

Power-Aware Scheduling for Periodic Real-Time Tasks*

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

Abstract

In this paper, we address power-aware scheduling of periodic tasks to reduce CPU energy consumption in hard real-time systems through dynamic voltage scaling. Our inter-task voltage scheduling solution includes three components: (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 our reclaiming algorithm alone outperforms other recently proposed inter-task voltage scheduling schemes. Our speculative techniques are shown to provide additional gains, approaching the theoretical lower-bound by a margin of 10%.

Keywords: Real-Time Systems, Power-Aware Computing, Low-Power Systems, Dynamic Voltage Scaling, Periodic Task Scheduling

*A preliminary version of this paper has appeared in the Proceedings of the 22nd Real-Time Systems Symposium (RTSS 2001). 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.

1 Introduction

In the last decade, the research community has addressed the low-power system design problems with a multi-dimensional effort. Reducing energy consumption of a computer system has necessarily multiple aspects, involving separate components such as CPU, memory system and I/O sub-system. Hardware and software manufacturers have agreed to introduce standards such as the ACPI (Advanced Configuration and Power Interface) [10] for power management of laptop computers that allows several modes of operation, such as predictive system shutdown. An obvious target for energy reduction is the processor: an early study found that 18-30 % of the total energy consumption is due to the CPU alone [17]. More recent reports show that this fraction can exceed 50 % for CPU-intensive workloads [24, 31]. *Dynamic voltage scaling* (DVS) framework, which involves dynamically adjusting the voltage and frequency to reduce CPU power consumption has recently become a major research area.

In fact, the power consumption of an on-chip system is a strictly increasing *convex* function of the supply voltage V_{dd} , but its exact form depends on the technology. For example, the dominant component of energy consumption in widely popular CMOS technology is the *dynamic power dissipation* P_d , which is given by $P_d = C_{eff} \cdot V_{dd}^2 \cdot f$, where C_{eff} is the effective switched capacitance and f is the frequency of the clock. On the other hand, the gate delay D is inversely related to the supply voltage V_{dd} as given by the formula $D = k \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2}$, where k is a constant, and V_t is the threshold voltage. From these equations, it can be seen that one can obtain striking power savings if the supply voltage and the clock frequency are reduced simultaneously. Thus, the *dynamic voltage scaling* technique (known also as *variable voltage scheduling*) is based on obtaining energy savings by *simultaneously* reducing the supply voltage and the clock frequency, at the expense of increased latency. The exact form of the power/speed relation can be obtained by resorting to *normalization*: recent research studies [9, 11, 15, 14] report formulas where the CPU power consumption per cycle when expressed in terms of CPU speed is a polynomial of the third degree. In general, this relation can be captured by strictly increasing *convex* functions. 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, 21]. For example, the Crusoe processor is able to dynamically adjust clock frequency in 33 MHz steps [28].

For systems with timing constraints and scarce energy resources, we solve the Real-Time Dynamic Voltage Scaling (RT-DVS) problem: **adjust the supply voltage and clock frequency to minimize CPU energy consumption while still meeting the deadlines.** First, task

speed assignments that minimize the energy consumption for the worst-case workload should be evaluated at the **static** level. However, limiting RT-DVS with this static solution would be very conservative: real-time applications usually exhibit large variations in *actual* workload experienced by the system; for example [4] 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. Thus, at the second, **reclaiming** level, dynamically monitoring task executions and further reducing CPU speed by re-allocating unused CPU time can provide additional savings. Finally, we can consider the **speculation** level where early task completions are anticipated and the CPU speed is aggressively reduced. However, it is imperative that all these components be designed not to cause any deadlines to be missed even under a worst-case workload that can happen after any speed adjustment point.

1.1 Related Work

The work by Weiser et al. [29] was among the first to propose (and evaluate the performance of) various DVS algorithms, though the focus of that study was non-real-time tasks. Yao et al. [30] provided a static, optimal and polynomial-time scheduling algorithm for real-time tasks with release times and deadlines, assuming aperiodic tasks and worst-case execution times. Pering and Brodersen [22] also addressed the static dimension of RT-DVS, but only for aperiodic tasks. Static solutions for extended/hybrid task models were addressed in various papers: Heuristics for on-line scheduling of aperiodic tasks while not hurting the feasibility of periodic requests are proposed in [8]. The same paper also suggested that the CPU speed be set to the utilization of the periodic tasks' utilization value in the absence of aperiodic tasks, but without mentioning or proving the optimality of this choice. Non-preemptive power aware scheduling is investigated in [7]. Concentrating on periodic task sets with identical periods, the effects of having an upper bound on the voltage change rate are examined in [9], along with a heuristic to solve the problem.

The possible variations in actual workload of real-time systems, and hence *reclaiming* and *speculation* dimensions attracted the attention of power management research community first in late 90's. These dynamic algorithms, according to an established classification [5, 26, 12] fall either into *inter-task* or *intra-task* DVS algorithm categories. In the former case, speed assignments are determined at task-level: though dynamic adjustments are performed, these occur only at task dispatch or completion times. That is, once a task is assigned CPU, the CPU speed is not changed until it is preempted or completed. On the other hand, intra-task algorithms adjust speed within the boundaries of a given task; typically, the speed is gradually

increased to assure the timely completion. Intra-task algorithms usually require some degree of compiler support to insert *power management points* to application code, in order to call explicitly Operating System services for speed reduction. On the other hand, inter-task algorithms do not involve such changes at the application code, and thus they are more practical.

In the *intra-task* DVS research, Lorch and Smith addressed the scheduling of aperiodic tasks with *soft* deadlines in [18]. Shin et al. [26] proposed an Intra-task DVS framework for aperiodic real-time tasks with (possibly) precedence constraints. Gruian [5] provided an intra-task DVS algorithm for periodic tasks scheduled according to Rate-Monotonic policy.

Among the *inter-task* DVS algorithms, an early technique based on slowing down the processor whenever there is a single task eligible for execution is given in [27]. Dynamic energy reclaiming issues (without speculation) in power-aware scheduling was addressed [13] for cyclic and periodic task models in the context of systems with two (discrete) voltage levels. To the best of our knowledge, the concept of “speculative speed reduction” was first introduced by the authors in [20]; however, only tasks sharing a common deadline were considered. A recent study particularly relevant for the settings of this paper is due to Pillai and Shin [23], where the authors proposed a reclaiming algorithm (Cycle-Conserving EDF) and a speculation-based algorithm (Look-Ahead EDF). These inter-task DVS algorithms are based on updating and predicting the instantaneous utilization of the periodic task set. Finally, a recent performance evaluation [12] compares several inter-task and intra-task DVS algorithms in their own categories, including the preliminary versions of some algorithms discussed in this paper.

1.2 Paper Organization and Contributions

In this paper, we address three dimensions of *inter-task* dynamic voltage scaling algorithms for periodic real-time task systems, assuming that the CPU speed can be varied over a continuous spectrum between a lower bound and an upper bound. We develop novel algorithms and evaluate their performance against other inter-task DVS algorithms proposed in the literature, as well as against the provably optimal, clairvoyant algorithm whose performance provides a lower-bound for any (inter- or intra-task) DVS algorithm. Thus, we present:

1. A static (off-line) solution to compute the optimal speed at the task level, assuming worst-case workload for each arrival¹ (Section 3).

¹ Due to the nature of DVS, 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).

2. A generic dynamic reclaiming algorithm for tasks that complete without consuming their worst-case workload (Section 4). Our reclaiming algorithm differs from other inter-task reclaiming algorithms (e.g. CC-EDF in [23]) in that it attempts to allocate the maximum (possibly, the entire) amount of unused CPU time (slack) to the first task at the appropriate priority level, *in a greedy fashion*. Further, when preempted, tasks implicitly return the slack they inherited to the favor of the newly dispatched task. We formally prove that the tasks will still meet their deadlines if the speed is reduced according to the rules we provide upon early completions. We achieve these objectives by keeping track and comparing against the remaining execution times of tasks in static optimal schedule, in which each task instance presents its worst-case workload.
3. An on-line, adaptive and speculative speed adjustment mechanism to anticipate and compensate probable early completions of future executions (Section 5). We explore two intertwined questions raised by the speculative component namely, (a) the *level* of aggressiveness that *justifies* speculative speed reductions under a given probability distribution of actual workload; and (b) the issue of guaranteeing the timing constraints even in aggressive modes. Our aggressive algorithms differ from other inter-task speculation-based algorithms (e.g. LA-EDF in [23]) by reducing exclusively the speed of the *dispatched* task, at the expense of borrowing CPU time from other ready tasks. Further, the algorithms are still able to reclaim the available slack (if any) and we show that the feasibility is preserved even after speculative speed reductions. The aggressiveness level of these algorithms can be adjusted by the designer. In fact, our results indicate that a balanced aggressiveness level that aims to achieve the speed that corresponds to the *expected* workload gives the best results.

Through extensive simulations, we validate our reclaiming and aggressive algorithms against other state-of-the-art inter-task DVS algorithms, as well as the lower-bound that can be achieved by any (intra- or inter-) DVS algorithm (Section 6). We experimentally show that the dynamic reclaiming algorithm alone provides higher energy savings than other existing techniques. Moreover, our aggressive algorithms yield additional gains, approaching the theoretical lower bound by a margin of 10 %.

Finally, in Section 7, we consider the effects of additional considerations such as the applicability to platforms with discrete speed levels and the consequences of using other (quadratic) power/speed functions. We underline that our objective is to minimize CPU energy consumption; low-power techniques for memory, disk and I/O subsystems, albeit important, are beyond the scope of this paper.

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 continuously between a minimum speed S_{min} (that corresponds to the minimum supply voltage necessary to keep the system functional) and a maximum speed S_{max} . We also assume 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 [24] that the voltage/speed change can be performed in less than 140 μ s 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 [9]. If the task T_i occupies the processor during the time interval $[t_1, t_2]$, then the *energy* consumed during this interval is $E(t_1, t_2) = \int_{t_1}^{t_2} g(S(t))dt$. We consider only CPU power/energy consumption in this paper.

In our detailed analysis of periodic power-aware scheduling, we will consider a set $\mathcal{T} = \{T_1, \dots, T_n\}$ 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 [16] 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

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 show that the speed value which minimizes the total energy consumption corresponds to the utilization of the system subject to the minimum CPU speed constraint. We underline that in [2], it is shown that solving an instance of RT-DVS problem is equivalent to solving an instance of Reward-Based Scheduling (RBS) problem with concave reward functions [1, 3]. Further, the equivalence of RBS and RT-DVS 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 under timing constraints [2]. This means that solutions devised for the RBS problem can be used for the RT-DVS problem. However, one can also compute the static optimal speed for periodic task sets by using the first principles as outlined below.

Proposition 1 *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 $\sum_{i=1}^n \frac{C_i}{\bar{S} \cdot P_i} = \frac{U_{tot}}{\bar{S}} = 1$). Note that stretching out each task in equal proportions guarantees the optimality, thanks to the convexity of power consumption function $g(S)$ (see [2] for a formal proof). 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} . \square

4 Dynamic Reclaiming Algorithm

Though the static scheme can be shown to be optimal under a worst-case workload, it is known that in many cases the instances of real-time tasks complete earlier than under the worst-case scenario [4]. A trivial remedy would be to shutdown the processor when there are no ready tasks. However, this technique is clearly sub-optimal because of the convexity of the power function: it is always more energy-efficient to transfer unused CPU time to *other* tasks by reducing their speeds whenever possible.

The dynamic reclaiming algorithm we present in this section 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 under \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 *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 1). 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 early at $t = 10$, leaving

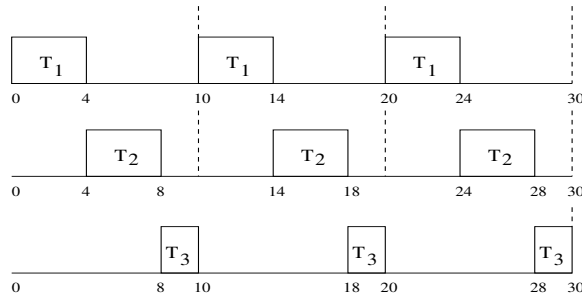


Figure 1: The static optimal schedule, \mathcal{S}^{can}

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. Moreover, 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. Let us denote by $rem_{i,j}(t)$ the remaining execution time of $T_{i,j}$ at time t in \mathcal{S}^{can} , under static optimal speed \bar{S} . 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). Specifically, at any time t , for each task $T_{i,j}$ in the α -queue the information about its identity, arrival time, deadline and $rem_{i,j}(t)$ value is available.

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 at most one instance of a given periodic task can be active in any schedule. Therefore, we will omit the instance number while referring to the α -queue elements, whenever possible.

Our approach assumes that tasks are scheduled according to EDF* policy. EDF* is the same as EDF (Earliest Deadline First [16]), 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 2 *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

Key Notation for Dynamic Reclaiming Algorithm

- \mathcal{S}^{can} : The schedule in which each task T_i presents its worst-case workload C_i at every instance and runs with nominal speed set to optimal static speed \bar{S}
- $rem_i(t)$: the remaining execution time of task T_i at time t in \mathcal{S}^{can}
- $w_i^S(t)$: the remaining worst-case execution time of task T_i under the speed S at time t in actual schedule
- $\epsilon_i(t)$: The earliness of task T_i at time t in actual schedule

Figure 2: Notation

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} .

Note that $\epsilon_x(t)$ is only a conservative estimation of the *actual* earliness of T_x . Any task with higher priority that becomes ready, preempts T_x and leaves the system early, before T_x ends, may further contribute to the delay of T_x and also increase the actual earliness of T_x ; yet in the absence of clairvoyant capabilities, not much can be done to deterministically take this into consideration. In addition, when T_x returns from preemption after such a future high priority interruption, it will anyhow be able to better evaluate its earliness, and further slow-down, thanks to the above formula. Therefore, we adopt this relatively simple and fast way of inferring the earliness amount *from past events*. The key notation that will be heavily used in the remaining of the paper is presented in Figure 2.

Implementing the α -queue: The α -queue can be 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*

Time	Event	The α -queue = $\{(Task_id, r_{i,j}, d_{i,j}, rem_{i,j})\}$
0	$T_{1,1}, T_{2,1}, T_{3,1}$ arrive	$\{(T_{1,1}, 0, 10, 4), (T_{2,1}, 0, 10, 4), (T_{3,1}, 0, 30, 6)\}$
4	$T_{1,1}$ completes	$\{(T_{2,1}, 0, 10, 4), (T_{3,1}, 0, 30, 6)\}$
8	$T_{2,1}$ completes	$\{(T_{3,1}, 0, 30, 6)\}$
10	$T_{1,2}, T_{2,2}$ arrive	$\{(T_{1,2}, 10, 20, 4), (T_{2,2}, 10, 20, 4), (T_{3,1}, 0, 30, 4)\}$
14	$T_{1,2}$ completes	$\{(T_{2,2}, 10, 20, 4), (T_{3,1}, 0, 30, 4)\}$
18	$T_{2,2}$ completes	$\{(T_{3,1}, 0, 30, 4)\}$
20	$T_{1,3}, T_{2,3}$ arrive	$\{(T_{3,1}, 0, 30, 2), (T_{1,3}, 20, 30, 4), (T_{2,3}, 20, 30, 4)\}$
22	$T_{3,1}$ completes	$\{(T_{1,3}, 20, 30, 4), (T_{2,3}, 20, 30, 4)\}$
26	$T_{1,3}$ completes	$\{(T_{2,3}, 20, 30, 4)\}$
30	$T_{2,3}$ completes	\emptyset

Table 1: Snapshots of the α -queue for the example task set

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) to 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 .

As an example, consider the 3-task system $\{T_1, T_2, T_3\}$ with parameters: $C_1 = 4, P_1 = 10, C_2 = 4, P_2 = 10, C_3 = 6, P_3 = 30$. $\bar{S} = U_{tot} = 1.00$, thus in \mathcal{S}^{can} , each task is to execute with maximum CPU speed. During the actual execution, the task set may present rather different workloads, but the α -queue will always be updated according to the rules given above, reflecting at any time the ready queue in \mathcal{S}^{can} when scheduled in conjunction with EDF* policy. Table 1 shows the snapshots at time points when a task is inserted to or removed from α -queue. Recall that, in principle, the rem_i field of the task at the head of the α -queue is decremented according to the rule R3.

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 *in actual schedule*, 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 3. Procedure Speed-Reduce(T_x, B, S), in Figure 4, will be used by GDRA to reduce the speed S of T_x , by allocating B extra units of CPU time to T_x under worst-case remaining load, subject to S_{min} constraint². GDRA is “generic” in the sense that the amount of additional time allocation Y in step 5.2 is not specified and it may assume any value between 0 and $\epsilon_x(t)$ without compromising the correctness. Moreover, the following theorem, whose proof is presented in Appendix, 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, $w_i^{\hat{S}_i}(t) \leq rem_i(t)$, for any ready task T_i .*

Focusing exclusively on task completion times (in \mathcal{S}^{can}), the result above 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 3 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).

Incorporating One Task Extension (OTE) Technique

As presented in [27], one can further slow down execution when there is only one task in the ready queue and its worst case completion time (under the current speed) does not extend beyond the *next event* (next arrival/closest deadline of any task). For example, consider the following 2-task system: $C_1 = 100, P_1 = 200, C_2 = 300, P_2 = 600$. Again it is easy to see that $\bar{S} = S_{max} = 1.00$. The worst-case static schedule \mathcal{S}^{can} is shown in Figure 5 (left).

² The argument T_x passed to the procedure Speed-Reduce effectively represents a pointer to the process descriptor of T_x , from which one can access the variables S_x and $w_x^{S_x}$. The same applies to Figure 10.

Rules for GDRA

1. Compute \bar{S} (as in Section 3) and assign $\widehat{S}_{i,j} = \bar{S} \forall i, j$.^a
2. Initialize the α -queue to the empty list.
3. 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.
4. At every new arrival, insert into the α -queue, in the correct EDF* order, the information regarding the new instance of task $T_{i,j}$ with $rem_i(t) = w_i^{\widehat{S}_i}$.
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 $\epsilon_x(t)$ (the earliness of T_x)
 - 5.2. Reduce the speed of task T_x by giving T_x an extra Y time units:
 $S_x = \text{Speed-Reduce}(T_x, Y, S_x)$, 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.

^a A separate nominal speed variable per task instance is required by the aggressive algorithm (Section 5), which extends GDRA.

Figure 3: 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** $S_x < S_{min}$ **then** $S_x = S_{min}$
3. **return** S_x

Figure 4: Speed Reduction Procedure

In the actual execution, suppose that the schedule follows \mathcal{S}^{can} until $t = 200$ and T_2 completes early at $t = 200$. When the second instance of T_1 arrives at $t = 200$, the α -queue has the 2-

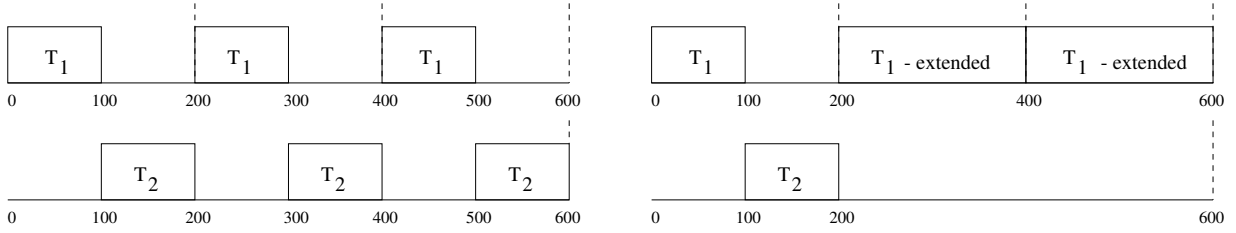


Figure 5: The static optimal schedule (left) and the actual schedule with OTE technique (right)

instance snapshot $\{(T_{1,2}, 200, 400, 100), (T_{2,1}, 0, 600, 200)\}$. Note that even though T_2 completed, $rem_2(200)$ is still 200 since the updating of the α -queue only reduced it by 100 units. At this point ($t = 200$), the earliness $\epsilon_1(200) = 0$ and DRA would not reduce the speed. But we can actually reduce the speed S_1 safely to 0.5, effectively adding the interval $[300, 400]$ for the execution of $T_{1,2}$, and $[500, 600]$ for the execution of $T_{1,3}$ (see Figure 5, right). Note that, at $t_1 = 100$, T_2 is also the only ready task and its earliness $\epsilon_2(100) = 0$, yet, it cannot reduce its speed since $t_1 + w_2^{S_2} > 200$ which is the time of the next event.

OTE technique can be used in conjunction with any scheduling policy: to improve (G)DRA, we add a new rule 5.3. 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 $Z = NTA - t - w_x^{S_x}(t) > 0$, set $S_x = \text{Speed-Reduce}(T_x, Z, S_x)$

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

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.

Let us illustrate the aggressive scheme by a simple example, in which we have $C_1 = C_2 = 25$, $P_1 = P_2 = 100$, $g(S) = S^3$. The static worst-case optimal schedule \mathcal{S}^{can} (Figure 6, left) is obtained by computing $\bar{S} = U_{tot} = 0.5$. Note that \mathcal{S}^{can} is optimal only when the actual required CPU cycles $AC_i = C_i$ for $i = 1, 2$, for which case the total energy consumption is $E_1 = 50(0.5)^3 + 50(0.5)^3 = 12.5$. In case that $AC_1 < C_1$ and/or $AC_2 < C_2$, \mathcal{S}^{can} is no longer optimal. For example, if $AC_1 = 15$ and $AC_2 = 20$ then starting with \bar{S} and applying the DR-OTE algorithm, yields the schedule shown in Figure 6 (right).

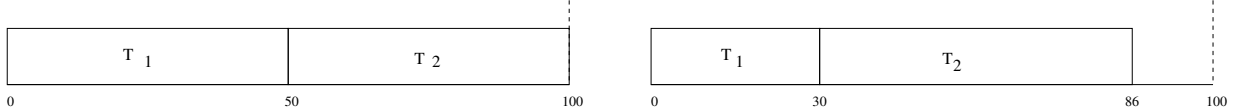


Figure 6: The static optimal schedule (left) and the schedule of DR-OTE (right)

Note that because of its actual low workload, T_1 completes at $t = 30$. Then the DR-OTE algorithm is able to dynamically reclaim unused 20 units of time, effectively reducing S_2 to $\frac{25}{50+20} = \frac{25}{70} = 0.35$. But since $AC_2 < C_2$, T_2 also completes well before the deadline. In this case, the total energy consumption is³ $E_2 = 30(0.5)^3 + 56(0.35)^3 + 14(0.1)^3 = 6.16$. However, by adopting an aggressive scheme, we can start executing T_1 with a speed lower than $S_1 = 0.50$, for example $S_1 = 0.35$. In this case, T_1 will complete at $t = 42.8$ and the dynamic reclaiming

³ Note that we are using the minimum CPU speed $S_{min} = 0.1$ in the interval $[86, 100]$

algorithm will further set S_2 to $\frac{25}{50+7.2} = \frac{25}{57.2} = 0.43$ (note that we cannot, at this point, set the speed S_2 any lower than this and still guarantee completion within the deadline). T_2 will complete at $42.8 + \frac{20}{0.43} = 89.3$. The schedule is depicted in Figure 7. The total energy consumption in this

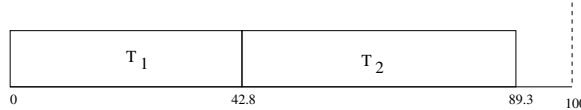


Figure 7: The schedule obtained via aggressive speed reduction

aggressive scheme is $42.8(0.35)^3 + 46.5(0.43)^3 + 10.7(0.1)^3 = 5.54275$, providing an additional 10% in power savings with respect to DR-OTE.

However, we should point out that under a very pessimistic scenario where $AC_1 = C_1$, the aggressive scheme would *have* to execute T_2 with high speed to prevent a deadline miss, resulting in a high energy consumption. We conclude this example by noting that it would not be possible to "aggressively" assign a speed lower than $\frac{25}{75} = 0.33$ to S_1 , since doing so and having a worst-case workload for T_1 might result in a deadline miss for T_2 , even under $S_2 = S_{max}$.

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} . This guarantee may come at the cost of increasing the CPU speed beyond \bar{S} later. However, any aggressive algorithm should (i) specify the amount of speed increase for different tasks, (ii) allow the use of other dynamic speed management techniques (e.g., DRA and OTE) in order to be able to gather the full benefit of aggressive schemes, and (iii) determine the *critical window* (the interval of time that the schedule is no longer ahead of \mathcal{S}^{can}), and prevent deadline misses during this window.

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.

Feasibility Guarantees 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 its effects upto the next event (arrival/deadline of any task). As the results in Section 6 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 generalize this technique in the following way: if T_1, T_2, \dots, 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, \dots, T_r). In fact, in this case, even T_{r+1} (if it exists) may provide a portion of its time allocation, if need arises. However, we must still guarantee that T_1, T_2, \dots, 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, \dots, T_r, T_{r+1} in the future through dynamic reclaiming. Finally, all these speed adjustments should adhere to the S_{min}, S_{max} and Sb bounds.

The details of the approach can be best illustrated with a simple example. Suppose at time t_1 , T_1 is the ready task with the highest EDF* priority, and that after the reclaiming step (5.2), its CPU allocation is $w_1^{S_1}$, and that its speed S_1 is still greater than both S_{min} and Sb . Further, assume that T_2, T_3 and T_4 are ready tasks with next highest priorities, such that $t_1 + \sum_{i=1}^3 w_i^{S_i} < NTA$ but $t_1 + \sum_{i=1}^4 w_i^{S_i} > NTA$ (see Figure 8). For the sake of simplicity, we will assume first that at $t = t_1$, the α -queue does not contain an element with EDF* priority between d_1^* and d_4^* ; we will comment on the relaxation of this assumption at the end.

During an aggressive move, we attempt to transfer up to Q units of CPU time to T_1 (and hence, decrease its speed); and it would still complete before *NTA* (Figure 8). In particular, T_2 and T_3 can provide a fraction of this amount (effectively, up to Z units), as long as their speed remains lower than S_{max} despite the increase. In addition, T_1 can try to transfer CPU time from

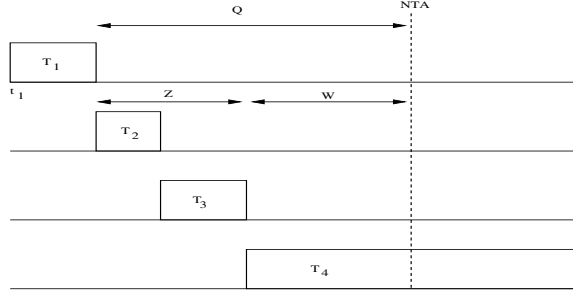


Figure 8: The schedule prior to speculative speed adjustment

T_4 , following T_2 and T_3 . However, this amount can *never exceed* $Q - Z = W$ units, regardless of the amount of previous transfers: Doing otherwise would shift the worst-case completion time of T_3 beyond NTA, with unpredictable consequences. In summary, we may aggressively try to transfer up to Q units of CPU time from T_2 and T_3 (in reality, the S_{max} constraint would allow a much smaller amount) and the amount transferred from T_4 can not exceed $Q - Z$ units.

Now, let us relax the assumption that the α -queue does not contain an element with EDF* priority between d_1^* and d_4^* at $t = t_1$. For example, assume that T_3 has already completed at $t = t_1$, though it is in the α -queue with (unused computation time) rem_3 . Further, assume that rem_3 is equal to the same amount of worst-case remaining execution time of T_3 in Figure 8. According to DR-OTE, T_1 would not be able to use the slack-time of T_3 which has lower EDF* priority. However, as it can be easily seen, T_1 can transfer this amount as well during an aggressive move, and the effects can be checked until the next event. A main difference with the case of previous paragraph is that the S_{max} constraint is irrelevant for the reserved CPU time of already completed tasks: T_1 can consume as much as possible until the next event. Needless to say, utmost care must be still exercised when a slack-time reservation exceeds the NTA boundary (such as T_4 above) and the transfer amount should not shift other ready tasks beyond NTA (i.e., it should be still less than $Q - Z$).

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**, shown in Figure 9.

The following points must be noted about the notation used in the algorithm: When 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, \dots, T_m , $2 \leq m \leq n$, in decreasing order of priorities. 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, \dots, S_m , and the next task arrival after t will occur at time NTA .

Note that, the aggressive speed reduction is performed only when the ready queue (in the actual execution) contains more than one task. Procedure *Speed-Increase* (Figure 10) 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 will remain unchanged and equal to \bar{S} . Moreover, it is possible to show the following (see [2] for the formal proof):

Proposition 4 *Aggressive-Speed-Adjustment* routine preserves the feasibility of the schedule.

We note that another approach while 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 feasibility would be preserved by doing so. To distinguish two variations, we will denote the original scheme and the new variation by **AGR1** and **AGR2**, respectively (or, AGR1 and AGR2, for short). The correctness of the new scheme follows from the correctness of AGR1, since AGR2 never slows down the processor more than AGR1.

Procedure Aggressive-Speed-Adjustment

- **begin**
- **if** $S_1 \leq \max\{S_{min}, k \cdot S_{optavg}\}$ **then** return;
- 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:

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

- Adjust Q in order not to extend beyond NTA :
if $NTA - t - w_1^{S_1}(t) < Q$ **then** $Q = NTA - t - w_1^{S_1}(t)$.
- $Q_{actual} = 0$ (already transferred slack amount).
- **if** $w_2^{S_2}(t) \geq Q$ **then begin** $r = 1; Z = 0$ **end**
 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 $T_2, \dots, T_{\min(m, r+1)}$ while reducing the speed of T_1 :
 - $j = 2$
 - **while** $j \leq \min(m, r + 1)$ **begin**
 - * **if** ($j < r + 1$) **then** $Requested_time = Q - Q_{actual}$
 else $Requested_time = Q - Z$
 - * **if** T_j is ready **then**
 - **begin**
 - $S_j = \text{Speed-Increase}(T_j, Requested_time, \widehat{S}_j)$ (see Figure 10)
 - $B = \left(\frac{S_j}{\widehat{S}_j} - 1\right) \cdot w_j^{\widehat{S}_j}$
 - $\widehat{S}_j = S_j$ (that is, commit to the new S_j as the default speed of that instance)
 - **end**
 - * **else if** T_j is completed but is in the α -queue **then** $B = \min(Requested_time, Rem_j)$
 - * $Q_{actual} = Q_{actual} + B$
 - * $j = j + 1$
 - **end**
 - $S_1 = \text{Speed-Reduce}(T_1, B, S_1)$
- **end**

Figure 9: Aggressive Speed Adjustment Procedure

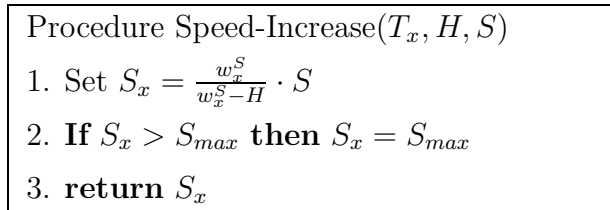


Figure 10: Speed Increase Procedure

6 Experimental Evaluation

In order to experimentally evaluate the performance of the proposed algorithms, we implemented a periodic scheduling simulator for EDF* policy. We implemented the following schemes for performance evaluation:

- **Static** uses constant speed \bar{S} , and switches to power-down mode (i.e., $S = S_{min}$) whenever there is no ready task;
- **OTE**: Static optimal speed scheme in conjunction with One Task Extension (but without dynamic reclaiming)
- **CC-EDF** (Cycle-conserving EDF): the reclaiming algorithm proposed by Pillai and Shin ([23])
- **LA-EDF** (Look-ahead EDF): the speculative algorithm proposed by Pillai and Shin ([23])
- **DRA**, which is implemented in two variations: with or without the OTE technique (DR-OTE and DRA, respectively)
- **AGR1** with the parameterized speed bound set to S_{optavg} (i.e., the aggressiveness factor k is set to 1)
- **AGR2** with the parameterized speed bound set to $0.9 \cdot S_{optavg}$ (i.e., the aggressiveness factor k is set to 0.9)⁴
- **Bound**, which is the *clairvoyant* algorithm that knows the exact *actual* workload in advance and adopts an optimal speed accordingly.⁵

⁴ The performance of AGR2 is almost identical to AGR1 when k is set to 1. However, we empirically determined that setting $k = 0.9$ can provide further savings for AGR2; see Section 6.2 for details.

⁵ Although **Bound** is not a practical algorithm (because no algorithm can predict the exact workload before hand), we found useful to include it as a yardstick in our experiment settings. Clearly, no dynamic voltage scaling algorithm (either inter- or intra-task) can offer a better performance than that of **Bound**.

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 first focused on the average energy consumption of 100 task sets, each containing 30 tasks. We repeated the experiments for 20- and 10-task sets as well. 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 or uniform probability distribution function. In case of the normal distribution, 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 [27]. The choice of this mean and standard deviation ensures that, on the average, 99.7% of the execution times fall in the interval $[BCET, WCET]$. In case of the uniform distribution, the mean is again set to $\frac{WCET+BCET}{2}$. For each task set we simulated the execution 10 times in interval $[0, LCM]$, where LCM is the Least Common Multiple of P_1, \dots, P_n , and we measured the average energy consumption per experiment using a cubic power/speed function [9, 11, 15, 14]. To investigate the effect of the specific form of the convex power consumption function, we also performed experiments with quadratic power/speed function. We comment on the results of this last set of experiments in Section 7.

6.1 General Trends

One remarkable outcome of the experimental evaluation is the following: Although the OTE scheme provides substantial improvements over techniques that continuously use S_{max} during the execution without reclaiming [27], 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) *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. The same observation holds for CC-EDF.

Effect of Utilization: Figure 11 shows the energy consumption of the techniques varying with the utilization of the task set under S_{max} , when $\frac{WCET}{BCET}$ is set to 5. We changed the utilization of the system between 20% (corresponding to low load) and 100% (maximum load). The results

are normalized with respect to **Static**. One can observe the following major patterns:

- The normalized energy consumption of all schemes is 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 and makes the CPU (still) fully utilized.
- 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 [27] 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.
- DRA, AGR1, AGR2, CC-EDF and LA-EDF have a definitive advantage over Static and OTE for all utilization values, providing additional energy savings that vary between 50% and 70%.
- The energy consumption of DRA is 17-20% lower than that of CC-EDF throughout the spectrum. It also outperforms LA-EDF, though the difference is around 7%.
- The results point to a consistent advantage of aggressive algorithms over DRA (and over other non-clairvoyant algorithms) throughout the spectrum. The improvement decreases as the utilization approaches 100%; at high utilization values all tasks assume a nominal (default) speed close to $S_{max} = 1.0$ and aggressive speed reduction is not always possible because of the maximum speed constraint. It is also interesting to observe the near-optimal performance of AGR1 and AGR2 especially in utilization values not exceeding 80%: the clairvoyant algorithm could provide only 10% improvement over aggressive algorithms in that part of the spectrum.
- AGR2 performs slightly better than AGR1; we present a detailed analysis of the performance of both algorithms in Section 6.2.

Effect of $\frac{WCET}{BCET}$ ratio: The simulation results confirmed our prediction that the energy consumption would be highly dependent on the variability of the 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 12. 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:

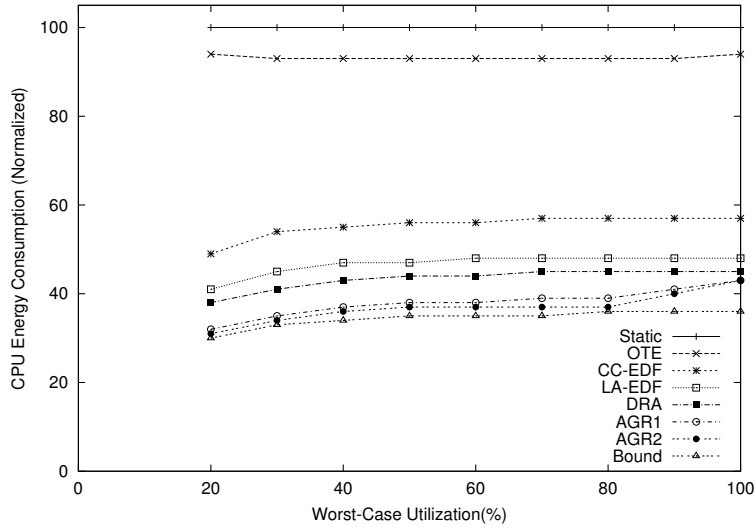


Figure 11: Normalized energy consumption (30 tasks). $\frac{WCET}{BCET} = 5$

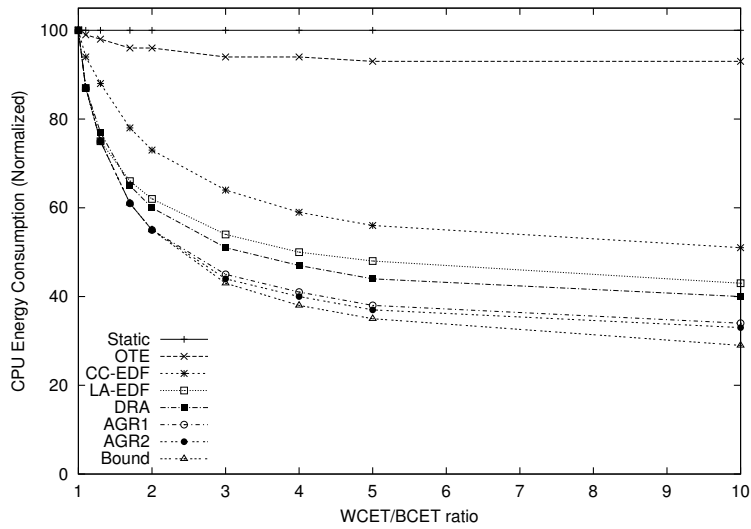


Figure 12: Effect of variability in actual workload (30 tasks); load = 60%

- 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 $\frac{WCET}{BCET}$), the algorithms are able to reclaim unused computation time and to save energy with respect to Static.
- The relative ordering of the schemes remains the same as we increase the $\frac{WCET}{BCET}$ ratio. Aggressive algorithms give the best energy savings, followed by DRA, LA-EDF and CC-EDF.

Increasing the ratio helps improving the relative performance of AGR, since speculative moves are justified more frequently.

- Once we increase the $\frac{WCET}{BCET}$ ratio beyond 4, power savings of DRA (and that of Bound) continue to increase, but the relative percentage of improvement is not as impressive as the case where that ratio is ≤ 4 . This is because of the expected workload of the system converges rapidly to 50% of the worst-case workload with increased $\frac{WCET}{BCET}$ ratio (remember that the mean of our probability distribution is $\frac{WCET+BCET}{2}$).

Effect of the number of tasks: To examine the effect of the number of tasks, we performed simulations for both 10- and 20-task sets. The figures illustrating the results are omitted due to the space limitations, but they can be found in [2]. In general, the trends are very similar to that of 30-task system. This is expected, since the main determinant of the workload is the variability in the actual workload. However, the percentages do differ from one case to the next. In particular, we see that all reclaiming and speculative speed reduction schemes provide a slightly larger advantage when the number of tasks increases, because there are more opportunities for reclaiming unused slack-times as well as for aggressively reducing the CPU speed with increasing number of tasks.

Effect of Uniform Distribution: We also experimented with generating actual execution times according to a uniform probability distribution, and observed the same relative ordering of schemes with similar general patterns. The normalized energy consumption of non-static schemes with varying utilization and $\frac{WCET}{BCET}$ ratio in the context of 30-, 20- and 10-task sets, is slightly higher for the uniform distribution [2]. This slight increase is due to the fact that the likelihood of having large execution times close to $WCET$ is higher for the uniform distribution. Since the large execution times adversely affect and limit the reclaiming opportunities, the performance is degraded, though marginally. However, the advantage of AGR is still consistent and the shapes of the curves remain rather similar to the normal distribution.

Comparing the mechanism of our algorithms to those of CC-EDF and LA-EDF schemes proposed in [23], we can make the following observations. When the reclaiming algorithm CC-EDF detects an early completion, the CPU speed is updated to reflect the new (lower) value of the instantaneous utilization. This effectively results in reducing the speed of *all* the ready tasks in *equal* proportions. Although it looks as a 'fair' distribution of slack, the better performance offered by DRA hints to the fact that a *greedy* approach is more effective when the actual

workload deviates from the worst-case scenario. Essentially DRA assigns the *entire* slack to the first low-priority task that is about to run. This greedy strategy pays off in the long run, since in many cases this 'lucky' task completes early as well and subsequent tasks are able to use its slack. Thus, in general, we can say that over-provisioning while distributing the available slack during the reclaiming phase hinders energy savings.

The same observation holds for LA-EDF. This algorithm uses the CPU time borrowed from future task instances to speculatively lower the instantaneous utilization over *multiple* tasks, thereby limiting the speed reduction that can be applied to the first task to run. In contrast, our aggressive algorithms 'steal' CPU time from other ready tasks to the exclusive benefit of the dispatched task. The experiments show that the performance is best when the aggressive slow-down is limited by the speed that would be optimal under the expected workload of the task set. Before concluding this section, we note that a recent study [12] provided an independent experimental comparison of these schemes, confirming the success of AGR and DRA when compared to other existing techniques.

6.2 Experimental Evaluation of AGR1 and AGR2

In this subsection, we present results of simulations performed to compare algorithms AGR1 and AGR2. The effect of the utilization and that of the $\frac{WCET}{BCET}$ ratio (denoted by r in the graphs) for 30-task sets generated with a normal distribution are shown in Figure 13 (left and right, in respectively). The curves and trends are similar for the other sizes of task sets. We also omit the uniform distribution results, since the relative performance of AGR1 versus AGR2 is extremely similar to the settings of normal distribution results. In these graphs, the performance curves of AGR1 are shown by lines, while AGR2 is depicted through points.

As it can be easily seen, the performance of aggressive schemes are hardly distinguishable. We observe that the *relative* performance of AGR2 over AGR1 tends to degrade with the increase with utilization. Remember that, in AGR2, a task is not allowed to reclaim or aggressively transfer CPU-time from other tasks if the speed bound Sb is reached, with the implicit assumption that there may be more suitable tasks that can further benefit from reclaiming. However, in low utilization (hence, in low nominal speed values), this decision may not be always justified since other tasks may be already executing at or close to S_{min} .

The effect of the aggressiveness level: Unlike the utilization and $\frac{WCET}{BCET}$ ratio, changing

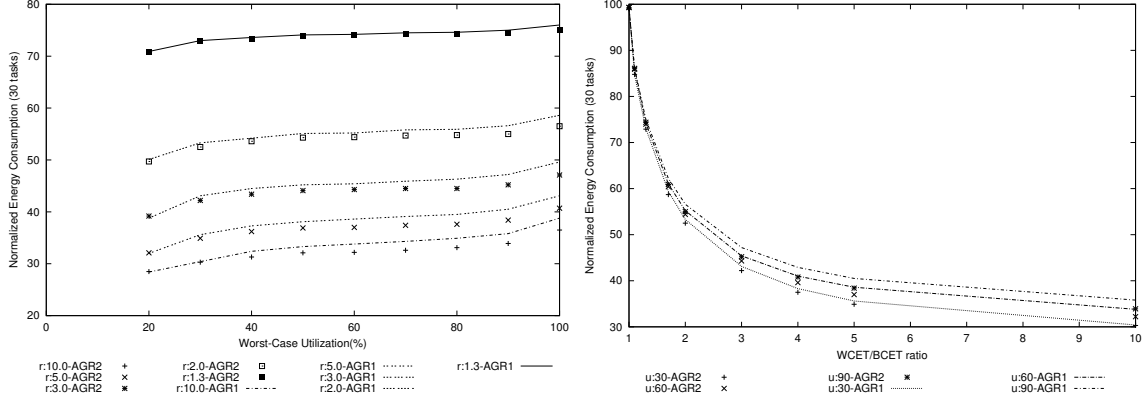


Figure 13: Effect of utilization (left) and effect of variability in actual workload (right)

the aggressiveness level deeply affects the results, as shown in Figure 14. The curves shown are for 60% utilization and $\frac{WCET}{BCET} = 5$; other parameter settings have very similar behavior. The performance of DRA and Static are insensitive to the parameter k . The maximum power savings 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 toward more 'balanced' aggressiveness levels, the aggressive schemes become preferable to DRA: AGR1 and AGR2 outperform 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 in this example. On the other hand, AGR2 converges to OTE (not shown in Figure 14) 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, the CPU speed is reduced only through the OTE

technique.

In general, 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%. The additional gains with respect to Static is around 10%. Exact k values that provide best energy savings for AGR1 and AGR2 as a function of the utilization are given in Table 2, in addition to the energy consumption values corresponding to these k values as percentage of the energy consumed by the static scheme. The best k values maximizing the performance of AGR1 and AGR2 are found to be rather insensitive to the $\frac{WCET}{BCET}$ ratio. As can be seen, AGR1 yields the best performance with $k = 1.00$ (almost) invariably. In contrast, keeping k in the range of $[0.9, 0.95]$ maximizes the energy savings of AGR2. Since AGR2 considers the speed bound even when reclaiming (prior to speculating), this indicates that its performance approaches (and even exceeds) that of AGR1 when used with smaller speed bounds (increased aggressiveness levels).

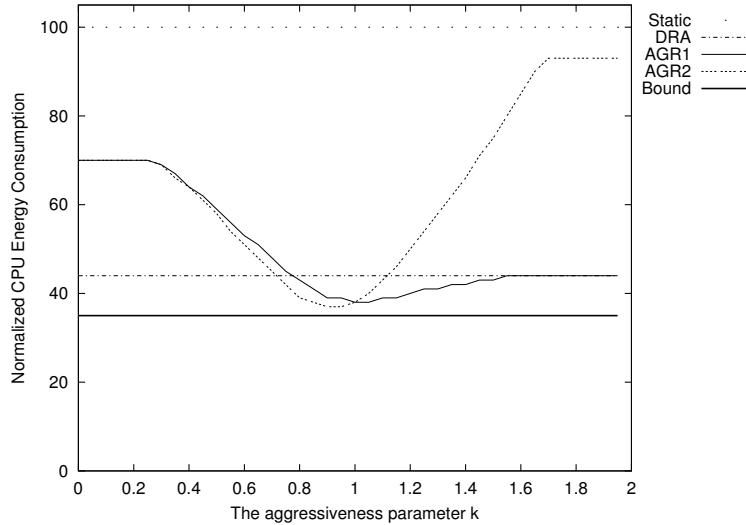


Figure 14: Normalized Energy Consumption as a function of the bounding factor k in Aggressive Schemes (30 tasks); utilization = 0.6, $\frac{WCET}{BCET} = 5$, normal distribution

7 Additional Considerations

Power/Speed Relation The simulation experiments we conducted were performed with cubic power/speed functions in accordance with other research work that appeared in literature [9, 11,

Utilization (%)	AGR1	AGR2	Best k (AGR1)	Best k (AGR2)
20	32	32	1.0	0.9
30	36	35	1.0	0.925
40	37	36	1.0	0.925
50	38	37	1.0	0.95
60	39	37	1.0	0.95
70	39	37	1.0	0.925
80	39	37	1.0	0.925
90	40	38	1.05	0.925
100	43	41	1.0	0.9

Table 2: The aggressiveness parameters maximizing energy savings as a function of utilization

15, 14]. However, we also investigated the effect of power/speed function’s specific form on the results we obtained. The results indicate that the relative ordering of schemes remains the same, though the improvement provided by non-static algorithms tends the increase (decrease) with increasing (decreasing) exponent in the power/speed function. This is to be expected, since the benefit of dynamically reclaiming/speculating tends to increase with that exponent. Figure 15 presents the results obtained with a 30-task set and normal distribution of execution times, assuming *quadratic* power/speed functions (compare with Figures 11 and 12).

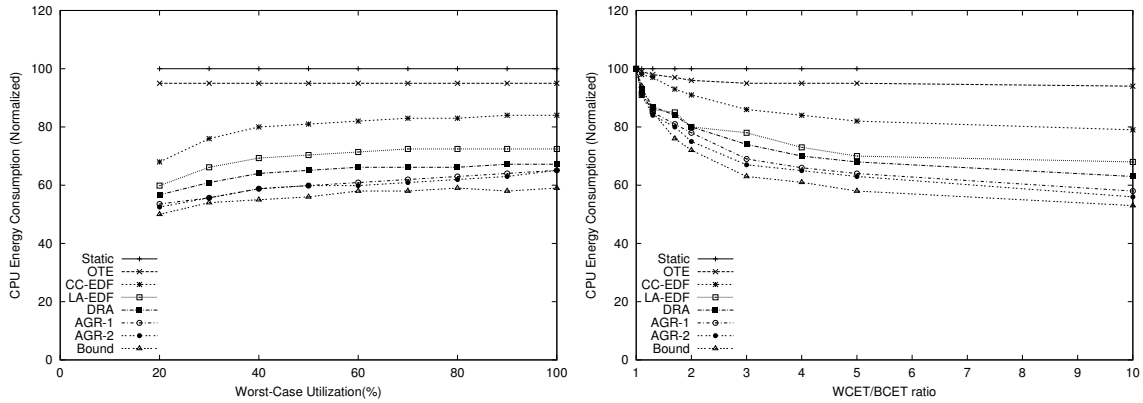


Figure 15: Normalized energy consumption as a function of utilization with $\frac{WCET}{BCET} = 5$ (left) and variation in workload with utilization = 60% (right) with quadratic power/speed function (30 tasks)

Continuous versus Discrete Speed Levels Throughout the paper, we made the assumption that the CPU speed can be changed continuously in interval $[S_{min}, S_{max}]$. Current technologies support only a finite number of speed levels [28], though the number of available speed levels is likely to increase in future. Our framework can be always adapted to these settings by choosing the lowest speed level that is equal to or greater than the value suggested by the algorithms. Our preliminary experimental results indicate that this simple approach results in an energy overhead of 15-17% with respect to continuous speed settings, when the number of available speed levels is only 5. When the number of speed levels exceeds 30, the difference reduces to 3%. More comprehensive recent studies that addresses this issue and other overhead sources in dynamic voltage scaling can be found in [19, 25].

8 Conclusions

In this paper we presented techniques for power-aware real-time computing through variable voltage scheduling. Our solution that aims to reduce the CPU energy consumption 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. All these techniques are formally proven to preserve the feasibility of the schedule even under a worst-case workload.

Our simulation results show that, when the actual workload deviates from the worst-case workload considerably, the reclaiming algorithm DRA (the component (b) above) can save up to 50% of the energy over the static algorithm, which takes into account the load in the system. It also offers a consistent advantage over other state-of-the-art inter-task voltage scheduling algorithms such as One Task Extension [27], Cycle-Conserving EDF [23] and Look-Ahead EDF [23]. Finally, our aggressive techniques further improve on DRA by 10-15% and the second speculative algorithm AGR2 approaches the theoretical lower-bound (that can only be achieved by a clairvoyant algorithm) by a margin of 10%. We concluded that, being too aggressive or not aggressive enough causes the algorithms to perform rather poorly. The experiments confirmed that a "balanced" aggressiveness level that aims to achieve the speed that would be optimal under the *expected* workload yields best results.

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. *Enhancing Performance and Fault Tolerance in Reward-Based Scheduling*. Ph.D. Dissertation, University of Pittsburgh, August 2001. Available on-line at <http://www.cs.gmu.edu/~aydin/dissertation>
- [3] 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.
- [4] 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.
- [5] Gruian, F. Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS Processors. In *International Symposium on Low Power Electronics and Design 2001*, August 6-7, 2001.
- [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] 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*
- [8] 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.
- [9] 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.
- [10] Intel, Microsoft, and Toshiba. Advanced Configuration and Power Management Interface (ACPI) Specification, 1999. www.intel.com/ial/WfM/design/pmdt/acpidesc.htm.
- [11] R. Jejurikar and R. Gupta. Energy Aware Task Scheduling with Task Synchronization for Embedded Real Time Systems. In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Grenoble, France, October 2002.
- [12] W. Kim, D. Shin, H.S. Yun, J. Kim, and S.L. Min. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In *Proceedings of the 8th Real-time Technology and Applications Symposium*, San Jose, 2002.
- [13] 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.
- [14] P. Kumar and M. Srivastava. Power-aware Multimedia Systems using Run-time Prediction. In *Proceedings of 13th International Conference on Computer Design*, January 2001.
- [15] T. Li and C. Ding, Instruction Balance and its Relation to Program Energy Consumption. In *Proceedings of International Workshop on Languages and Compilers for Parallel Computing*, Cumberland Falls, Kentucky, August 2001.
- [16] 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.

- [17] J. R. Lorch. A complete picture of the energy consumption of a portable computer. Masters thesis. *University of California, Berkeley*, December 1995.
- [18] 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.
- [19] B. Mochocki, X. Hu and G. Quan. A Realistic Variable Voltage Scheduling Model for Real-Time Applications. In *Proceedings of ICCAD 2002*.
- [20] 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.
- [21] 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
- [22] T. Pering and R. Brodersen. Energy Efficient Voltage Scheduling for Real-time Systems. In *Proceedings of the 4th Real-time Technology and Applications Symposium, WIP session*, 1998.
- [23] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low Power Embedded Operating Systems. In *Proceedings of the 18th Symposium on Operating System Principles*, October 2001.
- [24] J. Pouwelse, K. Langendoen and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. *7th International Conference on Mobile Computing and Networking (MOBICOM)*, Rome, Italy, July 2001.
- [25] S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority Real-time Systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, Washington D.C., May 2003.
- [26] D. Shin, S. Lee and J. Kim. Intra-task Voltage Scheduling for Low-Energy Hard Real-time Applications. In *IEEE Design and Test of Computers*, March 2001.
- [27] 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*, pp. 134-139.
- [28] <http://www.transmeta.com>
- [29] M. Weiser, B. Welch, A. Demers and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.
- [30] F. Yao, A. Demers and S. Shenker. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995.
- [31] H. Zeng, X. Fan, C. Ellis, A. Lebeck and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proceedings of ASPLOS 2002*, Oct. 2002.

APPENDIX: Supporting proofs

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

Proof: Throughout the proof, we will consider the actual CPU schedule as a *sequence of task execution segments*. A segment starts when a given task T_a is dispatched (say at $t = t_a$), and ends (at $t = t_b$) when T_a releases the CPU either by completion or by preemption. In other words, during the time

interval (segment) $[t_a, t_b]$, the CPU is allocated continuously to T_a . We will use induction to prove the validity of the statement for every such execution segment (on which a given task runs). **Note that we will need to show the validity of the proposition for each segment and for all the tasks that are (or incidentally, may become) ready during this segment.**

First, observe that initially (at $t = t_0$) all tasks are ready and the α -queue contains their $w_i^{\widehat{S}_i}$ values in the rem_i fields. Assume that T_1 has the highest EDF* priority at this time, and it will run with the speed \widehat{S}_1 until it completes at $t = t_1$ (note that the identical ready times and EDF* imply that it won't be preempted). At any time t in the interval $[t_0, t_1)$, $rem_1 = w_1^{\widehat{S}_1}$ since $S_1 = \widehat{S}_1$. Since by definition T_1 will not execute for more than its worst-case workload, at the time of its completion at $t = t_1$, $rem_1 \geq w_1^{\widehat{S}_1} = 0$. For every other task T_j waiting in the ready queue during this interval, rem_j will remain the same as $w_j^{\widehat{S}_j}$ since T_j did not have a chance to execute. Thus, the proposition holds for the first execution segment $[t_0, t_1]$.

Since the base case is established, now assume that the statement remains valid for the first k execution segments and consider the $(k + 1)^{st}$ segment, during which T_x executes. Hence, T_x runs continuously from t_{k-1} to $t_k = t_{k-1} + \Delta_t$, and it relinquishes the CPU either by completion or the arrival of a higher priority task (preemption). From the induction assumption, $rem_i \geq w_i^{\widehat{S}_i}$ for every ready task T_i at $t = t_{k-1}$.

We first focus on the changes in of rem_x and $w_x^{\widehat{S}_x}()$ in interval $[t_{k-1}, t_k]$: When we consider the speed decrease ratio in the procedure 'Speed-Reduce' and $w_x^{\widehat{S}_x}(t_{k-1})$, we can easily conclude that T_x can not run more than $Y + w_x^{\widehat{S}_x}(t_{k-1})$ units, that is $0 \leq \Delta_t \leq Y + w_x^{\widehat{S}_x}(t_{k-1})$.

The value of $w_x^{\widehat{S}_x}(t_{k-1} + \Delta_h)$ where $0 \leq \Delta_h \leq \Delta_t$ can be easily computed. During the interval $[t_{k-1}, t_k]$ we execute with a speed $S'_x = \widehat{S}_x \cdot \frac{w_x^{\widehat{S}_x}(t_{k-1})}{w_x^{\widehat{S}_x}(t_{k-1}) + Y}$; hence at $t_h = t_{k-1} + \Delta_h$,

$$\begin{aligned} w_x^{\widehat{S}_x}(t_h) &= w_x^{\widehat{S}_x}(t_{k-1}) - \frac{S'_x}{\widehat{S}_x} \Delta_h \\ &= w_x^{\widehat{S}_x}(t_{k-1}) \left[1 - \frac{\Delta_h}{Y + w_x^{\widehat{S}_x}(t_{k-1})} \right] \geq 0 \end{aligned}$$

for the preemption case, or $w_x^{\widehat{S}_x}(t_h = t_k) = 0$ for the completion case.

What is the value of $rem_x(t_{k-1} + \Delta_h)$? Let us define by $\beta_h(t, x)$ the sum of all rem_i values in the α -queue at time t , such that T_i has strictly higher EDF* priority than T_x . Observe that, by definition, $\epsilon_x(t) = \beta_h(t, x) + rem_x(t) - w_x^{\widehat{S}_x}(t)$. Hence:

$$rem_x(t_{k-1}) = \epsilon_x(t_{k-1}) + w_x^{\widehat{S}_x}(t_{k-1}) - \beta_h(t_{k-1}, x) \quad (1)$$

As time elapses, we consume from the α -queue starting from the head. We distinguish two cases:

- If $\Delta_h \leq \beta_h(t_{k-1}, x)$, we will only consume the slack of "higher" priority tasks in the α -queue, and rem_x will remain intact. Hence, using also induction assumption, we can obtain: $rem_x(t_{k-1} + \Delta_h) = rem_x(t_{k-1}) \geq w_x^{\widehat{S}_x}(t_{k-1}) \geq w_x^{\widehat{S}_x}(t_{k-1} + \Delta_h)$ for this special case.

- If $\beta_h(t_{k-1}, x) < \Delta_h \leq \Delta_t \leq Y + w_x^{\widehat{S}_x(t_{k-1})}$, we will start decreasing rem_x value only at $t = t_{k-1} + \beta_h(t_{k-1}, x)$. At $t = t_{k-1} + \Delta_h$, $rem_x(t_{k-1} + \Delta_h) = rem_x(t_{k-1}) - (\Delta_h - \beta_h(t_{k-1}, x))$, and using the equation (1), we get:

$$\begin{aligned}
rem_x(t_{k-1} + \Delta_h) &= \epsilon_x(t_{k-1}) + w_x^{\widehat{S}_x(t_{k-1})} - \Delta_h \\
&\geq Y + w_x^{\widehat{S}_x(t_{k-1})} - \Delta_h = (Y + w_x^{\widehat{S}_x(t_{k-1})}) \left[1 - \frac{\Delta_h}{Y + w_x^{\widehat{S}_x(t_{k-1})}} \right] \\
&\geq w_x^{\widehat{S}_x(t_{k-1})} \left[1 - \frac{\Delta_h}{Y + w_x^{\widehat{S}_x(t_{k-1})}} \right] = w_x^{\widehat{S}_x(t_{k-1} + \Delta_h)}
\end{aligned}$$

Thus, for task T_x , we have $rem_x(t_{k-1} + \Delta_h) \geq w_x^{\widehat{S}_x(t_{k-1} + \Delta_h)} \geq 0$ for $0 \leq \Delta_h \leq \Delta_t \leq Y + w_x^{\widehat{S}_x(t_{k-1})}$. A geometric interpretation of the changes in $rem_x()$ and $w_x^{\widehat{S}_x}$ during the time interval $[t_{k-1}, t_{k-1} + \Delta_t]$ is presented in Figure 16.

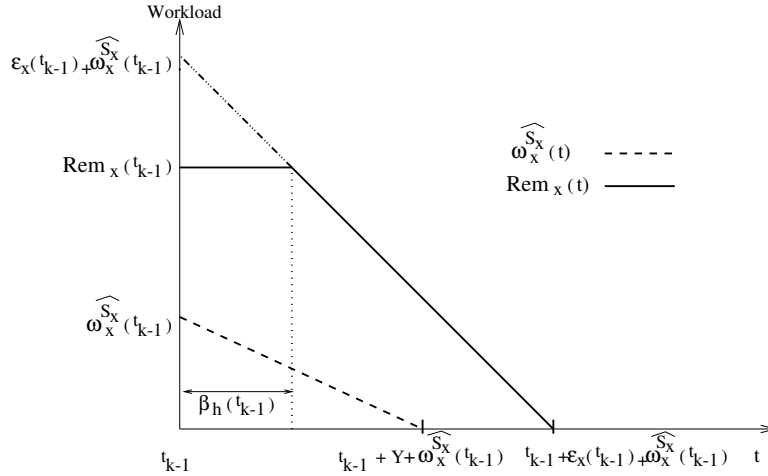


Figure 16: The changes in the workload

For any other task T_y which was also ready at $t = t_{k-1}$, but whose execution was delayed due to the high priority computation of T_x , $rem_y(t_{k-1} + \Delta_h) = rem_y(t_{k-1})$, since this value (correctly represented in the α -queue) can not be decreased before rem_x reaches zero (which can not happen before T_x completes, as we have shown above). But T_y was not executed either, hence $w_y^{\widehat{S}_y}(t_{k-1}) = w_y^{\widehat{S}_y}(t_{k-1} + \Delta_h)$. The induction assumption is that $rem_y(t_{k-1}) \geq w_y^{\widehat{S}_y}(t_{k-1})$, hence $rem_y(t_{k-1} + \Delta_h) \geq w_y^{\widehat{S}_y}(t_{k-1} + \Delta_h)$ as well. Finally, for any other task T_q which may arrive (and wait) while T_x is executing, clearly rem_q will remain equal to the $w_q^{\widehat{S}_q}$. From these results, we can infer that at any time t in the $(k+1)^{st}$ execution segment, $rem_i(t) \geq w_i^{\widehat{S}_i(t)}$ for any ready task T_i . Having established the statement for the $(k+1)^{st}$ segment, we prove the theorem. \square