SA-1100 processor. If not negligible, the 'voltage change overhead' can be incorporated into the worst-case workload of each task.

We assume that the process descriptor of the task T_i has two extra fields related to speed settings, in addition to other conventional fields. The first one, S_i, denotes the *current* CPU speed at which T_i is executing. The other field $\widehat{S}_i$ denotes the *nominal* speed of T_i, which is the indicator of the "default" speed of T_i. For each task that is dispatched, the operating system sets $S_i = \widehat{S}_i$, prior to any dynamic speed adjustment.

The power consumption of the processor under the speed S is given by $g(S)$, which is assumed to be a strictly increasing convex function, represented by a polynomial of at least second degree [10]. If the task T_i occupies the processor during the time interval $[t_1, t_2]$, then the *energy* consumed during this interval is $E(t_1, t_2) = \int_{t_1}^{t_2} g(S(t))dt$.

In our detailed analysis of periodic power-aware scheduling, we will consider a set $\mathcal{T} = \{T_1, \ldots, T_n\}$ of n periodic real-time tasks. The period of T_i is denoted by P_i, which is also equal to the deadline of the current invocation. We refer to the j^{th} invocation of task T_i as $T_{i,j}$. All tasks are assumed to be independent and ready at $t = 0$. Hence, the ready time of $T_{i,j}$ is $r_{i,j} = (j-1) \cdot P_i$, and its deadline is $d_{i,j} = j \cdot P_i$.

We define U_{tot} as the total utilization of the task set under maximum speed $S_{max} = 1$, that is, $U_{tot} = \sum_{i=1}^{n} \frac{C_i}{P_i}$. Note that the schedulability theorems for periodic real-time tasks [12] imply that $U_{tot} \leq 1$ is a necessary condition to have at least one feasible schedule; hence, throughout the paper, we will assume that $U_{tot} = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$.

3 Optimal Static Solution

3.1 The Reward-Based Approach to Power-Aware Scheduling

Before analyzing the periodic model in depth, we correlate *the reward-based scheduling* [1, 3] framework to the power-aware scheduling of real-time tasks. The reward-based scheduling framework encompasses real-time scheduling models such as Imprecise Computation [13] and Increased-

[1] Due to the nature of VVS, the actual execution time is dependent on the CPU speed, and therefore the worst-case number of required CPU cycles is a more appropriate measure of the worst-case workload (see Section 2).

Reward-with-Increased-Service [4] that exploit the timeliness and precision trade-off. We underline that the correlation that we prove is preserved regardless of the task model (aperiodic/periodic or preemptive/nonpreemptive), as long as our aim is to reach a solution for a given (worst-case) workload.

In the reward-based scheduling framework, each real-time task T_i comprises a mandatory part M_i and an optional part O_i. The worst-case execution times of M_i and O_i are denoted by m_i and o_i, respectively. The mandatory part runs first, producing an output of acceptable quality, which is subsequently enhanced by the optional part within the limits of available computational capacity. To quantify the quality improvement, a non-decreasing reward function $F_i(t_i)$ is associated with each optional execution where $t_i \leq o_i$ denotes the service time O_i receives. Most of the realistic applications are best represented by concave reward functions [1, 3, 4, 20]. In any *feasible* reward-based schedule, *each* mandatory part must be fully executed by the task deadline d_i, however, the optional parts may remain partially executed by the deadlines. Now we can formally define the reward-based scheduling problem.

Reward-Based Scheduling Problem: Consider the uniprocessor scheduling of a reward-based real-time task set $\mathbf{T} = \{T_1, \ldots, T_n\}$. Given a time point Δ_f, determine the *optimal schedule* in the interval $[0, \Delta_f]$, where each mandatory part M_i complete in a timely fashion before the task deadline d_i, and each optional part receives service for $t_i \leq o_i$ units of time so as to maximize the total system reward $\sum_i F_i(t_i)$.

The determination of the optimal schedule clearly involves the computation of optimal optional service times. Noting that the reward accrued by each optional part O_i does not increase beyond the upperbound o_i, this computation can be expressed as an optimization problem where the objective is to find t_i values[2] so as to:

$$\text{maximize} \qquad \sum_{i=1}^{n} F_i(t_i) \tag{1}$$

$$\text{subject to} \qquad 0 \leq t_i \leq o_i \qquad i = 1, \ldots, n \tag{2}$$

$$\text{There exists a feasible schedule with } \{m_i\} \text{ and } \{t_i\} \text{ values} \tag{3}$$

On the other hand, the real-time power-aware scheduling problem can be stated as follows.

Real-Time Power-Aware Scheduling (RT-PAS) Problem: Consider a CPU with variable voltage/speed S ($S_{min} \leq S \leq S_{max}$) facility, where the power consumption is given by a strictly increasing convex function $g(S)$, which is a polynomial of at least second degree. Given a set $\mathbf{T} = \{T_1, \ldots, T_n\}$ of real-time tasks, in which each task T_i is subject to a worst-case workload of C_i expressed in the number of required CPU cycles, and a time point Δ_f, determine *the schedule* and *the processor speed* $S(t)$ so as to minimize the total energy consumption $E(0, \Delta_f) = \int_0^{\Delta_f} g(S(t))dt$ in the interval $[0, \Delta_f]$.

[2] When considering the periodic task model, the execution time of each task instance (t_{ij}) should be considered as a separate unknown.

Before relating two scheduling problems, we observe that the convexity of speed/power function allows us to deduce the following (a formal proof can be found in [3]).

Proposition 1 *One can safely commit to a constant CPU speed during the execution of a task T_i requiring C_i CPU cycles, without increasing the energy consumption.*

Note that the determination of S_i in the RT-PAS problem is now effectively equivalent to determining the CPU time allocation to T_i, which will be denoted by x_i ($x_i = \frac{C_i}{S_i}$). We are now ready the establish the connection between RT-PAS and Reward-Based Scheduling problems.

Proposition 2 *Solving an instance of RT-PAS problem is equivalent to solving an instance of Reward-Based Scheduling problem with concave reward functions.*

Proof: To prove the statement, we will first formulate the computation of optimal speed values as an optimization problem. The total energy consumption, thanks to the constant speed assumption per task, can now be expressed as $E_{tot} = \sum_{i=1}^{n} x_i \cdot g(S_i) = \sum_{i=1}^{n} x_i \cdot g(\frac{C_i}{x_i})$. Further, observe that the minimum and maximum speed bounds impose natural lower and upper bounds on CPU allocation of T_i. In other words, the inequality $\frac{C_i}{S_{max}} \leq x_i \leq \frac{C_i}{S_{min}}$ should be satisfied. Hence, the computation of optimal CPU allocation assignments can be formalized as an optimization problem:

$$\text{minimize} \qquad \sum_{i=1}^{n} x_i \cdot g(\tfrac{C_i}{x_i}) \tag{4}$$

$$\text{subject to} \qquad \tfrac{C_i}{S_{max}} \leq x_i \leq \tfrac{C_i}{S_{min}} \quad i = 1, \ldots, n \tag{5}$$

$$\text{There exists a feasible schedule with } \{x_i\} \text{ values} \tag{6}$$

Now, consider the variable transformation $m_i = \frac{C_i}{S_{max}}$, $t_i = x_i - m_i$, $o_i = \frac{C_i}{S_{min}} - \frac{C_i}{S_{max}}$, and $F_i(t_i) = -(t_i + \frac{C_i}{S_{max}})g(\frac{C_i}{t_i + \frac{C_i}{S_{max}}})$.

This transformation can be interpreted as follows: First, T_i must be assigned at least $\frac{C_i}{S_{max}}$ units of CPU time ("mandatory" execution). Any allocation exceeding this minimum amount will be considered as "optional" execution, while the total CPU allocation $(m_i + t_i)$ can not exceed the upper bound $\frac{C_i}{S_{min}}$. Finally, the more we allocate CPU time to T_i by increasing t_i, the more we increase the energy savings thanks to the speed/power relation. It is not difficult to see that, by using the above transformation and by re-writing the optimization problem given by (4), (5) and (6), one reaches once again the formulation of the general reward-based scheduling problem defined by Equations (1), (2) and (3). Further, the reward function $F_i(t_i)$ above is clearly concave, since $(t_i + \frac{C_i}{S_{max}})g(\frac{C_i}{t_i + \frac{C_i}{S_{max}}})$ is convex. To see this, we can use the result from [15] stating that if a and b are both convex functions and if a is increasing, then $a(b(x))$ is also convex. Thus, by setting $h(t_i) = \frac{C_i}{t_i + \frac{C_i}{S_{max}}}$, and observing that the multiplication by $(t_i + \frac{C_i}{S_{max}})$ does not affect the convexity, we justify the concavity of $F_i(t_i) = -g(h(t_i))$. □

3.2 Specific Solution for Periodic Task Sets

In this section, we present the static optimal solution to the variable voltage scheduling problem for the periodic task model, *assuming that each task presents its worst-case workload to the processor at every instance*. We underline that one can use the equivalence obtained in Section 3.1 and the results from [1] to justify the proposition (as formally done in [3]); however, one can also reach the same conclusion by using the first principles as outlined below.

Proposition 3 *The optimal speed to minimize the total energy consumption while meeting all the deadlines is constant and equal to* $\bar{S} = max\{S_{min}, U_{tot}\}$. *Moreover, when used along with this speed* $\bar{S}$, *any periodic hard real-time policy which can fully utilize the processor (e.g., Earliest Deadline First, Least Laxity First) can be used to obtain a feasible schedule.*

Proof: First, observe that the convex nature of the power-speed function suggests that we should try to maintain a uniform speed while fully utilizing the CPU to the extent it is possible. If $U_{tot} \geq S_{min}$, then using the speed $\bar{S} = U_{tot}$ leads clearly to a schedule which is fully utilized (i.e., no idle time), through stretching out each task in equal proportions (in other words, in this case, we are achieving a total *effective* task utilization of $\sum_{i=1}^{n} \frac{C_i}{\bar{S} \cdot P_i} = \frac{U_{tot}}{\bar{S}} = 1$). However, if $U_{tot} < S_{min}$, then we should use the minimum CPU speed available, to stretch out task executions as much as possible. In any case, using the speed $\bar{S} = max\{S_{min}, U_{tot}\}$ will result in a total effective task utilization which is no greater than 1. Hence, any scheduling policy which can achieve up to 100% CPU utilization (Earliest Deadline First, Least Laxity First) can be used to complete all the task instances before their deadlines with the speed $\bar{S}$. □

4 Dynamic Reclaiming Algorithm

The dynamic reclaiming algorithm is based on detecting early completions and adjusting (reducing) the speed of other tasks *on-the-fly* in order to provide additional power savings while still meeting the deadlines. To this aim, we perform comparisons between the actual execution history and the canonical schedule $\mathcal{S}^{can}$, which is the static optimal schedule on which every instance presents its worst-case workload to the processor and runs at the constant speed $\bar{S}$. The CPU speed is adjusted only at task dispatch times: thus, we should be able to say whether the task is being dispatched *earlier* than $\mathcal{S}^{can}$, and if so, determine the amount of additional CPU time the dispatched task can *safely* use to slow down its execution; we will refer to this additional CPU time as the *earliness* of the dispatched task. Before providing the details of our approach, we underline that a simple approach that equates earliness with previously unused CPU time and and *blindly* slows down the processor is *not* a safe approach. To see this, consider a 3-task system with the following parameters: $C_1 = 4, P_1 = 10, C_2 = 4, P_2 = 10, C_3 = 6, P_3 = 30$. The worst-case utilization of the task set is equal to 1.00, hence the optimal speed for the static version is $\bar{S} = S_{max} = 1.00$ (from Proposition 3). If every task presents its worst-case workload at every instance and we use EDF, then the schedule in Figure 1 ($\mathcal{S}^{can}$) would be obtained. Now, suppose that T_3 completes

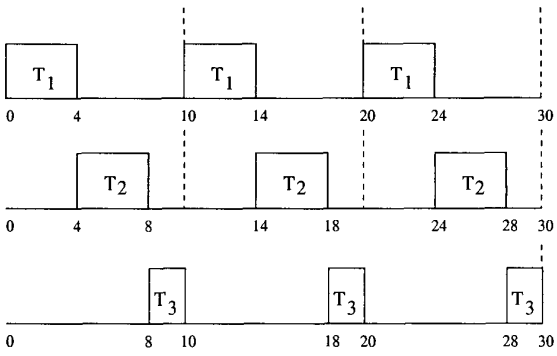

Figure 1. The static optimal schedule, $\mathcal{S}^{can}$

early at $t = 10$, leaving an unused computation time of 4 units before its deadline. If these 4 units of CPU time are used by $T_{1,2}$ (recall that $T_{i,j}$ is the j^{th} instance of task i), $T_{2,2}$ will miss its deadline, if both $T_{1,2}$ and $T_{2,2}$ require their worst-case workload.

As we can see, computing and managing earliness is not a trivial task. Due to the periodic nature of the tasks we consider, it is clearly impractical to a priori produce and keep the entire static optimal schedule $\mathcal{S}^{can}$ during the execution. In order to simultaneously address the problems of feasibility and efficiency, while tasks execute, complete, re-arrive dynamically and the actual schedule is produced, we choose to keep and update a data structure (called **α-queue)** that helps to compute the earliness of tasks when they are dispatched. At any time t during actual execution, the α-queue contains information about tasks that would be active (i.e., running or ready) at time t in the worst-case static optimal schedule $\mathcal{S}^{can}$ (in other words, α-queue is the ready queue of $\mathcal{S}^{can}$ at time t). We assume that the following information can be obtained for each task from the α-queue at any time t:

- i, the identity of the task (i.e., task number),
- $r_{i,j}$, the arrival time of the instance (i.e., the period boundary earlier than t),
- $d_{i,j}$, the deadline of the instance (i.e., the period boundary later than t), and
- $rem_{i,j}(t)$, the remaining execution time of $T_{i,j}$ at time t in $\mathcal{S}^{can}$, under the static optimal speed $\bar{S}$.

Clearly, given t, the $r_{i,j}$ and $d_{i,j}$ values can be easily computed for the periodic task model. Note that the α-queue at time t contains information about all instances $T_{i,j}$ such that $r_{i,j} \leq t \leq d_{i,j}$, and $rem_{i,j}(t) > 0$. The α-queue contains at most n elements, since the number of tasks in the ready queue

can never exceed the total number of tasks in any schedule. Therefore, we will omit the instance number while referring to α-queue elements, whenever clarity is not compromised.

Our approach assumes that tasks are scheduled according to EDF* policy. EDF* is the same as EDF (Earliest Deadline First [12]), except that, among tasks whose deadlines are the same, the task with the earliest arrival time has the highest priority (FIFO policy); in case that both deadline and arrival times are equal, the task with the lowest index has the highest priority. This EDF* *priority ordering* is essential in our approach because it provides a total order on the priorities. Further, we assume that the α-queue is also ordered according to EDF* priorities. We denote the EDF* priority-level of the task i by d_i^* (low values denote high priorities).

At this point, we are ready to relate the α-queue with the computation of earliness factor. Let $w_i^S(t)$ denote the remaining worst-case execution time of task T_i under the speed S at time t. Further, set the nominal speed $\widehat{S}_i = \bar{S}$ for each task T_i.

Proposition 4 *For any task T_x which is about to execute, any unused computation time (slack) of any task in the α-queue having strictly higher priority than T_x will contribute to the earliness of T_x along with already finished work of T_x in the actual schedule. That is,* total earliness *of T_x is no less than* $\epsilon_x(t) = \sum_{i|d_i^* < d_x^*} rem_i(t) + rem_x(t) - w_x^{\widehat{S}_x}(t) = \sum_{i|d_i^* \leq d_x^*} rem_i(t) - w_x^{\widehat{S}_x}(t)$.

To understand the above result, note that when T_x is being dispatched, tasks with higher priority that are still in the α-queue must be already finished in the actual schedule (since T_x currently has the highest EDF* priority), but they would have not yet finished in $\mathcal{S}^{can}$.

Implementing the α-queue: The α-queue can be easily implemented using the following rules:

R1. Initially the α-queue is empty.

R2. Upon arrival, each task T_j "pushes" its worst-case execution time under nominal speed $\widehat{S_j} = \bar{S}$ to the α-queue in the correct EDF* priority position (this happens only once for each arrival, no re-push at 'return from preemptions').

R3. As time elapses, the elements in the α-queue are updated (consumed) accordingly: the $rem_{i,j}$ field at the head of α-queue is decreased with a rate equal to that of the passage of time. Whenever the $rem_{i,j}$ field of the head reaches zero, that element is removed from α-queue and the update continues with the next element. No update is done when the α-queue is empty.

Observation 1 *At time t, the α-queue, updated according to the rules R1, R2 and R3, contains only the tasks that would be ready at time t in the static optimal schedule $\mathcal{S}^{can}$. Further, the $rem_{i,j}$ field contains the remaining allotted time of each active instance $T_{i,j}$ at time t in $\mathcal{S}^{can}$.*

Observation 1 stems from the following: (a) α-queue is ordered according to EDF* order, (b) every arriving task pushes its remaining worst-case execution time (under nominal speed) into the α-queue only once, (c) the queue is updated only at the head, reflecting the fact that only the task with the highest EDF* priority would be running in $\mathcal{S}^{can}$, and (d) a task that would have finished in $\mathcal{S}^{can}$ is removed from the α-queue. This effectively yields *a dynamic image* of the ready queue in $\mathcal{S}^{can}$ at time t.

Note that the dynamic reduction of $rem_{i,j}$ in R3 above does not need to be performed at every clock cycle; instead, for efficiency, we perform the reduction whenever a task is preempted or completes, by taking into account the time elapsed since the last update. The above approach relies on two facts: first, the speed adjustment decision will be taken only at arrival/preemption and completion times, and it is necessary to have an accurate α-queue only at these points (if speeds are to be changed at other points, the update of $rem_{i,j}$ must reflect that). Second, between these points, each task is effectively executed non-preemptively.

We are now ready to present our Generic Dynamic Reclaiming Algorithm, GDRA, shown in Figure 2. Procedure Speed-Reduce(T_x, B, S), in Figure 3, will be used by GDRA to reduce the speed S of T_x, by allocating an extra B units of time to T_x under worst-case remaining load, subject to S_{min} constraint. GDRA is "generic" in the sense that the amount of additional time allocation Y in step 5.2 is not specified, it may assume any value between 0 and $\epsilon_x(t)$ without compromising the correctness.

The following theorem establishes that the schedules produced by GDRA are always *ahead* of $\mathcal{S}^{can}$.

Theorem 1 *At any time t during the execution of GDRA, $w_i^{\widehat{S}_i}(t) \leq rem_i(t)$, for any ready task T_i.*

The formal proof of this theorem can be found in [3]. Focusing exclusively on task completion times, the theorem implies that in the actual schedule no task instance completes later than its completion time in $\mathcal{S}^{can}$ (which is feasible), proving the correctness of GDRA:

Corollary 1 *GDRA yields a feasible schedule under EDF* priority for a workload no greater than the worst-case workload.*

Note that any specific algorithm should specify the *exact* amount of earliness parameter Y, to use for speed reduction. One natural choice in Rule 5.2 of Figure 2 is to use $Y = \epsilon_x(t)$, that is, to reduce the speed so as to profit from the full earliness. We call this variation simply *Dynamic Reclaiming Algorithm* (DRA).

4.1 Incorporating One Task Extension (OTE) Technique

As presented in [21], one can further slow down execution when there is only one task in the ready queue and its worst

Rules for GDRA

1. Compute $\bar{S}$ (as in Section 3) and assign $\widehat{S_{i,j}} = \bar{S}\ \forall\, i, j$.
2. Initialize the α-queue to the empty-list.
3. At every new arrival, insert into the α-queue information regarding the new task T_i with $rem_i(t) = w_i^{\widehat{S_i}}$ value in the correct EDF* order.
4. At every event (arrival/completion), consider the head of the α-queue and decrease its rem_i value by the amount of elapsed-time since the last event. If rem_i is smaller than the time elapsed since the last event, remove the head, update the time elapsed since the last event, and repeat the update with the next element. This is done until all "elapsed time" is used.
5. Whenever T_x is about to be dispatched at time t:
   - 5.0. Set $S_x = \widehat{S_x}$.
   - 5.1. Consult the α-queue and compute $\epsilon_x(t)$ (indicator of the earliness amount of T_x)
   - 5.2. Reduce the speed of task T_x by giving T_x an extra Y time units: S_x = Speed-Reduce(T_x, Y, S_x), where $0 \leq Y \leq \epsilon_x(t)$
6. At every event of preemption or completion of a task, say T_i, decrease the value of the remaining execution time: $w_i^{S_i} = w_i^{S_i} - \Delta_t$, where Δ_t is the time elapsed since the task T_i was last dispatched.

Figure 2. Generic Dynamic Reclaiming Algorithm

Procedure Speed-Reduce(T_x, B, S):

1. Set $S_x = \frac{w_x^S}{w_x^S + B} \cdot S$
2. If $S_x < S_{min}$, $S_x = S_{min}$
3. return S_x

Figure 3. Speed Reduction Procedure

case completion time (under the current speed) does not extend beyond the *next event* (next arrival/closest deadline of any task). Since this technique can be used in conjunction with any scheduling policy, we add a new rule 5.3 to further improve (G)DRA. Let NTA be the next arrival time of any task instance in the system after t, and recall that S_x is the speed from step 5.2 in (G)DRA and t is the time T_x is dispatched.

5.3. If T_x is the only ready task and $Z = NTA - t - w_x^{S_x}(t) > 0$, S_x = Speed-Reduce(T_x, Z, S_x).

In other words, reduce the speed of T_x so as to use the idle CPU up to time NTA. We call this improved technique DR-OTE.

Clearly, the following holds.

Proposition 5 *If all instances meet their deadlines under DRA, they will also meet their deadlines under the algorithm DR-OTE.*

4.2 Experimental Results

In order to experimentally evaluate the performance of DRA, we implemented a periodic scheduling simulator for EDF* policy. We implemented the following schemes: (a) **Static** uses constant speed $\bar{S}$, and switches to power-down mode (i.e., $S = S_{min}$) whenever there is no ready task; (b) **OTE**: Static optimal speed scheme in conjunction with One Task Extension (but without dynamic reclaiming), and (c) **DRA**, which is implemented in two variations: with or without the OTE technique (DR-OTE and DRA, respectively).

In our experiments, we investigated the average performance of the schemes over a large spectrum of worst-case utilization (U_{tot}) and variability in actual workload. In particular, we focused on the average energy consumption of 100 task sets, each containing 30 tasks. The periods of the tasks were chosen randomly in the interval [1000, 32000]. The minimum speed S_{min} is set to 0.1. The nominal speed $\bar{S}$ is set to U_{tot}, as the optimality of this choice was shown in Section 3. The variability in the actual workload is achieved by modifying the $\frac{WCET}{BCET}$ ratio (that is, the worst-case to best-case execution time ratio). We ran experiments where the actual execution time follows a normal probability distribution function[3]. The mean and the standard deviation are set to $\frac{WCET+BCET}{2}$ and $\frac{WCET-BCET}{6}$ respectively, for a given $\frac{WCET}{BCET}$, as suggested in [21]. These choices ensure that, on the average, 99.7% of the execution times fall in the interval $[BCET, WCET]$. For each task set we simulated the execution up to LCM 10 times, where LCM is the Least Common Multiple of $P_1, \ldots, P_n$, and measured the average energy consumption per experiment using a cubic power/speed function [10].

One remarkable result is the following: Although the OTE scheme provides substantial improvements over techniques that continuously use S_{max} during the execution without reclaiming as shown in [21], throughout the entire spectrum, DR-OTE only provides a marginal (less than 1%) improvement over pure DRA. This result indicates that almost the entire power savings are obtained by initially committing to $\bar{S}$ which fully utilizes the CPU (static scheme) *and* to the dynamic reclaiming algorithm itself. To improve the readability of the graphs, we show below only the results of DR-OTE, since the results for the latter are almost identical to pure DRA.

Effect of Utilization: Figure 4 shows the energy consumption of the techniques varying with the utilization of the task set (i.e.

[3]The results with a uniform probability distribution function are rather similar[3]. We also repeated the simulations with task sets having different number of tasks. The full results can be found in [3], in the lack of space, we only mention that the main trends remain similar to that of 30-task system. This is expected, since the main determinant of the workload is the total utilization and the variability in the actual workload.

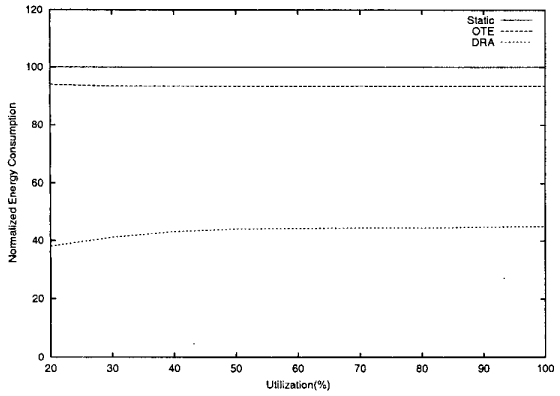

Figure 4. Normalized energy consumption (30 tasks). $\frac{WCET}{BCET} = 5$

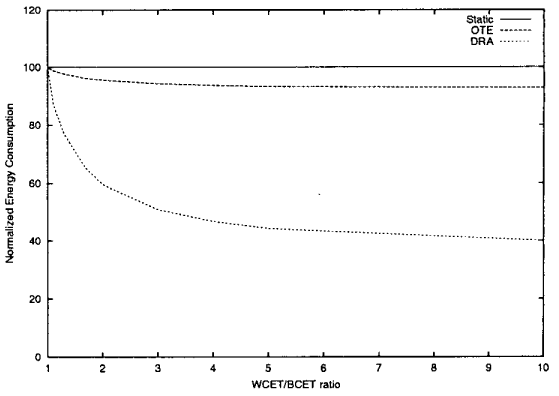

Figure 5. Effect of variability in actual workload (30 tasks); load = 60%

U_{tot}), when $\frac{WCET}{BCET}$ is equal to 5. The results are normalized with respect to **Static** which does not reclaim unused computation time. One can observe the following major patterns:

- The normalized energy consumption of all three schemes are rather insensitive to the variations in U_{tot}. This is due to the fact that, for a given scheme, the use of optimal nominal speed $\bar{S}$ results in having very similar *effective* utilization, for any value of U_{tot}. In other words, when the utilization decreases, the speed decreases making the CPU fully utilized.
- DRA has a definitive advantage over Static and OTE for all utilization values: the energy consumption of a system using DRA is only around 40% of a system which uses Static or OTE.
- OTE performs better than Static, but the improvement is usually less than 10%. This implies that the large power savings reported over continuously using S_{max} for some task sets in [21] are due largely to the shutting down of the processor when the processor is idle as the result of the actual workload. If and when one starts with the optimal static speed, the potential (additional) savings due to the OTE technique itself becomes rather limited.

Effect of $\frac{WCET}{BCET}$ ratio: The simulation results confirmed our prediction that the energy consumption would be highly dependent on the variability of actual workload. The (normalized) average energy consumption of the task sets, as a function of $\frac{WCET}{BCET}$ ratio (with $U_{tot} = 0.6$) is shown in Figure 5. In terms of shape and percentage difference, the curves for other utilization values are fairly similar. From these experiments we arrived at the following conclusions:

- When $\frac{WCET}{BCET} = 1$, there is no CPU time to reclaim dynamically, and thus the energy consumption is the same for all three techniques, as expected. However, once the actual workload starts decreasing (that is, increasing $\frac{WCET}{BCET}$), OTE and DRA are able to reclaim unused computation time and they are able to save additional energy.
- The DRA is capable of providing considerably higher power savings than OTE; and the difference increases rapidly with $\frac{WCET}{BCET}$ ratio. For instance, the savings of DRA even for $\frac{WCET}{BCET} = 1.1$ is better than the performance of OTE throughout the entire spectrum.
- Once we increase the $\frac{WCET}{BCET}$ beyond 4, power savings of DRA continue to increase, but the improvement is not as impressive as the case where that ratio is ≤ 4. This is because the expected workload of the system converges rapidly to 50% of the worst-case workload with increasing $\frac{WCET}{BCET}$ ratio (remember that the mean of our probability distribution is $\frac{WCET+BCET}{2}$).

5 Aggressive Speed Reduction

The DRA and DR-OTE algorithms provide sound dynamic speed reclaiming mechanisms, however they guarantee feasibility by always being 'ahead' of the static worst-case optimal schedule $\mathcal{S}^{can}$ (i.e., tasks never actually start or finish after the scheduled time in $\mathcal{S}^{can}$). $\mathcal{S}^{can}$ is feasible at any time, yet it is optimal only under the assumption that all future instances will present their worst-case workload. Whenever, under constant speed, the actual execution times of a task's instances exhibit large variation, starting a task with this assumption can be too conservative. Instead, whenever the current system state suggests, we may **assume speculatively that the current and future instances will most probably present a computational demand which is lower than the worst-case**. Hence, we can adopt an "aggressive" approach based on reducing the speed

of the running task under certain conditions to a level which is even lower than the one suggested by DR-OTE. But this speculative move might shift the task's worst-case completion time to a point later than the one in $\mathcal{S}^{can}$ under an actual high workload. And if this pessimistic scenario turns out to be true, we should be ready to **increase the CPU speed beyond $\bar{S}$ later to guarantee feasibility of future tasks**. This would hamper significant power savings since the convexity of power/speed curve suggests a uniform speed to achieve a given average speed value over any interval of time. On the other hand, in case that the actual workload turns out to be lower than the worst-case, the actual schedule will *still* be ahead of $\mathcal{S}^{can}$, even with the low speed, thereby achieving even higher power savings.

A powerful system design principle is to make the common case more efficient. This translates (in settings where the worst-case workload occurs only rarely) into having a power-efficient schedule for average or close to average cases, which can be achieved by reducing further the CPU speed. After having presented the rationale of aggressive speed management techniques, we should address and provide solutions for two important issues.

The first one is **feasibility**: when we reduce the speed of T_x aggressively, we should be ready to guarantee the timing constraints of T_x and that of any other task, since the schedule may no longer be 'ahead' of $\mathcal{S}^{can}$. The second issue is the **determination of the aggressiveness level**: even though it may be possible to show the existence of a feasible schedule (under a very aggressive speed reduction for T_x), if such a move is not justified by the expected workload of the system, it may be reasonable to adopt a more conservative speed reduction, to decrease the probability of speed increases which cause high energy consumption. A natural solution is to use a pre-defined *speed reduction bound* (Sb) below which we never attempt to decrease the CPU speed during an aggressive speed adjustment. Observing that the "average workload" is an appropriate estimator for the actual computational demand, we choose to parameterize the aggressiveness level with respect to the *optimal speed under an average workload* (S_{optavg}). More specifically, S_{optavg} is the optimal speed for the workload where each instance requires exactly its average computational demand (determined by a probability distribution function). Generally, we may set Sb to $k \cdot S_{optavg}$, where k is a constant such that $S_{min} \leq Sb \leq S_{max}$ (i.e., $\frac{S_{min}}{S_{optavg}} \leq k \leq \frac{S_{max}}{S_{optavg}}$). Observe that changing k in this range provides a complete spectrum of "aggressive techniques". At one end of the spectrum, $k = \frac{S_{min}}{S_{optavg}}$ (which is usually much smaller than 1.0) corresponds to the "extreme aggressiveness" where we attempt to obtain the lowest speed level for the running task; this is only subject to feasibility which might be achieved later only by executing the following tasks with very high speeds (i.e., by this choice, we are supposing that the current workload will be well below the worst-case workload). At the other end of the spectrum, setting $k = \frac{S_{max}}{S_{optavg}}$ reflects the DR-OTE algorithm itself. Another main point in the spectrum is the scheme which limits the aggressiveness speed bound by exactly S_{optavg}, that is, $k = 1$; this reflects the view that slowing down the CPU below S_{optavg} will hurt the aggregate power savings in the long run.

5.1 Feasibility for Aggressive Schemes

As mentioned above, when we attempt to aggressively reduce the CPU speed, we risk exceeding worst-case completion times of $\mathcal{S}^{can}$ in the current schedule, both for the running, ready and yet-to-arrive tasks. In general, to check the consequences of such an aggressive decision is a non-trivial problem (linked with response-time analysis complications of EDF), especially if it is to be addressed in a dynamic fashion, at run-time. In this study, we adopt a simple approach that restricts the aggressive power management to occur only when we can limit their effects upto the next event (arrival/deadline of any task). As the results in Section 5.4 below indicate, the aggressive schemes have the potential of providing additional power savings, even with this conservative feasibility test with limited horizon.

Whenever we can predict that the completion time of the currently ready task T_x will not extend beyond the next event (arrival/deadline), we can speculatively reduce the speed of T_x while guaranteeing that it will still complete before the next event (which is, by definition, earlier than or equal to the deadline of T_x). However, care must still be taken in order to guarantee the timely completions of other ready tasks which are waiting on the ready queue at a lower priority level than T_x, since the execution/completion of these tasks will be delayed until T_x completes.

A possible way to guarantee the feasibility in this case is *to increase* the speed of another suitable and ready task T_y which will run after T_x. This is *effectively equivalent* to increasing the time allocation of T_x, while *decreasing* the time allocation of T_y by the same amount. Clearly, from this point on, the system cannot blindly decrease the speed of T_y to its original $\widehat{S_y}$ (i.e., we should also change $\widehat{S_y}$ for that instance).

One can even generalize this with the following: if $T_1, T_2, ..., T_r$ are ready tasks that are guaranteed to run consecutively and *all* to complete before the next task arrival time (NTA) even under worst-case workload, we can arbitrarily swap CPU time allocations among them (in particular to reduce the speed of T_1 while increasing the speeds of one or more of $T_2, ..., T_r$). In fact, if it exists, even the highest priority ready task that is *not* guaranteed to complete before NTA (namely T_{r+1}) may provide a portion of its time allocation under certain conditions. However, we must still guarantee that $T_1, T_2, ..., T_r$ will complete before NTA and T_{r+1} will complete no later than before the time allocation swapping, under the worst-case scenario. Further, in all these computations, we should take into account the slack-time of already completed tasks in the α-queue (with EDF* priority lower than T_1) that may contribute to the worst-case CPU allocation of $T_2, \ldots, T_r, T_{r+1}$ in

the future through dynamic reclaiming. Finally, all these speed adjustments should adhere to S_{min}, S_{max} and Sb bounds.

To incorporate the aggressive speed reduction technique, we add a new rule 5.4, to the previous algorithm, thereby obtaining the new algorithm **Aggressive-DR**:

5.4. If $Z = NTA - t - w_x^{S_x}(t) > 0$ and there are other ready tasks in addition to T_x, call `Aggressive-Speed-Adjustment`.

Procedure *Speed-Increase* (Figure 7) increases the speed S of T_x so as to remove at most H units of time allocation under worst-case remaining workload of T_x with respect to the speed S, subject to S_{max}. In procedure Aggressive-Speed-Adjustment, whenever T_1 transfers slack-time from T_j, we perform the speed increase for T_j, increasing $\widehat{S_j}$, the *nominal speed of* T_j. Whenever T_j is about to be dispatched, its current speed will be set to $\widehat{S_j}$ by rule 5.0; rules 5.1 and 5.2 should consider this new (increased) speed when trying to reduce speed due to a (possible) earliness detection. Finally, T_j should assume the new nominal speed $\widehat{S_j}$ when it returns from preemption, since this is the lowest speed known to guarantee a feasible schedule in the case where every task presents its worst-case load to the processor after aggressive speed adjustments. However, we underline that the nominal speed $\widehat{S_j}$ of *future* instances of T_j are unchanged and equal to $\bar{S}$. A formal proof regarding the correctness of the `Aggressive-Speed-Adjustment` routine is provided in [3].

5.2 Evaluation of the Aggressive Scheme

We conducted experiments to assess the performance of the aggressive scheme (abbreviated by AGR), in the same settings as Section 4.2. The speed bound Sb for the speculative speed adjustment is equal to S_{optavg}, that is, the aggressiveness factor k is set to 1. In Figure 8, the relative energy consumption of AGR with respect to DRA is shown, for 30-task sets and normal distribution, as a function of the utilization. The results show a consistent advantage of AGR over DRA throughout the spectrum (around 15%). The improvement decreases as the utilization approaches 100%, where all tasks assume a nominal (default) speed $\bar{S} = 1.0$ and aggressive speed reduction at the expense of increasing the speed of others is not always possible.

The effect of variability in actual workload is shown in Figure 9. Again, AGR provided better performance than DRA with various $\frac{WCET}{BCET}$ ratios. Increasing this ratio improves the relative performance of AGR, since the speculative moves are justified more frequently.

5.3 More on Speed Bound Restrictions

Another possible approach for using the aggressive scheme is to adhere to the 'parameterized speed bound' **even when reducing the speed in Step 5.2 through dynamic reclaiming**. This approach assumes that reducing the speed below $k \cdot S_{optavg}$ will hurt the total performance in the long run, and prevents doing so even when the earliness factor would justify doing so. To distinguish two variations of the aggressive scheme, we will denote the original scheme and the new variation by **Aggressive-DR-1** and **Aggressive-DR-2**, respectively (or, AGR1 and AGR2, for short).

The correctness of the new scheme follows from the cor-

Procedure Aggressive-Speed-Adjustment

Notation: The algorithm is invoked at time t. The ready task with the highest EDF* priority is denoted by T_1. The other tasks that are ready, or that are completed but have their unused computation time in the α-queue with EDF* priority lower than that of T_1, are denoted by $T_2, ..., T_m$, $2 \leq m \leq n$, in decreasing order of priorities. Throughout the algorithm, at the cost of a slight abuse of notation, we will also use the expression $w_i^{S_i}(t)$ to refer to $Rem_i(t)$ value of any completed task T_i in the α-queue at time t. The current speed assignments are denoted by $S_1, ..., S_m$, and the next task arrival after t will occur at time NTA.

Algorithm:

- If $S_1 \leq max\{S_{min}, k \cdot S_{optavg}\}$ return; (that is, we should not decrease the speed any further)
- Determine the maximum amount of additional CPU time, Q, that can be assigned to T_1, subject to S_{min} and the aggressiveness level constraints:

$$Q = [\frac{S_1}{max(S_{min}, k \cdot S_{optavg})} - 1] w_1^{S_1}(t).$$

- Adjust Q in order not to extend beyond NTA: if $NTA - t - w_1^{S_1}(t) < Q$ then $Q = NTA - t - w_1^{S_1}(t)$.
- $Q_a = 0$ (already transferred slack amount).
- If $w_2^{S_2}(t) \geq Q$ then $\{r = 1; Z = 0\}$
  else find the largest r $(2 \leq r \leq m)$
  such that $Z = \sum_{i=2}^{r} w_i^{S_i}(t) \leq Q$.
- Increase the speed of $T_2, ..., T_{min(m,r+1)}$ while reducing the speed of T_1:
  - $j = 2$
  - while $(j \leq min(m, r+1)\ and\ Q_a < Q)$
    - if $(j < r+1)$ then $Extra\_time = Q - Q_a$
      else $Extra\_time = Q - Z$
    - if T_j is ready then:
      - S_j = Speed-Increase$(T_j, Extra_time, \widehat{S_j})$
      - $B = (\frac{S_j}{\widehat{S_j}} - 1) \cdot w_j^{\widehat{S_j}}$
      - $\widehat{S_j} = S_j$ (that is, commit to the new S_j as the default speed of that instance)
    - if T_j is completed but is in the α-queue then $B = min(Extra\_time, Rem_j)$
    - $j = j + 1$
    - $Q_a = Q_a + B$
    - S_1 = Speed-Reduce(T_1, B, S_1)

Figure 6. Aggressive Speed Adjustment Procedure

Procedure Speed-Increase(T_x, H, S)
1. $S_x = \frac{w_x^S + H}{w_x^S}$
2. If $S_x > S_{max}$ then $S_x = S_{max}$;
3. return S_x

Figure 7. Speed Increase Procedure

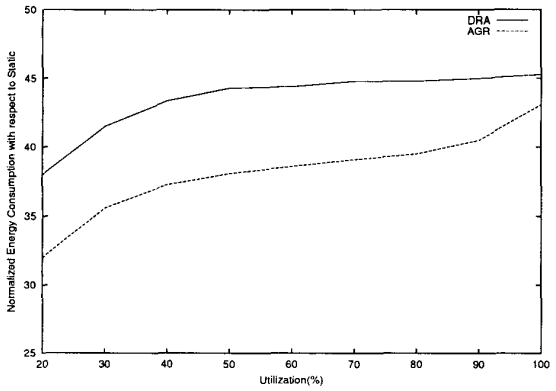

Figure 8. Normalized energy consumption ($\frac{WCET}{BCET} = 5$)

rectness of AGR1, since AGR2 never slows down the processor more than AGR1.

5.4 Evaluation of AGR1 and AGR2

In this section, we present results of simulations performed to compare algorithms AGR-1 and AGR-2. The simulation settings are identical to those of Section 4.2. When the utilization or the $\frac{WCET}{BCET}$ ratio is changed, the performance of AGR1 and AGR2 are hardly distinguishable [3].

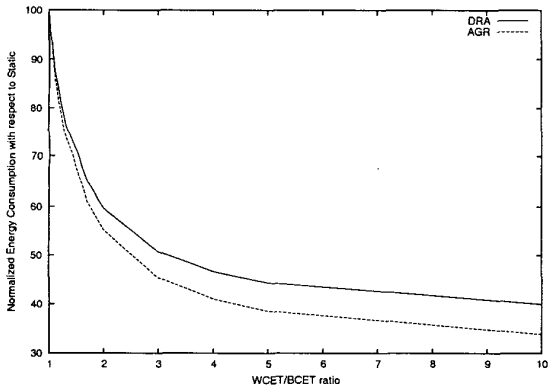

Figure 9. Effect of variability in actual workload (load = 60%)

However, unlike the utilization and $\frac{WCET}{BCET}$ ratio, changing **the aggressiveness level** deeply affects the results, as shown in Figure 10. The curves shown are for 60% utilization and $\frac{WCET}{BCET} = 5$; other parameter settings have very similar behavior. The performances of DRA and Static are insensitive to the parameter k. The maximum power savings is are obtained with algorithm AGR2 typically when $k = 0.9$. This represents a further 5% improvement over $k = 1$, yielding a net advantage of 20% over DRA. AGR1 reaches its minimum energy consumption usually with $k = 1$. Further, the curve suggests that unbounded or extreme aggressiveness (small values of k) hinders the power savings: for instance, both schemes consume 60% more energy than DRA for $k \leq 0.2$.

Yet, as we increase the value of k and move towards more 'balanced' aggressiveness levels, the aggressive schemes become preferable to DRA: AGR1 and AGR2 outperforms DRA, for $k \geq 0.75$ and $k \geq 0.7$, respectively. After the power savings reach their maximum at $k = 0.9$ (for AGR2) and $k = 1.0$ (for AGR1), the performance starts to degrade. Remarkably, for $k \geq 1.1$, AGR2 consumes considerably higher energy than AGR1: this is due to the fact that when the algorithm is run with large values of k, the algorithm is reluctant to reclaim or transfer CPU-time, even when the speed is higher than S_{optavg}. AGR1 does not suffer from this effect, since it automatically uses the earliness information to perform an initial speed reduction and considers the speed bound Sb only when aggressively reducing speed. Hence, even for large values of k, AGR1 remains better than DRA, and is guaranteed to converge to it for $k = \frac{\bar{S}}{S_{optavg}} = \frac{2}{1+\frac{BCET}{WCET}}$, which is 1.66 for this example. On the other hand, AGR2 converges to OTE (not shown in Figure 10) for the same value; this is because the actual speed starts with $\bar{S}$, and the aggressive or dynamic reclaiming is never possible since $Sb = \bar{S}$. In this case, CPU speed is reduced only through OTE.

In summary, keeping k in the range $[0.9, 1]$ and committing to an aggressiveness level which aims to achieve very close to S_{optavg} produces best results, yielding *additional* (i.e., beyond DRA or DR-OTE) energy savings which may be as high as 20%.

6 Conclusions

In this paper we presented techniques for power-aware real-time computing through variable voltage scheduling. Our solution comprised three parts (a) a static solution to compute the optimal speed based on the worst-case workload, (b) an on-line speed adjustment mechanism that reclaims unused time based on the actual workload, and (c) a speculative speed adjustment mechanism based on the expected workload. To our knowledge, this is the first time that aggressive and provenly safe techniques are used to anticipate and provision for the early completions in periodic real-time scheduling.

Our simulation results show that the reclaiming algorithm saves a striking 50% of the energy over the static algorithm,

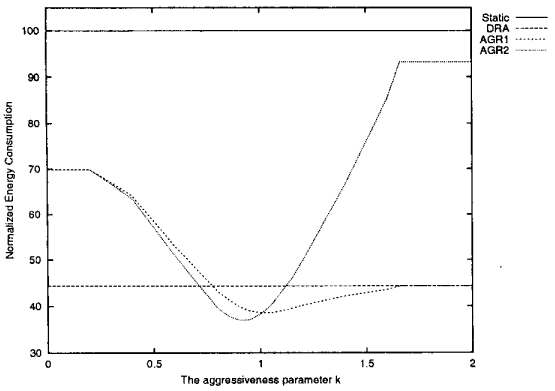

Figure 10. Effect of bounding factor k in Aggressive Schemes (30 tasks); utilization = 0.6 and $\frac{WCET}{BCET} = 5$

which takes into account the load in the system. This quite significant result shows that the lifetime of mobile or other battery operated devices can be extended on average to twice the levels of static solutions. Considering also the data presented in our previous work [16], we conclude that batteries can be extended to last up to one order of magnitude longer over no power management schemes.

Further, our preliminary aggressive techniques allow for an additional 20% savings over the reclaiming algorithm. We conclude that, being too aggressive or not aggressive enough causes the algorithms to perform rather poorly. We are currently studying less conservative approaches (that is, not stopping the aggressiveness by the "next event") that we believe will lead to further energy savings.

References

[1] H. Aydin, R. Melhem, D. Mossé and P.M. Alvarez. Optimal Reward-Based Scheduling for Periodic Real-Time Tasks. *IEEE Transactions on Computers* 50(2): 111-130, February 2001.

[2] H. Aydin, R. Melhem, D. Mossé and P.M. Alvarez. Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics. In the *Proceedings of the 13th EuroMicro Conference on Real-Time Systems (ECRTS'01)*, Delft, Netherlands, June 2001.

[3] H. Aydin. *Enhancing Performance and Fault Tolerance in Reward-Based Scheduling.* Ph.D. Dissertation, University of Pittsburgh, August 2001.

[4] J. K. Dey, J. Kurose and D. Towsley. On-Line Scheduling Policies for a class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks. *IEEE Transactions on Computers* 45(7):802-813, July 1996.

[5] R. Ernst and W. Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. In *Computer-Aided Design (ICCAD)'97.* pp. 598-604.

[6] V. Gutnik and A. Chandrakasan. An Efficient Controller for Variable Supply Voltage Low Power Processing. *Symposium on VLSI Circuits*, pp.158-159, 1996.

[7] P. J. M. Havinga and G. J. M. Smith. Design Techniques for Low-power Systems. *Journal of Systems Architecture.* Vol. 46:1, 2000

[8] I. Hong, D. Kirovski, G. Qu, M. Potkonjak and M. Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th Design Automation Conference, DAC'98*

[9] I. Hong, M. Potkonjak and M. B. Srivastava. On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *Computer-Aided Design (ICCAD)'98.* pp. 653-656.

[10] I. Hong, G. Qu, M. Potkonjak and M. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, December 1998.

[11] C. M. Krishna and Y. H. Lee. Voltage Clock Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, Washington D.C., May 2000.

[12] C.L. Liu and J.W.Layland. Scheduling Algorithms for Multiprogramming in Hard Real-time Environment. *Journal of ACM* 20(1): pp.46-61, 1973.

[13] J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer,* 24(5): 58-68, May 1991.

[14] J. R. Lorch and A. J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, Cambridge, MA, June 2001.

[15] D. Luenberger, *Linear and Nonlinear Programming,* Addison-Wesley, Reading Massachusetts, 1984.

[16] D. Mossé, H. Aydın, B. Childers and R. Melhem. Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications. *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, Philadelphia, PA, October 2000.

[17] W. Namgoang, M. Yu and T. Meg. A High Efficiency Variable-Voltage CMOS Dynamic DC-DC Switching regulator. *IEEE International Solid-State Circuits Conference*, pp.380-391

[18] M. Pedram. Power Minimization in IC Design: Principles and Applications. *ACM Transactions on Design Automation of Electronics Systems.* 1:1 - pp. 3-56, January 1996.

[19] J. Pouwelse, K. Langendoen and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. 7^{th} *International Conference on Mobile Computing and Networking (MOBICOM)*, Rome, Italy, July 2001.

[20] R. Rajkumar, C. Lee, J. P. Lehoczky and D. P. Siewiorek. A Resource Allocation Model for QoS Management. In *Proceedings of 18th IEEE Real-Time Systems Symposium*, December 1997.

[21] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proceedings of the 36th Design Automation Conference, DAC'99.*

[22] http://www.transmeta.com

[23] F. Yao, A. Demers and S. Shenker. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995.