DVSleak: Combining Leakage Reduction and Voltage Scaling in Feedback EDF Scheduling *

Yifan Zhu Frank Mueller

Department of Computer Science/Center for Embedded Systems Research
North Carolina State University, Raleigh, NC 27695-7534
mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7925

Abstract

Recent trends in CMOS fabrication have the demand to conserve power of processors. While dynamic voltage scaling (DVS) is effective in reducing dynamic power, microprocessors produced in ever smaller fabrication processes are increasingly dominated by static power. For such processors, voltage/frequency pairs below a critical speed result in higher energy per cycle than entering a processor sleep mode. Yet, computational demand above this critical speed is best met by DVS techniques while still conserving power.

We develop a novel combined leakage and DVS scheduling algorithm for real-time systems, DVSleak, based on earliest-deadline-first scheduling (EDF). Our method trades off DVS with leakage, where the former slows down execution while the latter intelligently defers dispatching of jobs when sleeping is beneficial. We further capitalize on feedback knowledge about actual execution times to anticipate computational demands without sacrificing deadline guarantees. As such, we contribute a novel feedback delay policy for leakage awareness, which addresses structural limitations of prior approaches. Experiments show that this combined DVS/leakage algorithm results in an average of (a) 50% additional energy savings over a leakage-oblivious DVS algorithm, (b) 20% more energy savings over a more simplistic combination of DVS and sleep policies and (c) 8.5% or more over dynamic slack reclamation with procrastination. Particularly task sets with periods shorter than ten milliseconds profit from our approach with 15% energy savings over best prior schemes. This makes DVSleak the best combined DVS/leakage regulation approach for real-time systems that we know of.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—scheduling; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems

General Terms Algorithms, Experimentation

Keywords Real-Time Systems, Scheduling, Dynamic Voltage Scaling, Leakage, Feedback Control

* This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

LCTES'07 June 13–16, 2007, San Diego, California, USA.

1. Introduction

Power consumption in a CMOS-based processor consists of three elements: dynamic, static, and short-circuit power [16]. Most of the previous work on dynamic voltage scaling (DVS) only considers dynamic power consumption of a CMOS circuit while ignoring the static portion [23, 19, 4, 13, 7, 8, 18, 22, 21]. Static power consumption stems from leakage current that exists even in the absence of logic operations of a circuit. While both static and dynamic power increase (at different rates) with the supply voltage and, hence, processor frequencies, a better metric for the effectiveness of a frequency/voltage setting is *energy per cycles*, which relates the amount of work to energy. Jejurikar *et al.* found that as the supply voltage is reduced below a certain threshold value, static energy per cycle exceeds the dynamic energy per cycle, *i.e.*, it becomes the dominant cause of power consumption *per se* (see Figure 2 of [12]). The processor frequency associated with this threshold voltage is called the *critical speed*. Above the threshold voltage (and associated frequency), the total energy per cycle increases as the processor voltage scales up and can operate at a higher frequency to get more work done (instructions executed). But below the threshold voltage, the total energy per cycle *also increases* as the voltage scales down. This is because lower frequencies imply lower performance, *i.e.*, increased execution time for the same amount of work during which leakage power is consumed. Instead of prolonged execution below the critical speed, the processor should be suspended in a shallow sleep mode during which only negligible power is consumed, typically three orders of a magnitude less than during any execution mode. This result leads us to re-consider two issues when combining DVS and leakage awareness:

1. It is not energy-efficient to scale down processor voltage and frequency to an extremely low level if that level is below the threshold value and it is beneficial to enter a sleep mode instead.
2. Due to leakage power, forcing the processor into sleep mode may be more energy-efficient than keeping the system idle at a low frequency as long as the idle period is long enough to compensate for the shutdown overhead.

Capitalizing on these observations, we devise a novel feedback delay policy that, in contrast to structural limitations of any prior work, makes online decisions about when to delay a task release based on actual execution times without jeopardizing deadlines.

Of course, any combined DVS/leakage policy has to take into account the above items and should make decisions according to the actual power consumption characteristics. These issues have been addressed in previous work where both static and dynamic power consumption are reduced [10, 15, 12]. These approaches either assume that all tasks are running at the same speed (to conserve static power) or they use off-line schemes without fully exploiting the power saving potentials. Lee *et al.* further use a greedy method to locally maximize the duration of alternating idle and busy periods based on the worst-case execution time [14]. Jejurikar *et al.* use static slack combined with deferring/delaying execution (procrastination) [12]. Since actual execution times often diverge con-

siderably from the WCET, a conceptually busy period is actually interspersed with dynamic slack due to early completion of jobs. The potential of dynamic slack remains unused. An extension of the latter work also exploits dynamic slack for suspension [11]. They promote a policy similar to our trade-off between scaling and delaying for suspension in Section 4. However, they do not provide a look-ahead policy for deciding whether to delay early or late, such as our novel feedback delay policy detailed in Section 5. They report between 0.3-16% energy savings over their static procrastination algorithm with larger savings for higher utilizations and actual execution times up to a tenth of their WCET. In contrast, we uniformly outperform their most advanced dynamic slack reclamation scheme for varying utilizations, actual to WCET ratios and execution patterns with an average of 8.5% additional savings. For shorter periods, these savings increase to 15%. This shows that our policy of *delay now or later is crucial* in achieving additional energy savings. Furthermore, the work by Jejurikar *et al.* assumes that a power manager, implemented as a controller in hardware, handles interrupts and sets timers when new tasks are released. In contrast, our scheme does not require any special hardware support beyond DVS and sleep modes, nor does it assume execution times equal to their worst-case bounds.

In this paper, we present an on-line combined DVS/leakage control scheme that minimizes both static and dynamic power consumption. This scheme profits from our feedback-DVS algorithm that exploits a modified earliest-deadline-first (EDF) scheduling variant. It automatically chooses between voltage scaling and a processor sleep mode according to the run-time execution scenario of tasks. Voltage scaling is used when dynamic power dominates the total power consumption. Conversely, a processor sleep mode is entered when static power dominates the total power consumption. Our scheme also locally adjusts the dispatch time of a task so that adjacent tasks are either clustered together or scattered apart to increase the opportunity of entering the sleep mode. It incorporates preemption handling and models wakeup overhead. We also show that the overhead in time for suspend/wakeup is 225 and 80 μs, respectively, with actual measurements on a test platform. Hence, sleep mode utilization in real-time systems is realistic for contemporary embedded architecture.

The rest of the paper is organized as follows. Section 2 introduces the system model. Section 3 presents the motivation for combined leakage reduction and DVS. Section 4 discusses the relationship between speed reduction and task delaying. Section 5 details the delay policy of our algorithm. Section 6 shows experimental results based on simulation. Section 7 discusses related work, and Section 8 summaries the paper.

2. System Model

This paper targets hard real-time systems with a periodic, preemptive and independent task model. There are n periodic tasks in the system. Each task T_i in the task set is defined by a triple (P_i, D_i, C_i), where P_i, D_i and C_i are the period, relative deadline, and worst-case execution time (WCET) of T_i, respectively. While a task can execute at different processor frequencies, C_i always refers to the execution time measured at the maximal processor frequency. We also assume that D_i=P_i in our model, which is the most common case in real-time systems. The periodically released instances of a task are called jobs. T_{ij} is used to denote the j^{th} job of task T_i. Its release time is $P_i * (j-1)$ and its deadline is $P_i * j$. We use c_{ij} to represent the actual execution time of job T_{ij}. The hyperperiod H of the task set is the least common multiple (LCM) among the tasks' periods.

To assess power consumption, we employ the power model of a CMOS circuit due to Martin *et al.* [16]. The power consumed in a processor consists of three portions: dynamic power P_{AC}, static power P_{DC}, and short-circuit power. Short-circuit power is only consumed during signal transitions and, in practice, is generally negligible [16]. Hence, we consider static and dynamic power in our model. Similar power models are also used in related work [12, 20]. Dynamic power is given by:

$$P_{AC} = C_{eff} V_{dd}^2 f \tag{1}$$

where C_{eff} is the average switched capacitance per cycle, V_{dd} is the supply voltage, and f is the processor clock frequency. Static power is given by:

$$P_{DC} = V_{dd} I_{subn} + |V_{bs}|(I_{jn} + I_{bn}) \tag{2}$$

where I_{subn} is the sub-threshold leakage current, V_{bs} is the body bias voltage, and I_{jn} and I_{bn} are the drain and source to body junction leakage currents, respectively.

A DVS-enabled processor is capable of operating at multiple frequency and voltage levels. Reducing the voltage also reduces the highest stable frequency supported by the system. Since the processor frequency determines the speed of the system, we use these two terms interchangeably in this paper. Static and dynamic power can be traded off against each other in practice. It has been shown that there exists a threshold voltage V_{th} below which execution is no longer energy efficient, *i.e.* the voltage should not be scaled below this threshold value [12]. From the threshold voltage V_{th}, one can derive a corresponding threshold frequency f_{th}, the *critical speed*. Instead of operating at a speed below the threshold value, it is more power efficient to execute tasks at or above the critical speed. The reason for this observation is that the leakage power, which dominates below the critical speed, is incurred for a longer period of time as frequency is scaled below the critical speed. Thus, at extremely low speeds, longer execution results in higher overall energy per cycle than shorter execution at a higher frequency followed by a sleep interval since there is virtually no leakage power consumed during sleeping.

Sleep modes are widely supported by contemporary processors as a special state. However, the transition into and out of a sleep mode does not come without cost. Such a transition incurs additional power consumption, termed sleep overhead from here on. This overhead is mostly due to warm-up of resources (particularly caches) when resuming execution. Hence, sleeping is only a viable option when the energy saved by sleeping exceeds that of the sleep overhead itself.

In the following, we assume a deep sleep mode during which only the interrupt line of a processor remains receptive. Other parts of the processor, including caches, are turned off and will lose their state. In our model, we assume that the processor consumes a negligible amount of energy when in sleep mode. Power consumption in the sleep mode is documented as being three orders of a magnitude lower than the power consumption in active mode [9]. Transitioning into and out of a sleep mode incurs, as a side-effect, cold misses in cache among other resource refresh overheads. Let E_{sd} be the additional energy per wakeup. The overhead of entering and exiting the sleep mode is also included in the sleep threshold derived below.

Let p_{idle} be the power consumption when the system is idle plus the overhead due to cold misses upon wakeup. Then, $t_{th} = E_{sd}/p_{idle}$ defines a sleep threshold. It is energy efficient to enter sleep mode if and only if the slack time in the schedule exceeds t_{th}. Otherwise, the processor should remain idle at a power-efficient DVS level. These parameters are platform dependent but are available to the operating system scheduler at system initialization.

In the following, we describe the DVSleak algorithm, which is integrated into the task scheduler and contains policies for reducing both static and dynamic power.

3. Motivation

Our leakage-aware DVS algorithm is based on a feedback-DVS algorithm adopted from Zhu *at al.* [25, 26]. Their approach handles of preemptions, simultaneous task releases and provides a proof for correctness in terms of never missing deadlines. We provide a brief description of this algorithm in the following. Instead of executing each task at a uniform speed, a task's worst-case execution time is divided into two sub-tasks, T_A and T_B, as shown in Figure 1. Their

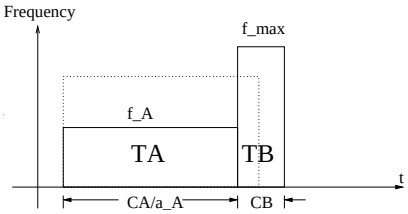


Figure 1. Task Splitting

worst-case execution time under the maximal frequency is C_A and C_B, respectively. These two subtasks are allowed to execute at different frequency and voltage levels. T_B always executes at the maximum frequency level f_{max}, which allows T_A to execute at a lower frequency level f_A than it could without task splitting. Based on this frequency,

$$\alpha_A = f_A / f_{max} \tag{3}$$

is the so-called *scaling factor* by which execution speed is reduced. By splitting each task into at most two subtasks, at most one speed change is incurred for every task. This keeps the impact of voltage and frequency switching overhead to a minimum.

A task is split in such a way that its anticipated execution can complete within the T_A portion. If its execution exceeds the anticipated value, there is still enough time reserved in T_B to meet its deadline (see their paper [26] for a proof). With this scheme, the frequency of T_A can be safely scaled exploiting the available slack while T_B executes at the maximum frequency following a last-chance approach [6]. In addition, feedback is used to collect each task's previous execution history for the scheduler to assist in making scheduling decisions for future tasks. A task's expected actual execution time is used to determine the length of the T_A subtask of the next instance. In the following, a task's expected actual execution time is also used in our delay policy to decide when to delay the task's release time.

To make the DVS algorithm leakage aware, our feedback-DVS scheme takes into account the impact of dynamic power as well as the threshold voltage to consider the effect of static power. A naïve scheme is to mark all voltage and frequency levels below the threshold as invalid, so that whenever the DVS algorithm wants to assign a speed below that threshold, it uses the threshold value instead. A task then runs at a higher speed than its original assignment. It completes earlier providing more dynamic slack prior to its deadline. As long as the slack is long enough to compensate for the shutdown overhead, the DVS scheduler can put the processor into a sleep mode during that interval to further reduce the impact of static power consumption.

Unfortunately, such a naïve scheme does not fully exploit the energy saving potential. Consider the three tasks depicted in Figure 2(a). Task T_1 completes at time t_1. Task T_2 is released at time t_2 and completes at time t_3. Task T_3 is released at time t_4. Let the lengths of both idle intervals $[t_1, t_2]$ and $[t_3, t_4]$ be less than the threshold t_{th}. Hence, the processor is kept in an idle state during the above intervals instead of entering a sleep mode. Both static and dynamic power consumption exist in the idle state. The energy cost for an idle state, although lower than the energy for a non-idle running state, is still significantly higher than the energy for a sleep duration of the same length of time.

Figure 2. Combining DVS and Leakage Savings

To further exploit the savings for both static and dynamic power, we adapt the schedule of the system to reduce static leakage as much as possible. Consider shifting task T_2 to line up with the release time of T_3 as depicted in Figure 2(b). T_2 is now activated at time $t2'$. The interval $[t_1, t_2']$ then exceeds the sleep threshold value t_{th} so that the processor enters a sleep state during that interval. Static power is almost eliminated while sleeping. The only consumption the processor pays is dynamic power as well as the wakeup overhead. Figure 2(b) is the ideal case where T_2 completes exactly before the release of T_3, thus maximizing the processor sleep period. Even if T_2 takes less cycles than expected and completes earlier, delaying the activation time of T_2 costs less energy than the non-delay policy. As shown in Figure 2(c), if T_2 completes earlier, the processor enters the idle state till the release time of T_3. The energy saved in $[t_1, t_2']$ due to sleeping makes the delay policy superior to the non-delay schedule, as shown in Figure 2(a).

The above example illustrates the benefit of the delay policy in terms of reduced leakage in a DVS-aware system. In the following, we present an algorithm that combines this delay policy with dynamic slack reclamation and feedback of actual execution times.

4. Speed Reduction vs. Task Delaying

DVS technology modulates the processor speed according to the amount of slack in the schedule. The existence of slack is due (1) either to the under-utilization of system workload, which can be determined statically, (2) or to the early completion of tasks, which is determined dynamically. A dynamic voltage scaling algorithm, when integrated with leakage saving schemes, needs to address two issues. First, it needs to determine how to distribute the amount of slack between speed reduction and task delaying. Second, it needs to decide if the release time of a job should be delayed. This section focuses on the first challenge while the next section addresses the second one.

Consider the example in Figure 3(a). Lowering the processor voltage and frequency, *i.e.*, reducing the application speed, de-

creases the amount of slack available in the schedule, as depicted in Figure 3(b). Similarly, delaying the activation time of a task by

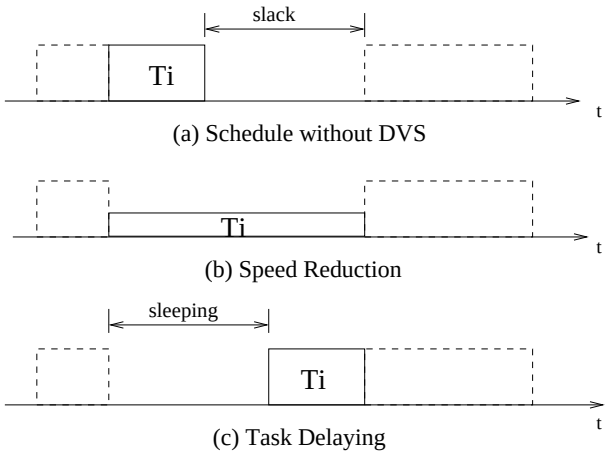


Figure 3. Speed Reduction vs. Task Delaying

putting the processor into a sleep mode also decreases the amount of slack, as depicted in Figure 3(c). During execution, the amount of slack is always a shared resource between these two competing operations. The DVS algorithm has to define a policy to distribute the slack between these two schemes. This dilemma can be solved based on the critical speed (frequency). We prefer a lower frequency over task-delaying as long as the resulting frequency is higher than the critical speed, *i.e.*, if such a choice results in lower energy. Conversely, when our frequency scaling scheme suggests a speed lower than the critical speed, we default to the critical speed and activate the delay policy. This scheme reflects a best effort to reduce power. According to the above analysis, whenever a task completes and a new task T_i is released, the DVS algorithm uses a feedback-EDF scheme (adopted from [25]) to calculate a frequency level f_i. The actual frequency f assigned to task T_i is defined by:

$$f = min(f_i, f_{th}) \tag{4}$$

Given the actual frequency of task T_i, a corresponding voltage can be derived. But before task T_i is released, the DVS scheduler has to decide whether or not the release time of the task needs to be delayed. This issue is detailed in the following section.

5. Feedback Delay Policy

The example in Section 3 seems to imply that a task should always be delayed (procrastinated) as much as possible against its deadline. This is also the strategy used in previous work [12, 20, 11]. Such an intuitive approach, however, is not always the best solution. This is due to the variability of the actual execution time of tasks. Figure 2(b) shows the case where the execution time of task T_2 equals its worst-case execution time. In reality, the actual execution time of a task is generally shorter than its worst-case execution time.

A schedule without delay of T_i's release time leaves the processor idle in the beginning. Some time later, the processor enters a sleep mode, as shown in Figure 4(a). Figure 4(b) depicts the effect of a delayed schedule, where the processor enters the sleep mode first and later on incurs a potentially longer idle period, thereby consuming more power than in case (a). This effect is due to the delay policy, which relies on the WCET instead of the actual execution time of T_i to determine the delay. When a task completes earlier than expected, it produces additional dynamic slack, which significantly reduces the benefit of the delay policy.

Taking this short-coming into consideration, we present the following delay policy as part of our DVSleak algorithm. We observe

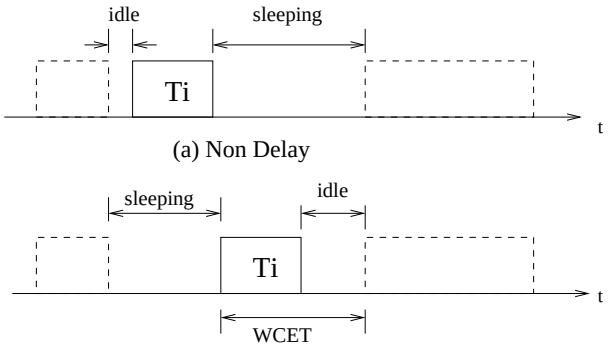


Figure 4. Delay vs. Non-delay

that at any time t, the DVS algorithm can infer the amount of slack s_t in the schedule. If the ready queue of the scheduler is not empty, the delay policy remains inactive. As shown in Figure 5, the next task T_i will be released at time t_r ($t_r \geq t$) according to the EDF scheduling. With the knowledge of s_t, the feedback-DVS algorithm assigns a processor frequency f_A and a scaling factor α_A, as defined in Equation 3, for T_A, which is the first subtask of T_i in the task splitting scheme. Since the number of execution cycles of T_i is also split into two parts, we have

$$\frac{C_A}{\alpha_A} + C_B = \frac{C_i}{\alpha_i} \tag{5}$$

where α_i is a unified scaling factor of the entire task (as if the task had not been split). By introducing α_i, the delay policy of the following task can be easily integrated into any DVS algorithm. From Equation 5, we derive α_i:

$$\alpha_i = \frac{C_i \alpha_A}{C_A + C_B \alpha_A} \tag{6}$$

Let c_i be the expected actual execution time of T_i provided by a feedback scheme based on the execution times of previous instances of task T_i. The latest time that T_i can complete without missing its deadline is given by:

$$t_d = t + s_t + C_i \tag{7}$$

where C_i is the WCET of T_i. Time t_d can also be represented as the minimum of the absolute deadline of the task and the release time of the next task in the EDF schedule after T_i, *i.e.*,

$$t_d = min(d_i, t_{r_{i+1}}) \tag{8}$$

Notice that if the next task is released together with T_i, there will only be one idle period prior to T_i.

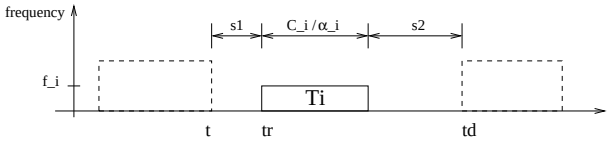


Figure 5. Rules for Task Delaying

We use the following rules, based on feedback of actual execution time, c_i, to determine the modified release time of T_i.

1. Task T_i is released at time t_r (as under standard EDF) if and only if
   (a) $t_d - t - C_i/\alpha_i \leq t_{th}$, or,
   (b) $t_d - t - C_i/\alpha_i > t_{th}$ and $t_r - t < t_{th}$ and $C_i/\alpha_i - c_i \geq t_r - t$.

2. Task T_i is released at time $t_d - C_i/\alpha_i$ (later than under standard EDF) if and only if
   (a) $t_d - t - C_i/\alpha_i > t_{th}$ and $t_r - t \geq t_{th}$, or,
   (b) $t_d - t - C_i/\alpha_i > t_{th}$ and $t_r - t < t_{th}$ and $C_i/\alpha_i - c_i < t_r - t$.

Rule 1 covers the cases where the release time of task T_i is not delayed. Conversely, Rule 2 captures the cases where it should be delayed. Rule 1(a) applies when the total amount of slack time in $[t, t_d]$ (equivalent to $s_1 + s_2$ in Figure 5) is less than the sleep threshold t_{th}. Task T_i is not delayed since there is not enough slack to benefit from sleeping, regardless of whether or not the task is delayed. Rule 2(a) applies when the total amount of slack is greater than the sleep threshold t_{th} and the initial slack s_1 is at least as large as this threshold, which ensures that sleeping will be beneficial. By delaying T_i's release time to $t_d - C_i/\alpha_i$, we increase the amount of slack prior to T's execution as much as possible to prolong the initial sleep duration.

Rule 1(b) and Rule 2(b) capture cases where the length of the first slack s_1 is less than the threshold t_{th} while the overall slack $s_1 + s_2$ exceeds this threshold. In these cases, delaying the release time of task T_i does not always result in the longest sleep duration. Figures 4(a) and (b) illustrate the best efforts reflected by Rules 1(b) and 2(b), respectively. The decision is, in fact, based on the anticipated portion of unused execution time (WCET - actual execution time). If this portion is equal or larger than slack s_1, it is beneficial to accumulate more slack (due to early completion within $C_i/\alpha_i - c_i$) with s_2, which does not require the task to be delayed, as reflected in Rule 1(b). Conversely, if the unused portion is less than slack s_1, late slack (s_2) is merged with early slack (s_1) by shifting the execution of T to the latest possible point in time, which lengthens the beneficial sleep duration prior to the shifted task, as reflected in Rule 2(b). This heuristic approach is relatively simple but yields promising results, as will be shown. Notice that c_i, the expected actual execution time of task T_i, is provided by the feedback controller according to previous execution history (based on related work, see [25, 26] for details).

We combine this task delay policy with the existing DVS algorithm. By enhancing the algorithm with the delay policy, we still guarantee the feasibility of the schedule for the task set, as stated by the following theorem.

THEOREM 1. *If a feasible schedule exists for a task set under EDF scheduling, the modified schedule after applying the delay Rules 1 and 2 is guaranteed to be feasible as well.*

PROOF. For any task T_i in the task set, let d_i be its absolute deadline. If T meets its deadline under EDF, then its release time t_r satisfies:

$$t_r + C_i + s_t \leq d_i \tag{9}$$

According to the above relationship and Equation 7, we know that $t_d \leq d_i$. Delay Rules 1 and 2 either release T_i at its original EDF time t_r or at time $t'_r = t_d - C_i/\alpha_i$. In the former case, T_i will not miss its deadline since T_i is scheduled as in conventional EDF. In the later case, let T_i's worst-case execution time after frequency scaling be C'_i. Then, $C'_i = C_i/\alpha_i$. In the worst case, T_i will complete before

$$t'_r + C'_i = t_d - C_i/f_t + C_i/\alpha_i = t_d \leq d_i \tag{10}$$

since it will be activated at its new release time t'_r and no other tasks are ready in $[t, t_d]$ due to Equation 8. Hence, T_i again completes before its deadline and the task set can still be feasibly scheduled. □

6. Execution at Sub-Critical Speed

The critical speed result was originally derived in a pure DVS environment [12]. When considering the option to enter a sleep mode to reduce leakage, the above argument that one should never scale down below the critical speed no longer holds. The reason for this lies in the overhead for sleeping, which requires one to decide on whether to sleep or not. If it is not beneficial to sleep (due to the energy cost when entering/exiting the sleep mode), the processor will for one part execute jobs and for the other part be idle. Hence, the lowest possible energy results when scaling at the lowest possible frequency to still meet deadlines, even below the critical speed, since the processor will be awake in any case.

We incorporate this technique in DVSleak by allowing jobs to scale below f_{th} if and only if a sleep mode is not entered for a given period of time (and only for this period). Thus, Equation 4 is constrained to *only* be applicable prior or after sleeping, as depicted in Figure 5 for T_i's execution and the sleep duration but *not* for any idle intervals. Instead, slack due to idle can be greedily exploited for DVS by jobs prior and after T_i. For those jobs, the original feedback DVS technique based on task splitting with Equation 3 applies at *any* frequency.

7. Framework

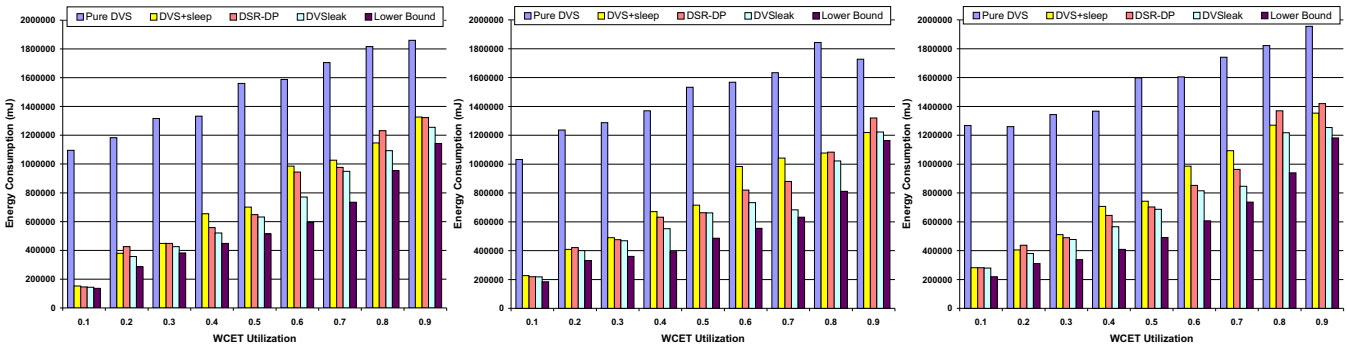
Our experimental framework is based on extensions of a simulator [26] that has proved to reliably model energy at the granularity of tasks. In addition, we assessed the overhead of dynamic sleeping for suspend and wakeup operations on a IBM PowerPC 405LP test board [17]. This 32-bit processor contains split caches and a TLB, all of which are volatile during suspension while the RAM remains refreshed in this so-called suspend mode. We used the Programmable Interrupt Timer (PIT) to assess the sleep overhead and verified these results using a wall-clock based skewing technique. In both experiments, we observed overheads of suspend/wakeup operations as 225 and 80 μs, respectively, with variations of $\pm 5\mu$s. This shows that sleep modes can be quickly entered, and execution can swiftly resume on contemporary microprocessors if properly designed for power awareness. These results and related work [26] show that such small overheads can be safely ignored in simulation frameworks and that Equation 1 is sufficient to accurately compare policies for task-level power consumption of real-time systems.

We subsequently implemented our leakage-aware DVSleak algorithm in a simulation environment obtained from [26] using the power model described in Section 2. We assume the processor has four discrete frequency levels, which are 25%, 50%, 75% and 100% of f_m. f_m is the maximal frequency supported by the processor. The corresponding power consumption at these four frequency levels are $550mW$, $650mW$, $990mW$ and $1480mW$, which are calculated using the approach described elsewhere [12]. The processor enters an idle state when no ready tasks are available.

From this, we derive the shutdown energy overhead, E_{sd}, as $483\mu J$. The idle power, p_{idle}, is $240mW$, and the sleep threshold, t_{th}, is $2ms$ for a deep sleep mode that looses cache content and only keeps a processor's interrupt line active. The critical speed, *i.e.*, the threshold frequency, f_{th}, is 41% of f_m, which means that jobs execute at the next-higher frequency level of 50% in the presence of sleeping while a level of 25% remains valid in intervals without sleeping (see Section 6). These overhead settings are consistent with prior work, which facilitates a direct comparison [12, 11, 20].

In order to assess the energy saving potential of our combined leakage-aware DVSleak algorithm, the following four algorithms are implemented in the simulator.

Pure DVS: A pure feedback-DVS algorithm with preemption handling but without any leakage power saving technologies (from [25]). The algorithm does not observe trade-offs due to the threshold frequency, *i.e.*, the processor frequency can be scaled below this threshold.



(a) c=25%WCET (b) c=50%WCET (c) c=75%WCET

Figure 6. 3 Tasks, Pattern 1, under Different Actual Execution Times (Constant) and Utilization

DVS+sleep: The feedback-DVS algorithm with a sleep policy. This algorithm puts the processor into sleep mode whenever an idle period is longer than the sleep threshold. The algorithm disables any frequency levels lower than the threshold frequency (41% of f_m). Hence, 25% of f_m is never used. However, this algorithm does not implement a delay policy to further postpone the release of a task.

DSR-DP: The dynamic procrastination algorithm with dynamic slack reclamation (DSR-DP). This is an algorithm proposed in previous work [11], which also considers leakage energy in the processor model. Their algorithm reclaims slack from both dynamic slowdown and dynamic procrastination.

DVSleak: Our algorithm presented in this paper. This is the hybrid algorithm based on feedback-DVS with (a) DVS/sleep policy (similar to [11]) and (b) a "delay now or later" policy described in the last sections. This algorithm is the most aggressive one. Besides putting the processor into sleep mode, it also delays the release time of tasks according to our delay rules. The delay rules increase the length of the sleep durations, which saves more energy than other algorithms. DVSleak also exploits knowledge about the threshold frequency.

A suite of task sets with synthetic CPU workloads was used in the experiment. Depending on the experiments, each task set contains three or ten independent periodic tasks whose worst-case execution time (WCET) is in the range of 1ms to 100ms. The actual execution time of a task follows four different patterns. Energy is used as a metric for comparison as is valid in real-time systems since we measure the power consumption over the same amount of time (hyperperiod) and require the same amount of work to be performed for a given utilization level and task set.

1. In pattern one, a task's actual execution time remains constant between different jobs.
2. In the second pattern, the actual execution time of a job starts at 50% of the task's WCET before spiking to a peak value c_m every 10th job and then receding with a decay of $c_i = 1/2^{(t-c_m)}$ again. This pattern simulates event-triggered activities that result in sudden, yet short-term computational demands due to complex inputs often observed in interrupt-driven systems.
3. The third pattern differs from the second in its more gradual decay function $c_i = c_m sin(t + \pi/2)$. This pattern simulates events resulting in computational demands in a phase of subsequent complex inputs (with a decaying tendency).
4. In the fourth execution pattern, the actual execution time of the jobs alternates between two random extremes every 10 jobs with an average execution time of $c_i = \pm c_m sin(t)$. This pattern represents periodically fluctuating activities with gradually increasing and decreasing computational needs around extremes.

For each execution pattern, the task sets' WCETs were uniformly distributed in the range of 1ms to 100ms. Each task's period was chosen so that the worst case utilization of the task set varies from 0.1 to 1.0 in increments of 0.1.

In order to assess the performance of our algorithm, we also calculate a lower bound on energy for each utilization case. We construct an ideal schedule where each task runs at either the ideal optimal speed or the critical speed, whichever the greater. Such a schedule is used to approximate the minimum lower bound on energy. Deadline misses are not considered in this schedule. The number of times the processor enters a sleep or idle state is also minimized by dividing the hyperperiod with the longest task period in the task set. This ratio multiplied with the energy cost for wake-ups, E_{sd}, comprises the overhead considered for the lower bound.

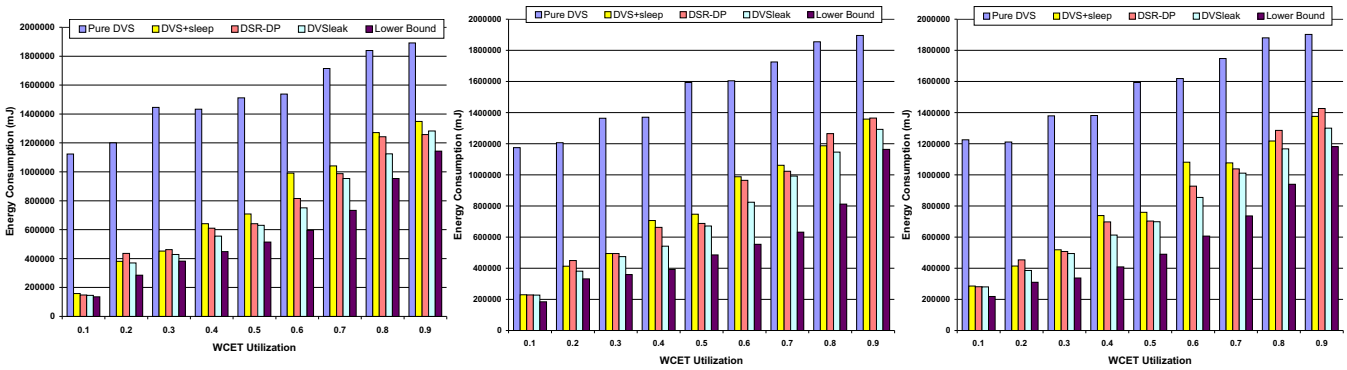
8. Experiments

We conducted a set of experiments with task sets of three and ten tasks assessing all of the above DVS policies under four different patterns. We further assessed the affect of harmonic *vs.* non-harmonic periods and short *vs.* long periods.

8.1 Energy for Set of Three Tasks

Figure 6 depicts the energy for three tasks of the different algorithms with execution pattern one, as well as the lower bound energy for each utilization case. Putting the processor into sleep mode saves as much as 80% more energy than the pure DVS algorithm, which leaves the processor running in idle mode, sacrificing both dynamic and static energy. When the utilization increases to 0.4 and larger, the sleep policy alone is not attractive since there is not enough static slack in the schedule anymore. The DSR-DP procrastination algorithm saves more energy than the DVS+sleep algorithm at most utilization levels. But it also loses to the DVS+sleep algorithm at one of the low and one of the high utilization cases. Our DVSleak algorithm produces the lowest energy among all algorithms, close to the optimal level. The biggest energy savings are observed for medium-level utilizations, which are common in practice.

On average, DVSleak shows its strength by saving 10% more energy than the dynamic DSR-DP algorithm. This is because DSR-DP uses a less aggressive slack reclamation scheme where only the



(a) Avg. c=25%WCET (b) Avg. c=50%WCET (c) Avg. c=75%WCET

Figure 7. 3 Tasks, Pattern 2, under Different Actual Execution Times (Variable) and Utilization

slack from higher priority tasks can be reclaimed for either task slowdown or procrastination. Our experiments revealed that this limitation often results in tasks running at higher frequencies than necessary. As long as a low-priority job has finished its execution, assigning its dynamic slack for other high priority tasks is still safe for meeting deadlines. Our DVSleak exploits that feature and uses a more flexible slack reclamation and task delaying scheme. Our experiments show that DVSleak outperforms the best prior scheme, to the best of our knowledge.

DVSleak also saves 20% more energy than the DVS+sleep algorithm and 50% more energy than the pure DVS algorithm. The energy cost of DVSleak is often close to the lower energy bound, never exceeding it by more than 25%.

Figure 7 depicts the energy of these DVS algorithms under dynamic execution pattern two. In contrast to pattern one, dynamic execution patterns exhibit a variable task execution time among different jobs, which results in higher energy than pattern one in corresponding cases. When considering the energy savings of these DVS algorithms, we observe that DVSleak again shows its advantage, particularly for medium and high utilization scenarios. Even with varying workloads, the delay policy generates more opportunities for sleeping than any of the other policies. It saves 40% more energy on average than the pure DVS algorithm. For all execution patterns, the energy produced by DVSleak is also very close to the lower bound in most of the utilization cases.

Patterns three and four are omitted since their energy is only insignificantly higher ($< 1\%$) than that of pattern two. While the three patterns (patterns 2, 3, and 4) follow different fluctuations in execution time, DVSleak works equally well for all patterns. It saves 5-10% more energy on average than the DSR-DP algorithm and 40% more energy on average than the pure DVS algorithm. Overall, the combined sleep and delay scheme, DVSleak, exhibits stable performance under different patterns due to the feedback control scheme used in our DVS algorithm, which adjusts automatically according to workload variations.

Occasionally, DSR-DP performs worse than DVS+sleep. While the former utilizes a dynamic procrastination scheme, the latter sometimes benefits from early delay decisions that result in smaller remaining slack than any procrastination scheme with its heuristic nature for energy savings. This behavior is more dominant for very high utilizations.

8.2 Factors Affecting Energy

In a second set of experiments, we assessed the affect of task periods. Each experiment is based on a different task set with a utilization of about 60% and energy values normalized to the hyperperiod of task set two (even if measured for a longer/shorter hyperperiod). Actual task execution times are 50% of their WCET under pattern one (constant between jobs). The results are depicted in Table 1.

First, we contrast harmonic *vs.* non-harmonic periods in task sets one and two, respectively. We observe that non-harmonic periods (task set two) result in higher energy for the same policy, varying between 10-27% depending on the policy. This is caused by simultaneous releases of jobs, which are effectively folded under DVS policies to one longer execution period before choosing an appropriate frequency or sleeping. The savings of DVSleak over DSR-DP are around 2% for the harmonic and 2.7% for the non-harmonic case.

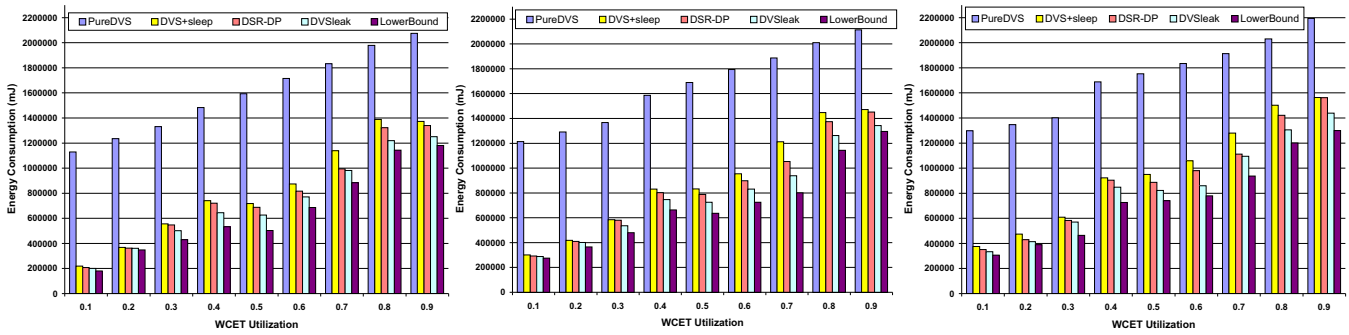
Second, we compare the effect of short *vs.* long periods for task sets two and three. Shorter periods again result in higher energy for the same policies, varying between 2-28% depending on the policies. More frequent releases result in less opportunities to sleep, which increases the overall energy requirements. The savings of DVSleak over DSR-DP are around 15% for shorter non-harmonic periods, which is larger than in the earlier experiments. Hence, the length of periods is a significant contributor to the effectiveness of different policies. Our DVSleak particularly excels over others when periods are short. DSR-DP, in contrast, performs worse than even a simple DVS+sleep policy.

8.3 Energy for Set of Ten Tasks

Figures 8 and 9 depict the energy for ten tasks of the different algorithms with execution patterns one and two, respectively. They also include the lower bound energy for each utilization case. We observe that ten tasks result in 5-10% higher energy depending on the policy than for three task. Similarly, the energy cost increases by about 5% when the utilization under WCET is increased from 0.25 to 0.5 and again from 0.5 to 0.75. The overall trends are similar to the 3-task experiments. Hence, we only summarize the observations. Sleep modes result in significant savings (up to 80%) over sleep-oblivious pure DVS. DVSleak outperforms DSR-DP

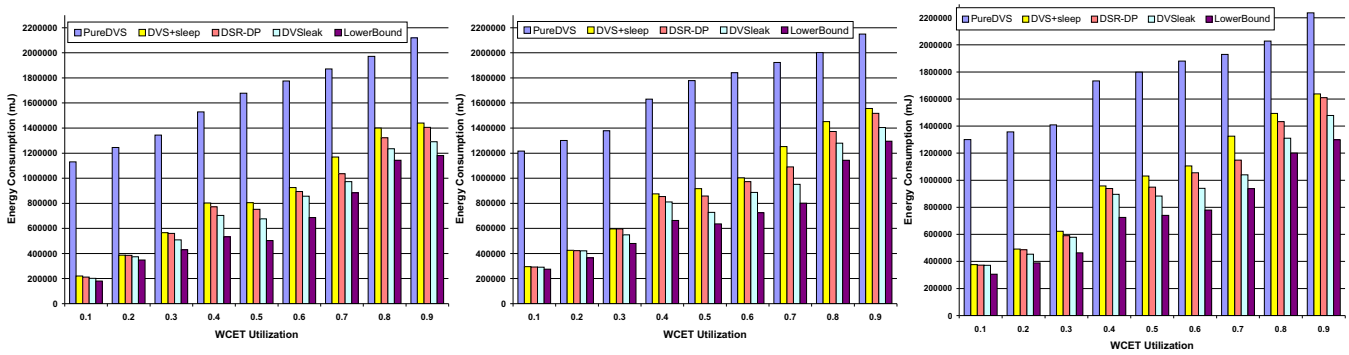
Task	Period			WCET			Energy $[mJ]$			
Set	T1	T2	T3	T1	T2	T3	DVS	DVS+sleep	DSR-DP	DVSleak
1	240	240	120	40	60	20	131.8	93.0	93.3	91.4
2	60	32	40	8	1	4	145.8	109.2	105.9	103.1
3	9	4.8	6	1.2	1	6	148.9	126.4	135.8	115.9

Table 1. Vary Periods/Overheads, U=50%, Pattern 1



(a) c=25%WCET (b) c=50%WCET (c) c=75%WCET

Figure 8. 10 Tasks, Pattern 1, under Different Actual Execution Times (Constant) and Utilization



(a) Avg. c=25%WCET (b) Avg. c=50%WCET (c) Avg. c=75%WCET

Figure 9. 10 Tasks, Pattern 2, under Different Actual Execution Times (Variable) and Utilization

while DVS+sleep provides the least amount of savings of the three sleep policies. In fact, DVSleak is close to the lower bound in most cases.

8.4 Summary of Results

These experiments provide a better understanding of the benefits of each policy. The DVS+sleep algorithm, the DSR-DP algorithm and the DVSleak algorithm show only insignificant differences under extremely low utilization cases. In such cases, there is always enough slack for suspending processor execution, no matter whether the release time of a task is delayed or not. For high utilizations, in contrast, hardly any slack exists at all, irrespective of delayed release times. Using the sleep policy alone in low utilization case is sufficient by itself to achieve virtually the same reduction in energy as the combined policy, albeit at a lower algorithmic complexity. At medium utilization levels, DVSleak excels due to its combined sleep and delay policy and shows its true potential of energy savings.

Scaling below the critical speed, as described in Section 6, did not provide additional benefits in the experiments due to a chosen shutdown overhead of 2ms. Given the shortest WCET of 1ms in our task set and an actual execution time of 0.5ms, scaling of T_A at 25% would result in 1.5ms execution plus the T_B part of 0.5ms at 100%, which adds up to 2ms. Hence, choosing a lower than critical speed is only beneficial for short jobs. This was demonstrated by significantly lower energy results of 15% under DVSlead than under DSR-DP (task set three in Table 1).

9. Related Work

Static power consumption caused by leakage current has attracted much attention in recent years. Conventional scheduling strategies are modified to be leakage-aware, which saves energy when combined with dynamic voltage scaling algorithms. Lee *et al.* [15] proposed greedy methods to locally maximize the duration of alternating idle and busy periods based on the worst-case execution time [15]. Algorithms are integrated into conventional dynamic priority scheduling and fixed priority scheduling policies. Their algorithm is most effective when many relatively short inter-task idle periods can be grouped together. But since actual execution times often diverge considerably from the WCET, a conceptual busy period will actually be interspersed with idle due to dynamic slack inflicted by early completion of tasks. The potential of such dynamic slack remains unused.

Quan *et al.* described an enhanced DVS algorithm to reduce both dynamic and static power consumption [20]. A latest release time of each job in the task set is computed off-line and subsequently used by an on-line scheduler. Their approach is based on fixed-priority scheduling while ours is based on dynamic priority EDF scheduling. Their online scheduler always delays the release time of a task to its latest start time (last chance) as long as the processor is idle. Such an aggressive scheme, as shown in this paper, is not always the most energy efficient solution. In our algorithm, we make delay decisions based upon the actual execution time of tasks *via* feedback, which is more energy efficient on average.

Jejurikar *at al.* enhanced EDF scheduling with a procrastination algorithm [12]. A delay interval is calculated for each task, which only considers static task set information and may result in a pessimistic schedule. An extension of the latter work also exploits dynamic slack for suspension [11]. They promote a policy similar to our trade-off between scaling and delaying for suspension in Section 4. However, they do not provide a look-ahead policy for deciding whether to delay early or late as detailed in Section 5. Hence, they report less than 1.5% energy savings over their static procrastination algorithm. In contrast, we observe up to 15% savings over static procrastination, which shows that the latter policy ("delay now or later") is *crucial* in achieving additional energy savings. Our approach benefits from the knowledge about actual execution times obtain by the feedback approach to make such delay decisions. It further capitalizes on dynamic variations in execution time due to feedback DVS while their scheme is more limited in that respect.

Aydin *et al.* developed a system-wide approach to energy management of real-time tasks and devices under EDF by non-linear constrained solving [3]. They incorporate the concept of a critical speed [12], albeit reformulated as energy relative to CPU speed. Their design allows for sleep modes, yet their evaluation does not cover sleep modes while our work does.

Chen and Kuo addressed procrastination in the context of rate-monotone scheduling [5]. Our work, in contrast, builds on EDF scheduling, which does not allow a direct comparison. But past work on DVS indicates advantages of dynamic priority scheduling, such as EDF, over static schemes [19]. Zhang *et al.* presented a compiler-supported solution to reduce leakage energy [24]. Dataflow analysis is employed to identify basic blocks that do not utilize a given functional unit, which is temporarily deactivated by compiler-generated software instructions. While their solution targets micro-architectural effects within a processor, our approach takes a macro-level view that temporarily puts the processor, including *all* of its resources, into sleep mode.

Andrei *et al.* presented an optimal algorithm for combined DVS and leakage control to reduce energy for time-constrained systems with dependent tasks represented in a precedence task graph assuming a continuous voltage range [1]. Their work combines DVS with adaptive body biasing (ABB) to reduce leakage based on either repeaters or fat wires for on-chip buses. They also show that discrete voltage levels make the problem NP-hard and develop a heuristic approach in later work that tries to reduce the number of voltage transitions [2]. Our work does not consider the effect on buses and assumes independent tasks under EDF. Since we consider discrete voltage ranges, as seen in actual hardware, we also use a heuristic approach but based on feedback of actual execution times (in contrast to their variation of WCET). Most of all, ABB reduces leakage in a manner orthogonal to our work, *i.e.*, during our DVS modulations, ABB can further reduce leakage power but during sleep intervals our method goes beyond this by practically eliminating leakage altogether.

10. Conclusion

Static power consumption has shown to be a crucial concern for ever-smaller fabrication sizes of processors. Static power is caused by leakage current, which even exists in the absence of logic operations in a CMOS circuit. In this paper, we presented a combined leakage reduction and DVS algorithm, DVSleak. We pointed out that greedily delaying the release time of a task to put the processor into sleep mode does not necessarily yield the most energy-efficient solution. The delay decision has to be made considering a task's dynamic execution behavior. Hence, a static delay policy does not suffice. DVSleak uses on-line information to derive a combined DVS and leakage-aware schedule. After the DVS scheme has assigned a speed for a job, a novel feedback delay policy determines the job's release time according to its expected actual execution time, which is provided by a feedback controller. It incorporates preemption handling and models wakeup overhead in terms of power and time. Our experiments show that our combined algorithm saves 50% more energy on average than a pure DVS algorithm. DVSleak further saves 20% more energy on average than a more simplistic DVS+sleep approach and 8.5% or more than dynamic slack reclamation with procrastination. For shorter periods, these savings increase to 15%. Furthermore, some prior schemes require special hardware support beyond DVS and sleep modes while DVSleak does not. This makes DVSleak the best combined DVS/leakage regulation approach for real-time systems that we know of.

References

[1] A. Andrei, M. T. Schmitz, P. Eles, Z. Peng, and B. M. Al-Hashimi. Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In *Design, Automation and Test in Europe*, pages 518–525, 2004.

[2] A. Andrei, M. T. Schmitz, P. Eles, Z. Peng, and B. M. Al-Hashimi. Simultaneous communication and processor voltage scaling for dynamic and leakage energy reduction in time-constrained systems. In *International Conference on Computer-Aided Design*, pages 362–369, 2004.

[3] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *IEEE Real-Time Systems Symposium*, pages 313–322, 2006.

[4] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Comput.*, 53(5):584–600, 2004.

[5] J.-J. Chen and T.-W. Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, page (accepted), June 2006.

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

[7] 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.

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

[9] Intel. *Intel PXA255 Processor: Electrical, Mechanical, and Thermal Specification*, Feb. 2004.

[10] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 37–46, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[11] R. Jejurikar and R. Gupta. Dynamic slack reclemation with procrastination scheduling in real-time embedded systems. In *Design Automation Conference*, June 2005.

[12] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 275–280, New York, NY, USA, 2004. ACM Press.

[13] D. Kang, S. Crago, and J. Suh. A fast resource synthesis technique for energy-effi cient real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[14] Y.-H. Lee and C. M. Krishna. Voltage-clock scaling for low energy consumption in fi xed-priority real-time systems. *Real-Time Syst.*, 24(3):303–317, 2003.

[15] Y.-H. Lee, K. P. Reddy, and C. M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *EuroMicro Conf. on Real Time Systems*, pages 105–112. IEEE Computer Society, 2003.

[16] S. M. Martin, K. Flautner, T. N. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In L. T. Pileggi and

A. Kuehlmann, editors, *Intl. Conference on Computer Aidded Design*, pages 721–725. ACM, 2002.

[17] K. Nowka, G. Carpenter, and B. Brock. The design and application of the powerpc 405lp energy-effi cient system on chip. *IBM Journal of Research and Development*, 47(5/6), September/November 2003.

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

[19] P. Pillai and K. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Symposium on Operating Systems Principles*, 2001.

[20] G. Quan, L. Niu, X. S. Hu, and B. Mochocki. Fixed priority scheduling for reducing overall energy on variable voltage processors. In *25th IEEE Real-Time System Symposium*, pages 309–318. IEEE Computer Society, 2004.

[21] C. Scordino and G. Lipari. Using resource reservation techniques for power-aware scheduling. In *International Conference on Embedded Sofware*, 2004.

[22] 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.

[23] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, page 374, Washington, DC, USA, 1995. IEEE Computer Society.

[24] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. Irwin, and V. De. Compiler support for reducing leakage energy consumption, 2003.

[25] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting dynamic voltage scaling. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 84–93, May 2004.

[26] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 203–212, June 2005.