Energy-Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints

Ajay Dudani, Frank Mueller, Yifan Zhu
Department of Computer Science/
Center for Embedded Systems Research
North Carolina State University
448 EGRC, Raleigh, NC 27695-7534

ABSTRACT

Embedded systems have limited energy resources. Hence, they should conserve these resources to extend their period of operation. Recently, dynamic frequency scaling (DFS) and dynamic voltage scaling (DVS) have been added to a various embedded processors as a means to increase battery life. A number of scheduling techniques have been developed to exploit DFS and DVS for real-time systems to reduce energy consumption. These techniques exploit idle and slack time of a schedule. Idle time can be consumed by lowering the processor frequency of selected tasks while slack time allows later tasks to execute at lower frequencies with reduced voltage demands.

Our work delivers energy savings beyond the level of prior work. We enhance the earliest-deadline first (EDF) scheduling to exploit slack time generated by the invocation of the task at multiple frequency levels *within the same invocation*. The technique relies strictly on operating system support within the scheduler to implement the approach. Early scaling at a low frequency, determined by a feedback mechanism and facilitated by a slack-passing scheme, capitalizes on high probabilities of a task to finish its execution without utilizing its worst-case execution budget. If a task does not complete at a certain point in time within its low frequency range, the remainder of it continues to execute at a higher frequency. Our experiments demonstrate that the resulting energy savings exceed those of previously published work by up to 34%. In addition, our method only adds a constant complexity at each scheduling point, which has not been achieved by prior work, to the best of our knowledge.

1. INTRODUCTION

Energy consumption is a major concern for embedded systems. The availability of services provided by mobile devices powered by batteries is clearly limited by the amount of power drawn from the batteries over time. But energy consumption is also a cost factor for non-battery powered systems since the operational costs of embedded systems running non-stop may be significant. Energy conservation should be a central objective in the design of such systems.

Another factor is posed by the objective of ensuring proper operational behavior of embedded systems. Hard real-time systems in particular, *i.e.*, systems with strict temporal constraints on the execution of tasks, must produce results on time to ensure operational safety and prevent potentially catastrophic effects. Real-time schedulability theory provides firm guarantees for periodic executions of a set of tasks on uniprocessors in terms of assurances to meet deadlines [26, 2, 35, 3, 5, 37]. The schedulability theory for real-time systems relies on *a priori* knowledge of the worst-case execution time (WCET) of hard real-time tasks to check if the deadline of a task can be met. A safe upper bound on the WCET of a task can be provided through static analysis, dynamic analysis or even a combination of both techniques [34, 30, 15, 41, 24, 16, 1, 22, 23, 9, 29, 38]. Regardless of the methods utilized to obtain the WCET of tasks, experiments show a wide variation between longest and shortest execution times for many embedded applications. In [38], execution times of real-world embedded tasks vary by as much as 87% relative to their measured WCET. Specifically, variations of 78%, 87% and 74% where observed for graphics, defense, automotive tasks while 30% and 89% variations were reported for the benchmarks matrix and sort, respectively, with an average case close to the mean of the extreme execution times (worst-case and best-case times). Wolf reports variations of up to 85% for different image recognition tasks, such as shape fitting and graph matching, commonly performed in embedded systems [40].

Recent trends in embedded architecture provide support for dynamic frequency scaling (DFS) and dynamic voltage scaling (DVS) at the processor level. By modulating the frequency level, the speed of a processor can be throttled or sped up to meet actual computing demands. DFS and DVS are generally combined in the sense that lower frequencies allow reduced voltage levels to operate a processor. Since the energy consumption scales linearly and quadratically with frequency and voltage modulations, respectively, the combination of DFS and DVS can result in significantly lower power consumption.

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

DFS and DVS techniques are particularly attractive to real-time systems for two reasons. First, real-time systems commonly execute periodic tasks, which implies that they cannot enter low-power sleep modes that put the processor in a standby state, effectively halting any execution. Second, real-time requirements force system designers to choose embedded processors that are powerful enough to meet the worst-case execution demands although these demands may rarely occur. As a result, the system utilization will often be low and energy consumption can be high under such circumstances to ensure operational safety. By exploiting DFS/DVS techniques, we can guarantee hard deadlines of real-time systems and lower power consumption at the same time. This opens new opportunities for reduced operational costs and for embedded applications requiring longer battery life.

Prior work has shown the potential to save energy by combining these scaling techniques with operating system scheduling, and significant savings have been reported for general-purpose computing systems [10, 13, 19, 27, 31, 39, 33, 12] as well as real-time systems [17, 18, 20, 36, 32, 14, 7, 28, 11] detailed in the related work section. Our work goes beyond the techniques explored for real-time systems in these previous studies. We contribute a novel approach for exploiting the slack time of a schedule. Slack time is generated by actual executions of tasks that complete under budget with respect to their WCET.

Past methods distributed idle and slack time of the subsequent execution of following tasks. Our method exploits slack time generated by the invocation of the task at multiple frequency levels *within the same invocation*. The technique relies strictly on operating system support within the scheduler to implement the approach. Early scaling at a low frequency capitalizes on high probabilities of a task to finish its execution without utilizing its worst-case execution budget. The scaling level is determined by a feedback mechanism that utilizes the average execution time of past task executions to find a suitable frequency setting likely to complete the task for the current invocation. If a task does not complete at a certain point in time within its low frequency range, scaling is repeated, and another portion of the task executes at a slightly higher frequency until finally the remainder of the task continues to execute at the highest frequency in the worst case. In addition, scaling is facilitated by a slack-passing scheme. Our experiments demonstrate that the resulting energy savings exceed those of previously published work.

The paper is structured as follows. In Chapter 2, we introduce the basic terminology of real-time and power-aware scheduling in general. We then develop a new scheduling technique in Chapter 3. Chapter 4 discusses the scheduling algorithm and examples to illustrate the benefits of our approach in contrast with past work. Chapter 5 presents an algorithmic description of our technique. Chapter 6 presents results to demonstrate the performance of our algorithm under different load conditions. Chapter 7 discusses related work and Chapter 8 summarizes our efforts.

2. POWER-AWARE SCHEDULING

Embedded systems with temporal constraints traditionally use real-time scheduling techniques. While legacy systems often rely on cyclic executives [4], the static nature and inflexibility in terms of scheduling of such an approach makes it hard to conserve energy in a flexible manner. We focus on dynamic scheduling paradigms, such as earliest deadline first (EDF) [25], to incorporate power awareness into the dynamic decision process. As in the original work, we assume a given set of tasks T_i with deadlines equal to their periods P_i, a maximum computational budget (WCET) of C_i and an actual execution time of c_{ij} for the j^{th} instance of T_i. By always scheduling the task with the earliest deadline first in a preemptive fashion, a schedule is feasible (*i.e.*, none of its deadlines are missed) if the following necessary and sufficient condition holds.

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{1}$$

Building on this result, Pillai and Shin [32] extended Inequation 1 to EDF with DFS modulation. As discussed before, lower processor frequencies allow a reduction of the voltage as well (DVS). Since the power consumption is proportional to the frequency and the square of the voltage, DVS can result in considerable power savings. For DVS under EDF, let $\alpha = \frac{f_i}{f_m}$ denote the scaling factor representing the fraction of the current processor frequency f_i over the maximum frequency f_m:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \alpha \tag{2}$$

This schedulability test was exploited in [32] to exploit idle time and slack time due to early completion of a prior task in order to scale frequencies (α) for later tasks. One of the limiting factors of this work was that an entire task was scaled to execute at a fixed frequency, and another one was the lack of feedback mechanisms. Our approach takes a more aggressive approach, as detailed in the next section.

3. FEEDBACK EDF SCHEDULING

In the introduction, we cited various studies that observed actual execution times or tasks well below the WCET. Past approaches of real-time scheduling with DVS assumes that a task executes at one frequency level. With this restriction, EDF scheduling requires that scaling is constrained to a frequency level high enough to not exceed a certain utilization threshold, as expressed by Equation 2. If, as commonly observed, a task only consumes 50% of its execution time, the processor frequency could have been cut in half. But without knowing the actual execution time in advance, we cannot reduce frequencies to such a low level if a schedule should remain feasible.

In the following, we develop a framework for greedily scheduling real-time tasks under DFS/DVS and guaranteeing deadlines. One objective of our approach is to aggressively and optimistically reduce power consumption as early as possible. A second objective is to exploit the existing invocations of the scheduler during execution to perform frequency scaling so that DFS/DVS essentially comes for free (other than the calculation of frequency levels). The third objective is to utilize feedback mechanisms for selecting appropriate scaling levels in order to reduce the energy consumption for task completion at a certain scaling level.

Our scheduling methods takes a novel approach to frequency scaling for EDF. Instead of assuming uniform scaling over all *future* tasks as in previous work, we scale the *current* task T_k (incidentally also the task with the earliest deadline). The remaining tasks are assumed to execute at maximum frequency f_m, *i.e.*, $\alpha = 1$. This can be expressed as:

$$\alpha^{-1}\frac{C_k}{P_k} + \sum_{i \in \{1,\ldots,n\} \setminus \{k\}} \frac{C_i}{P_i} \leq 1 \qquad (3)$$

The motivation for only scaling the current task is that average execution times are typically smaller than worst-case execution times C_i. If the current task finishes early, its slack can be used by the next task to scale frequencies again, and so on. Hence, early scaling is likely to leave enough potential for later scaling.

Slack Passing

The challenge in DVS scaling for EDF scheduling is posed by the ever changing precedence on scheduling tasks based on the shortest deadline. If a task is scaled, its execution may exceed the equivalent WCET without scaling. Past schemes determined the scaling factor based on variations of the EDF utilization test. Their complexity was $O(n)$ in the number of tasks n at each scaling point, *i.e.*, at each task release.

We promote a simple approach of slack passing with a constant complexity for scaling. Slack passing is based on the observation that slack generated by early completion of a prior task can be passed on to the next task if this next task was released prior to the slack origination point. This is depicted in Figure 1.

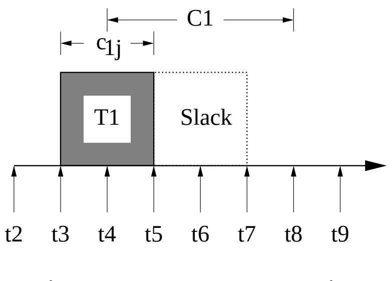


Figure 1: Slack Generation

Let task T_1 with WCET C_1 execute its j^{th} invocation with an actual execution time of c_{1j}. The figure indicates that C_1 spans $t4..t7$ in the worst-case EDF schedule. Let us assume its release time and deadline were T2 and T9, respectively. During the actual schedule, T_{1j} was invoked at T3 due to early completion of a prior task (not depicted). The generated slack is $C_1 - c_{1j}$. If other tasks are released within the interval $t2..t9$, they then can utilize all of this slack ($r_{ij} \leq t5$), part of the slack ($t5 \leq r_{ij} \leq t9$) or none of this slack ($\geq t9$) assuming these tasks have a deadline $\geq t9$. This is based on the observation that the time allotted to T_1 that remained unused is available within the reservation scheme for EDF based on the worst-case scenario. Consider the j^{th} invocation of an arbitrary T_i with an absolute deadline of d_{ij}. Let f_{ij} and F_{ij} be its absolute actual and worst-case completion (finish) times, respectively. Furthermore, let r_{ij} be its absolute release time, and let I_{ij} be its worst-case invocation time. Then, the amount of slack s_{cj} generated by a previous task T_{pk} that can be consumed by the current task T_{cj} is defined as:

$$s_{cj} = \begin{cases} C_p - c_{pk} & if\ r_{cj} \leq I_{pk} + c_{pk} \\ F_{pk} - r_{cj} & if\ I_{pk} + c_{pk} < r_{cj} < F_{pk} \\ 0 & if\ r_{cj} \geq F_{pk} \end{cases} \qquad (4)$$

Equation 4 only defines the relation between any two tasks' invocations. This scheme can be extended to transitive slack passing by keeping track of slack consumption of the current task. Any slack passed by T_{pk} to T_{cj} allows the latter task to be activated prior to its worst-case scenario. Hence, any execution without preemption up to s_{pk} units of time contributes to the allotment of execution budget of the previous task (or tasks). Thus, at task completion time f_{cj} (or at an earlier preemption point at time t) we only need to account for the actual execution *in excess of the slack*, denoted by c'_{cj}:

$$c'_{cj} = max(0, c_{cj} - s_{pk}) \qquad (5)$$

We can then calculate the slack T_{cj} can pass to its successor task T_{ni} according to Equation 4 by substituting these tasks with T_{pk} and T_{cj}, respectively, and by substituting c'_{cj} for c_{pk}:

$$s_{ni} = \begin{cases} C_p - c'_{cj} & if\ r_{ni} \leq I_{cj} - c'_{cj} \\ F_{cj} - r_{ni} & if\ I_{cj} + c'cj < r_{ni} < F_{cj} \\ 0 & if\ r_{ni} \geq F_{cj} \end{cases} \qquad (6)$$

Notice that transitive slack passing is only safe in the absence of preemption. This shortcoming is addressed next.

Preemption Handling

If a task T_{cj} is preempted before it can complete, it obviously cannot generate any slack based on its own execution since the remaining time is not known. However, slack passed to this task may not have been exhaustively utilized yet. Let c_a be the portion of the execution of the task prior to the preemption point. Then, we can allocate *prior slack* s_{pk} within the last units of c_a, as depicted in Figure 2. This slack s_{cj}, calculated as

$$s_{cj} = min(s_{pk} - c_a, c_a) \qquad (7)$$

can be passed to the preempting task T_{ni}. If we consider frequency scaling as well, Equation 7 changes since execution

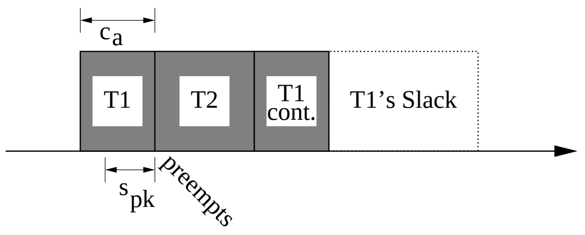


Figure 2: Slack Generation at Preemption

at a lower scaling factor α only contributes to a fraction of the equivalent budget at the highest frequency:

$$s_{cj} = min(s_{pk} - \frac{c_a}{\alpha}, \frac{c_a}{\alpha}) \tag{8}$$

Slack passing at preemption follows a greedy scheme based on the same assumptions again. We pass as much slack as possible to scale the preempting task aggressively. And we speculate on early completion of the preempting task to aggregate more slack. When the preempted task is resumed, this aggregated slack can be utilized by the next EDF task, such as for the remaining execution of T_{cj} or a task with an even earlier deadline.

Greedy Task Partitioning and Feedback for Ideal Scaling

In order to determine the best scaling factor α that allows a task to complete execution at this level and still guarantees a feasible schedule, we scale a portion of task T_k. We take the approach of *splitting* the next task T_k into two subtasks T_A and T_B, a method previously not employed for EDF. T_B executes at maximum frequency f_m while T_A can be scaled down to an arbitrary α level.

$$C_k = C_A + C_B \tag{9}$$

We can now formulate the constraints for frequency scaling based on slack passing and task splitting. Let s_k be the slack passed to T_k. Then,

$$\frac{C_A}{\alpha} + C_B = C_k + s_k \tag{10}$$

denotes the scaling of C_A units at α, including the utilization of aggregated slack. By substituting C_B in Equation 10 using Equation 9 and solving for α, we get:

$$\alpha = \frac{C_A}{C_A + s_k} \tag{11}$$

There is one pitfall in our assumption to scale task T_k. We assume that we can exploit *all* the slack for scaling. However, the task's absolute deadline d_{kj} may constrain us. Let *now* be the current time. EDF scheduling guarantees that $now + C_k \leq d_{kj}$. But we can only scale up to a duration bounded by the difference between the WCET and the deadline. We can reassess the portion of slack s'_k that can be utilized by the current task as:

$$s'_k = min(d_{kj} - (now + C_k), s_k) \tag{12}$$

Hence, only after bounding the slack to s'_k (and substituting its value for s_k in Equation 11) is it safe to combine scaling with slack passing.

Equation 11 is based on the assumption that scaling can only be based on the WCET C_k. Our objective, in contrast, is to finish execution of T_k within C_A units *on the average*.

We use feedback about the average execution time within each task to anticipate the required budget for future execution times. Our scheme simply aggregates actual execution times over task invocations to calculate an average time C_{avg} of past CPU activity (scaled to the maximum frequency level). Hence, by substituting C_{avg} for C_A in Equation 11, we can obtain an ideal scaling factor for a single frequency setting that, on the average, allows us to complete the task at this setting.

Actual Scaling Factor

Realistic processors typically support different frequency settings in certain fixed-rate increments. Hence, the minimal scaling factor α obtained through Equation 11 may not match any fixed frequency setting. We choose the next higher setting since we intend to finish our execution *on the average* under the current scaling level. We select an α' corresponding to a fixed rate frequency, obtained from the discrete frequency levels $\{f_1, ..., f_m\}$, such that $\alpha' \geq \alpha$ with α from Equation 11, where f_1 and f_m are the minimum and maximum frequency levels, respectively (the slack s_k is substituted by its portion before the deadline s'_k).

$$\alpha' = min\left\{ \frac{f_1}{f_m}, \frac{f_2}{f_m}, \ldots, \frac{f_m}{f_m} \;\middle|\; \frac{f_i}{f_m} \geq \frac{C_{avg}}{C_{avg}+s'_k} \right\} \tag{13}$$

Scheduling the split task T_k then amounts to scaling the initial execution of T_A down to α' and creating a timer interrupt at the end of its maximum execution C_A. If the actual execution of T_k completes within T_A, there is no overhead for this method. If T_k cannot complete in T_A, the scheduler is triggered by the timer interrupt at the end of T_A. The scheduler then increases the frequency to f_m and dispatches T_B to still complete T_k on time.

Idle Time Utilization

The last challenge is to exploit idle time within this scheme. Since EDF is a dynamic scheduling policy where tasks are scheduled according to deadlines, it becomes challenging to anticipate the actual execution slots for tasks when execution time varies. We take a novel approach that is intriguing, not just because of its simplicity, but also because it naturally fits our slack-passing scheme. Embedded systems with timing constraints may, in theory, be constructed with task sets of up to 100% utilization. In practice, the worst-case utilization of realistic systems is generally lower than 100%.

In an under-utilized system, we simply add another task, the idle task T_{n+1}. This task fills the gap between the utilization of the value obtained by Equation 1 and 100%. In other words,

$$P_{n+1} = P_1, C_{n+1} = P_1(1-U), c_{n+1} = 0.$$

It is called the idle task because its actual execution time is zero, even though its WCET is not. Our slack-passing scheme exploits the difference between these times to generate slack in the amount of C_{n+1} whenever the idle task is released. It never actually runs, but it indicates where the reservation slots for idle time are within an EDF schedule. We choose a period of the idle task equal to the shortest period of any task in the task set. This ensures that at least one idle invocation lies within the invocation of any other task, which allows other tasks to have at least one source for slack. The idle task is special in its algorithmic handling in that future invocations can provide slack for regular tasks if the worst-case idle invocation lies within the invocation of another task. This handling is orthogonal to future slack stealing at preemption, as discussed before. We explain idle slack stealing in more detail in the algorithm and an example.

We now turn to a description of the algorithm.

4. ALGORITHM AND EXAMPLE

An algorithmic description of the optimistic scheduling technique derived in the last section is depicted in Figure 3. We use the following notation:

- T_{ij}: the j-th instance of task T_i
- ij, pk, nl, ab : indices for the current, previous, next and idle jobs, respectively, relative to T_{ij}
- r_{ij}: the release time of T_{ij}
- d_{ij}: the deadline of T_{ij}
- C_i: the WCET of T_i (without scaling)
- c_{ij}: the actual execution time of T_{ij} up to now (with scaling)
- $left_{ij}$: the remaining execution time of T_{ij} (without scaling)
- $slack_{ij}$: future slack (preallocated at preemption)

Initially, the overall system utilization is determined to configure the idle task (see label (0) in Figure 3). At each scheduling point for a task activation, the scaling level is calculated and the scaled task portion is scheduled. If a task was preempted, the newly released task receives its recalculated slack prior to scaling. Slots from idle jobs increase the slack. We omitted partial credit for idle jobs in legal intervals (as discussed in the subsection titled "Slack Passing") to keep the presentation simple (1). In the absence of slack, there is no scaled portion, and the task proceeds to execute at the highest frequency. Otherwise, a timer interrupt is set at the end of the scaled portion.

At task preemption, future slack for the completion of T_{ij} is calculated, and the remaining slack is adjusted. Future slack is reserved in idle slots within two ranges (2). First, we attempt to reserve slots part the deadline of the preempting jobs since the preempting deadline is an earlier one that cannot utilize these future slots. If this is not sufficient, any other legal idle slots are reserved. Again, partial credit for idle slots is omitted.

```
Procedure Initialization
    for each $T_k \in \{T_1, T_2, \dots, T_n\}$ do
        $C_{avg\_k} \leftarrow C_k/2$
        $left_{k0} = C_k$
    $U \leftarrow \frac{C_1}{P_1} + \frac{C_2}{P_2} + \dots + \frac{C_n}{P_n}$
    $P_{n+1} \leftarrow P_1, C_{n+1} \leftarrow P_1 \times (1-U), c_{n+1} \leftarrow 0\}$      (0)
    let $slack \leftarrow 0$

Procedure TaskActivation($\mathbf{T_{ij}}$)
    if processor was idle for $d$ then
        $slack \leftarrow slack - d$
    if $T_{ij}$ was preempted/interrupted then
        $slack \leftarrow slack + slack_{ij} - left_{ij}$
    forall $T_{ab}$ idle task jobs in $d_{pk}..d_{ij}$ do
        $slack \leftarrow slack + C_a$                    (1)
    $\alpha\prime \leftarrow \min\{\frac{f_1}{f_m}, \dots, \frac{f_m}{f_m} | \frac{f_i}{f_m} \geq \frac{C_{avg\_i}}{C_{avg\_i}+slack}\}$
    if ($\alpha\prime = 1$) then
        $C_A \leftarrow 0$
    else
        $C_A \leftarrow slack \times \alpha\prime/(1-\alpha\prime)$
    SetInterrupt($T_i, C_A/\alpha\prime$)
    SetFrequency($\alpha\prime$)

Procedure TaskPreemption($\mathbf{T_{ij}}$)
    $slack_{ij} \leftarrow c_{ij} + left_{ij} - C_i$
    $slack \leftarrow slack - slack_{ij}$
    let $s \leftarrow slack_{ij}$
    forall $T_{ab}$ idle task jobs
      in $d_{ij}..d_{pk}$ and in $r_{ij}..t$ while $s > 0$ do      (2)
        $slack \leftarrow slack - C_a$
        $s \leftarrow slack - C_a$
        reserve $C_a$ for $T_{ij}$

Procedure TaskCompletion($\mathbf{T_{ij}}$)
    if $T_{ij}$ was preempted then
        if $c_{ij} > C_i$ then (late finish)
            $slack \leftarrow slack - c_{ij} + C_i$
    else (early finish)
        $slack \leftarrow slack + C_i - c_{ij}$
    forall $T_{ab}$ idle task jobs in $r_{ij}..d_{nl}$ do
        $slack \leftarrow slack - C_a$                    (3)
    $C_{avg\_i} \leftarrow (C_{avg\_i} \times (j-1) + c_{ij} \times \alpha\prime)/j$
    $left_{i(j+1)} = C_i$

Procedure SetInterrupt($\mathbf{T_{ij}, C_A}$)
    Set timer interrupt for $T_{ij}$,
    triggered $C_A$ time units later

Procedure InterruptHandler($\mathbf{T_{ij}}$)
    if $T_{ij}$ not completed then
        $slack \leftarrow slack - (c_{ij} + left_{ij} - C_i)$
        SetFrequency(1)

Procedure SetFrequency($\alpha\prime$)
    $f \leftarrow \alpha\prime \times f_m$
```

Figure 3: Pseudocode of Feedback DVS-EDF Algorithm

At task completion, either generated slack is aggregated upon early completion or consumed slack is accounted for.

For preemption, we only need to adjust for the unscaled difference. Unused slack not eligible for consumption by the next task is subtracted (3). As always, partial credit for idle slots is omitted. We also keep track of average execution times per task.

Upon receiving an interrupt, we scale to the maximum frequency to ensure timely completion in case the WCET is exploited. Consumed slack is subtracted, including the budget for the remaining execution at the maximum frequency.

The algorithm has a constant complexity if the idle task period is chosen adequately. So far, we have argued for an idle task period equal to the shortest period of the task set. In this case, each loop in the algorithm may iterate up to $\frac{P_n}{P_1}$ times. Conversely, an idle period equal to the period of the longest task results in only one iteration. This low runtime overhead is unique, to the best of our knowledge. Other DVS scheduling schemes impose a linear overhead in the number of tasks.

The following example illustrates the benefits of aggressive DVS under EDF that we promote. Consider the task set and actual execution times in Figures 4(a) and 4(b), respectively. Figure 4(c) depicts the scaling factors determined

Task T_i	WCET C_i	Period P_i
1	3 ms	8 ms
2	3 ms	10 ms
3	1 ms	14 ms
idle	1 ms	4 ms

(a) Task Set

	Invocations	
Task T_i	c_{i1}	$c_{ij}, j > 1$
1	2 ms	1 ms
2	1 ms	1 ms
3	1 ms	1 ms
idle	0 ms	0 ms

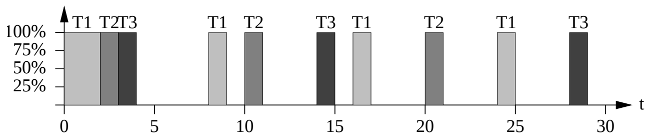
(b) Actual Execution Times

time	Task T_{ij}	α'	c_{ij}
0.00	11	0.50	4.00
4.00	21	0.50	2.00
6.00	31	0.25	6.00
10.00	12	0.75	1.33
11.33	22	0.50	2.00
14.00	32	0.25	2.00
16.00	13	0.25	4.00
20.00	32	0.25	2.00
22.00	23	0.50	2.00
24.00	14	0.25	4.00
28.00	33	0.25	2.00

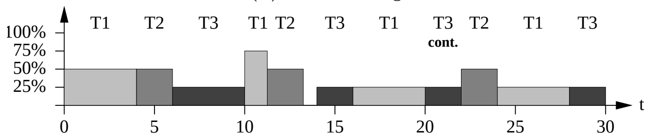
(c) Scheduling Parameters

Figure 4: Sample Task Set

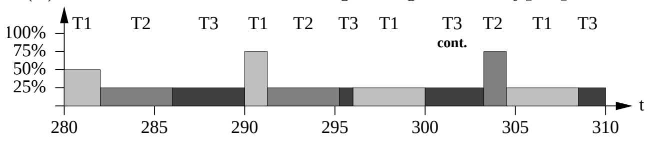
by our scheduling approach and the actual execution time with scaling. Figure 5(a) shows the time line for the execution of tasks, which shows frequent processor idle times in the absence of scaling. Figures 5(b) and (c) show the execution with scaling under our feedback EDF scheme for the beginning of the first and second hyperperiods, respectively. Notice the changes in scaling between the two hyperperiods. These changes are due to our adaptive scheme responding to the average execution times seen in the past.



(a) No Scaling



(b) Our Feedback DVS at Beginning of 1st Hyperperiod



(c) Our Feedback DVS at Beginning of 2nd Hyperperiod

Figure 5: Sample Execution

We shall demonstrate our approach for a few examples of task releases.

- At time zero, the idle task is generating 2ms of slack (first and second instance), which allows T1 to lower its frequency to 0.5 assuming an actual execution time of 1.5ms. Our implementation uses 50% of the WCET as the initial setting for the actual execution times, which was motivated in the introduction. After 4ms, a portion of the original slack (1ms) can be passed to T2. T2 receives additional slack (1ms) from the third instance of the idle task.
- At time 16, T1 preempts T3 and receives enough slack (3.5ms) to start its execution at 25%. T1 cosumes one unit of slack and passes the rest on to the preempted task. T3 then adds its future slack (1ms) reserved at preemption time, reassesses its scaling level and continues at 25%.

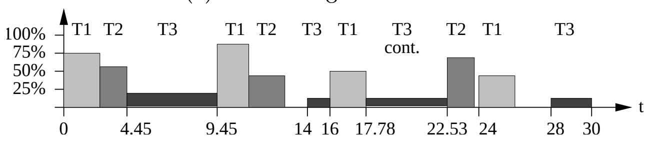
In comparison, our approach results in a schedule that matches the optimal energy consumption more closely than prior work, as will be demonstrated in comparison with Pillai and Shin's approach in the next section. For this example, we measured a 27% reduction in energy consumption, using the model detailed in the next section.

We also compared our work with Melhem's speculative dynamic DVS schemes [14], using the same task set and execution times as Figure 5(a) and 5(b).

We see from Figure 6 that, in theory, the aggressive algorithm based on speculation can reduce the processor speed of some tasks to a very low level. However, their scheme does not take discrete processor levels into account, as found in

time	Task T_{ij}	α	c_{ij}
0.00	11	0.75	2.66
2.66	21	0.56	1.78
4.45	31	0.20	5.00
9.45	12	0.77	1.30
10.75	22	0.46	2.17
14.00	32	0.12	2.00
16.00	13	0.56	1.78
17.78	32	0.16	4.75
22.53	23	0.72	1.38
24.00	14	0.45	2.22
28.00	33	0.08	2.00

(e) Scheduling Parameters



(f) Schedule Diagram

Figure 6: Sample Execution of DRA algorithm

practice. Furthermore, our algorithm results in lower power consumption on the average.

5. EXPERIMENTS

We implemented our algorithm in a simulation environment that supports EDF scheduling. In the same environment, we also implemented the look-ahead DVS algorithm, which is the best dynamic scheduling algorithm for energy conservation with EDF scheduling that we know of [32]. The results of both algorithms were verified by ensuring proper task release, activation and completion. We provide different frequency settings to simulate DFS. In addition, we calculate energy consumption based on DVS with voltage levels specified for each processor frequency. For this purpose, we assume scaling to the lowest level during idle times since it is not realistic to put a processor into sleep mode for frequent task releases, as given in periodic scheduling. We restrict ourselves here to report results based on four frequency settings and associated voltage levels, as depicted in Table 1.

frequency	voltage
25%	2 V
50%	3 V
75%	4 V
100%	5 V

Table 1: Processor Model for Scaling

We experimented with a task set of three tasks varying utilizations and changing proportions between actual and worst-case execution times. Utilizations were varied between 10% and 100% in increments on 10%. Our EDF feedback approach with variable frequencies, *Feedback-DVS*, modulates frequencies based on average execution times of the past and uses task splitting. It assumes an actual execution time that is 50% of the WCET at time 0, and the values are adjusted over time. The energy consumption is compared to Look-ahead EDF [32], an upper bound in the absence of scaling within tasks (but scaling at 25% during idle time) and a lower bound for scaling at an optimal level based on the actual utilization. The lower bound assumes ideal scaling levels instead of discrete levels supplied by a processor, and it does not take missed deadlines into account.

Figures 7, 8, 9 and 10 summarize the results for actual execution times of 25%, 50%, 75% and 100% of the WCET, respectively. Overall, our Feedback-DVS outperforms the Look-ahead-DVS of [32]. Both schemes reduce energy consumption considerably compared to the upper bound without scaling for tasks. In the absence of any scaling (fixed processor frequency and voltage, even during idle time), the energy consumption would be that of the right top corner (equivalent to 100% utilization). Our scheme achives considerable improvements over the upper bound. In fact, it closely approximates the lower bound (more closely than the look-ahead scheme).

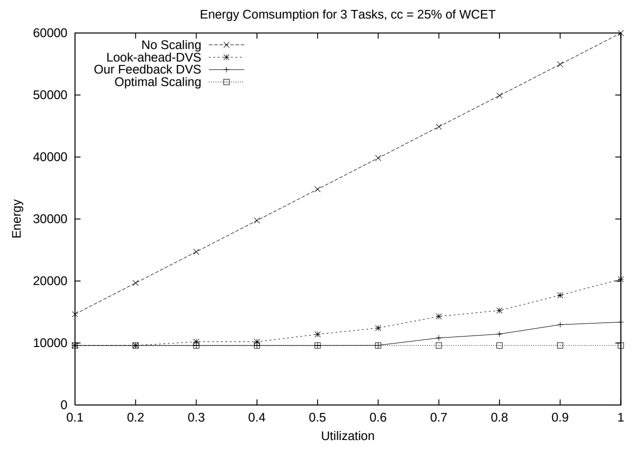


Figure 7: Energy Consumption for 25% of WCET

At 25% of the WCET, savings mostly materialize above 40% utlization for our feedback scheme. Below this point, any task can be scaled at the lowest energy level, *i.e.*, there are virtually no differences. At high utilizations, energy savings of up 34% are achieved by our feedback scheme compared to the look-ahead method.

At 50% of the WCET, we observed similar results with a regular savings of up to 22% of our feedback approach over the look-ahead scheme.

At actual executions of 75% of the WCET, up to 16% engery savings can be obtained by our feedback scheme compared to the look-ahead approach.

For 100% of the WCET, our feedback EDF technique results in slightly worse performance than the look-ahead mechanism in most cases but differences are in the order of 1-5%. This can be explained as follows. Since all of its execution budget is used by each task, speculation on early completion does not materialize any slack, only idle time can provide slack.

Figure 11 depicts the savings of our feedback EDF approach relative to the look-ahead approach and the optimal lower

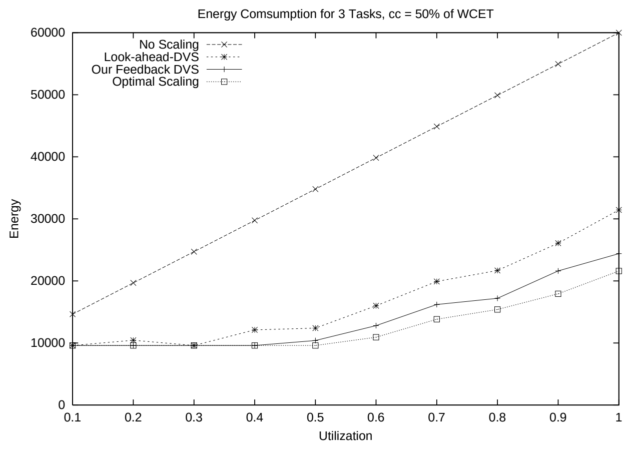


Figure 8: Energy Consumption for 50% of WCET

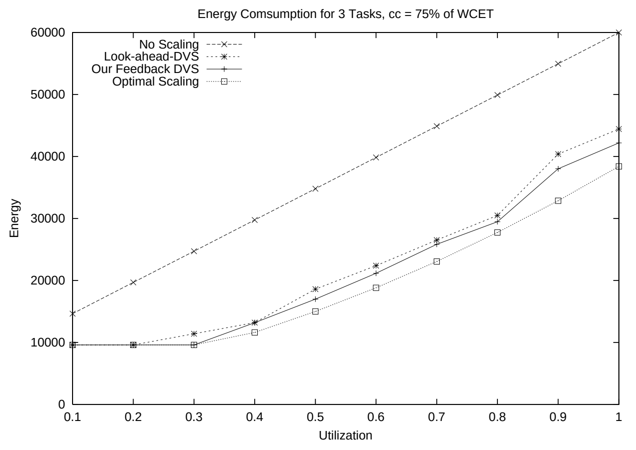


Figure 9: Energy Consumption for 75% of WCET

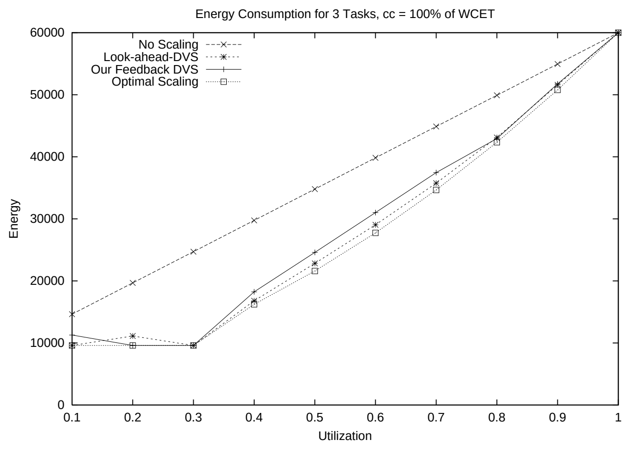


Figure 10: Energy Consumption for 100% of WCET

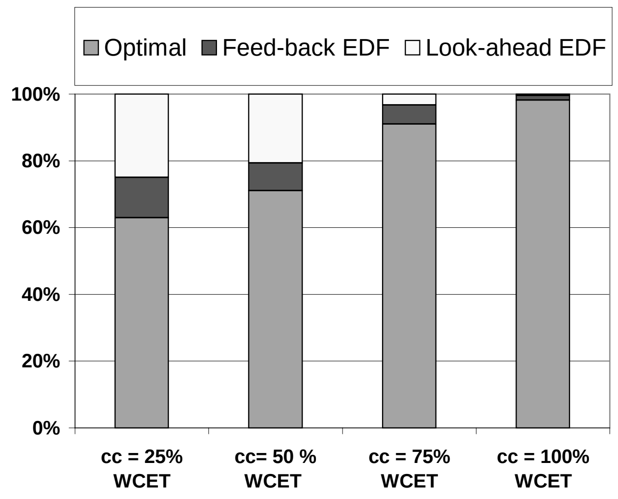


Figure 11: Energy Consumption at 80% Utlization, Normalized for Look-Ahead EDF

bound. Numbers are normalized for the look-ahead approach. This is a snapshot for 80% utilation at the four different proportionate rates between actual and worst-case execution time. Our approach more closely approximates the optimal case, particularly when execution budgets are not fully exhausted. As mentioned before, actual executions are typically well below the worst case, *i.e.*, our feedback approach caters to the more typical case.

Our Feedback-DVS algorithm has some interesting properties. During the first hyperperiod, it exhibits slightly worse energy results due to the fact that its initial tasks cannot accurately predict the worst-case execution times. After one hyperperiod, the values approach a stable point, and scaling is more accurate resulting in higher energy savings. In the graphs, we reported the results for up to ten hyperperiods. Hence, our actual results for later hyperperiods are even slightly better than shown.

6. RELATED WORK

There have been a number of efforts for incorporating DVS techniques into general-purpose computing systems. But only recently did researchers begin to apply DVS to embedded systems with timing constraints. Most of those specifically target hard real-time systems, just as in our study, where deadlines cannot be missed. In some frameworks, worse-case execution time is used to make voltage-frequency scaling decisions, such as in [7]. They designed a software tool to convert an application automatically into a low energy version solely based on remaining worse-case execution time obtained off-line. Others combine off-line and on-line information to make DVS scheduling decisions. The off-line component determines the lowest possible maximum processor speed as the initial speed setting while guaranteeing deadlines of all tasks based on WCET. The on-line component dynamically adjusts the processor speed (or brings a processor into sleep mode) according to the status of task sets, so as to exploit execution time variations and idle intervals. The on-line adjustment mechanism can be based on actual workload to reclaim unused time and energy [17, 19, 36], or it can be based on statistical information about the practical workload to anticipate and exploit early completions of future executions, as is the case in [14, 11].

Our research is more closely related to the work described in [32, 14, 11]. In [32], Pillai and Shin proposed a set of dynamic DVS algorithms based on traditional hard real-time mechanisms, namely rate-monotone (RM) scheduling and EDF. It extends the schedulability test of RM and EDF algorithms to incorporate CPU frequency scaling. Unlike our algorithm that applies frequency scaling to only the current task, they assume the same frequency scaling factor upon all tasks in a real-time task set. In their most aggressive variant, a look-ahead technique is used to achieve extensive energy savings by deferring as much work as possible. However, the frequency value obtained in their algorithm is not the lowest possible frequency for a single task, as comparisons with the optimal case and with our work show.

Another aggressive real-time DVS scheme is presented by Melhem et al.[14], which speculates on and exploits early completions of future task executions on the fly based on statistical information about the workload. We used an even more aggressive scheme to calculate future task scaling levels based on their past execution profile. Our work not only incurs lower overhead during scheduling but also produces lower energy levels, as demonstrated in Section 4.

The idea of deriving a feasible dual-level voltage schedule from an ideal case is first proposed by F. Gruian [11, 12]. It combines off-line and on-line scheduling, both at task level and task-set level. Stochastic data is used to derive energy-efficient schedules. We use a different method that splits a task into two parts and always assigns the highest frequency to the second part. Our algorithm focuses on dynamic scheduling (EDF) while [11] restricts the focus to fixed-priority schemes.

Slack stealing has been used in dynamic and static scheduling schemes [6, 21, 8]. The purpose of slack stealing was to capitalize on slack for the sake of executing aperiodic jobs. In contrast, our approach passes slack to the periodic tasks of a task set.

7. FUTURE WORK

We are working on extensions with stochastic scaling levels, which improves energy consumption for tasks with a high variance of its actual execution time. We are also implementing the algorithm on an embedded architecture with support for DVS and DFS to obtain real-world measurements.

8. CONCLUSION

We have developed a new approach for DFS/DVS that extends EDF in a highly efficient manner. Our technique relies strictly on operating system support within the scheduler to implement the approach. Early scaling at a low frequency, determined by a feedback mechanism and facilitated by a slack-passing scheme, capitalizes on high probabilities of a task to finish its execution without utilizing its worst-case execution budget. If a task does not complete at a certain point in time within its low frequency range, the remainder of it continues to execute at a higher frequency. Our experiments demonstrate that the resulting energy savings exceed those of previously published work by to 34%. In addition, our method adds only a constant complexity at each scheduling point, which has not been achieved by prior work, to the best of our knowledge.

9. REFERENCES

[1] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[2] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *J. of Real-Time Systems*, 8:173–198, 1995.

[3] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[4] T.P. Baker and Alan Shaw. The cyclic executive model and Ada. Technical Report 88-04-07, University of Washington, Department of Computer Science, Seattle, Washington, 1988.

[5] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[6] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[7] J. Kim D. Shin and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design and Test of Computers*, March 2001.

[8] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In Susan Davidson and Insup Lee, editors, *Proceedings of the Real-Time Systems Symposium*, pages 222–231, Raleigh-Durham, NC, December 1993. IEEE Computer Society Press.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techiniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 37–46, June 1997.

[10] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *1st Int'l Conference on Mobile Computing and Networking*, Nov 1995.

[11] F. Gruian. Hard real-time scheduling for low energy using stochastic data and dvs processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, Aug 2001.

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

[13] D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. In *Symp. on Operating Systems Design and Implementation*, Oct 2000.

[14] D. Mosse H. Aydin, R. Melhem and P.M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of 22nd Real-Time Systems Symposium*, December 2001.

[15] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*, pages 68–77, December 1992.

[16] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[17] I. Hong, M. Potkonjak, and M. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Int'l Conference on Computer-Aided Design*, Nov 1998.

[18] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *19th Real-Time Systems Symposium*, Dec 1998.

[19] C. Krishna and Y. Lee. Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems. In *6th Real-Time Technology and Applications Symposium*, May 2000.

[20] Y. Lee and C. Krishna. Voltage clock scaling for low energy consumption in real-time embedded systems. In *6th Int'l Conf. on Real-Time Computing Systems and Applications*, Dec 1999.

[21] John P. Lehoczky and Sandra Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1992*, pages 110–124, Phoenix, Arizona, USA, December 1992. IEEE Computer Society Press.

[22] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.

[23] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[24] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[25] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[26] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[27] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

[28] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power*, October 2000.

[29] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[30] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[31] T. Pering, T. Burd, and R. Brodersen. The simulation of dynamic voltage scaling algorithms. In *Symp. on Low Power Electronics*, 1995.

[32] Padmanabhan Pillai and Kang G. Shin. Real-Time dynamic voltage scaling for Low-Power embedded operating systems. In Greg Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 89–102, New York, October 21–24 2001. ACM Press.

[33] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor, 2000.

[34] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[35] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, September 1990.

[36] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Int'l Conf. on Computer-Aided Design*, 2000.

[37] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer, 1998.

[38] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[39] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *1st Symp. on Operating Systems Design and Implementation*, Nov 1994.

[40] W. Wolf. Smart cameras and embedded computing. seminar presentation, January 2002.

[41] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, October 1993.