Hybrid Leakage and Voltage Reduction under 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 resulted in an increasing need to conserve power of processors. While dynamic voltage scaling (DVS) is effective in reducing dynamic power, the latest dies are increasingly dominated by static power. For such processors, voltage/frequency pairs below a critical speed result in higher power consumption 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. This combined DVS/leakage algorithm results in an average of 45% additional power savings over a leakage-oblivious DVS algorithm in experiments. Its benefits over prior schemes are significant in that it neither requires special hardware support beyond DVS and sleep modes, nor does it assume execution times equal to their worst-case bounds.

1. Introduction

Power consumption in a CMOS-based processor consists of three elements: dynamic, static, and short-circuit power [11]. Most of the previous work on dynamic voltage scaling (DVS) only considers dynamic power consumption of a CMOS circuit while ignoring the static portion [16, 13, 1, 8, 3, 4, 12, 15]. Static power consumption stems from leakage current that exists even in the absence of logic operations of a circuit. As pointed out by Jejurikar *et al.* [7], when voltage supply is reduced below a certain threshold value, static power exceeds the dynamic power and becomes the dominant cause of power consumption *per se*. The processor frequency associated with this threshold voltage is called the *critical speed*. Above the threshold voltage, the total energy per cycle increases as the processor voltage scales up and can operate a at higher frequency resulting in higher overall performance. But below the threshold voltage, the total energy per cycle *also increases* as the voltage scales down, even through only *lower frequencies* with lower performance can be sustained. 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 / below the critical speed.
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.

Any combined DVS/leakage policy has to take into account the above issues 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 [6, 10, 7]. 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 [9]. Since actual execution times often diverge considerably 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. Jejurikar *et al.* assume that a power manager, implemented as a controller in hardware, handles interrupts and sets timers when new tasks are released [7]. 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 bundled together or scattered apart to increase the opportunity of entering the sleep mode.

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 DVSleak, our algorithm. Section 6 shows experimental results based on simulation. Section 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. 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.

We use the power model of a CMOS circuit first presented by Martin *et al.* [11]. 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 [11]. Similar power models are also used in related work [7, 14]. Hence, we only consider static and dynamic power in our model. 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 consumption is given by:

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

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

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 [7]. 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 transition into and out of a sleep mode does not come without cost. Such a transition incurs additional energy 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 [5]. 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 sleep overhead. Let the overhead of entering and exiting the sleep mode also be included in the sleep threshold derived below.

Let p_{idle} be the power consumption when the system is idle. 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 scheduler at system initialization.

In the following section, 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 feedback-DVS described in detail elsewhere [18, 19]. Instead of executing each task at a uniform speed, a task's worst-case execution time is divided into two subtasks, T_A and T_B, as shown in Figure 1. Their worst-

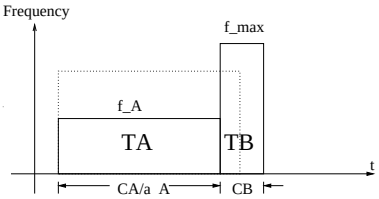


Figure 1. Task Splitting

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, we incur at most one speed change for every task and keep the impact of voltage and frequency switching overhead to a minimum. The details of the feedback-DVS scheme can be found in [18, 19]. 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.** With this scheme, we can safely scale the frequency of T_A exploiting the available slack while T_B executes at the maximum frequency following a last-chance approach [2]. 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 slack (idle time) 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 the static power consumption.

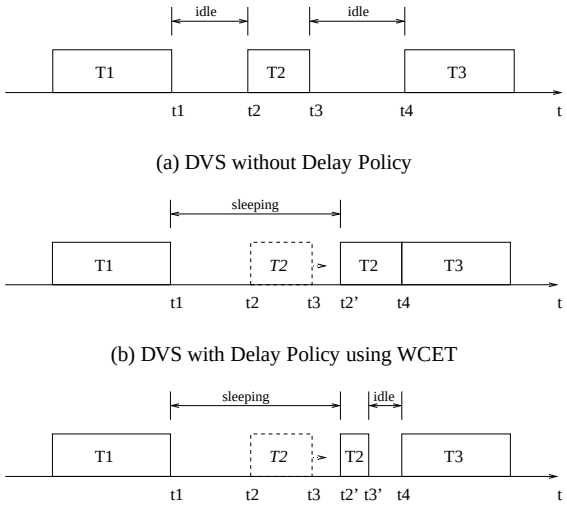


Figure 2. Combining DVS and Leakage Savings

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 processor energy consumption in an idle state, although lower than the energy consumption in a non-idle running state, is still significantly larger than the energy consumed in sleep mode.

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 energy consumption the processor pays is dynamic power as well as the wakeup overhead. Figure 2(b) is the ideal case where T_2 com-

pletes 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 or idle time in the schedule. The existence of slack is either due to the underutilization of system workload, which can be determined statically. Or, it is due 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, decreases the amount of slack available in the schedule, as depicted in Figure 3(b). Similarly, delaying the activation time of a task by putting the processor into a sleep mode also decreases the amount of slack, as depicted in Figure 3(c). At any time 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 determine the distribution of the slack between these two schemes. This dilemma can be solved based on the critical speed (frequency). We prefer a lower frequency over delaying a task as long as the resulting frequency is higher than the critical speed, *i.e.*, if such a choice results in lower energy consumption. Conversely, when our frequency scaling scheme suggests a speed lower than the critical speed, we default to the critical speed and activate the delay scheme. This policy reflects a best effort to reduce power. According to the above analysis, whenever a task completes and a new task T_i is released, our DVS algorithm uses a feedback-EDF scheme (see [18]) to calculate a frequency level f_i. The actual frequency f assigned to task T_i is defined by:

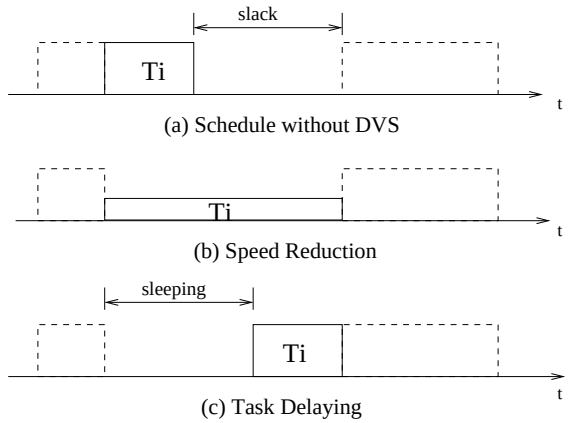


Figure 3. Speed Reduction vs. Task Delaying

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

Given the actual frequency of task T_i, a corresponding voltage can also be determined. 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. Delay Policy

The example in Section 3 seems to imply that a task should always be delayed as much as possible against its deadline. This is also the strategy used in previous work [7, 14]. 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 the real world, 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 that at any time t, the DVS algorithm can infer the amount of slack s_t in the schedule.

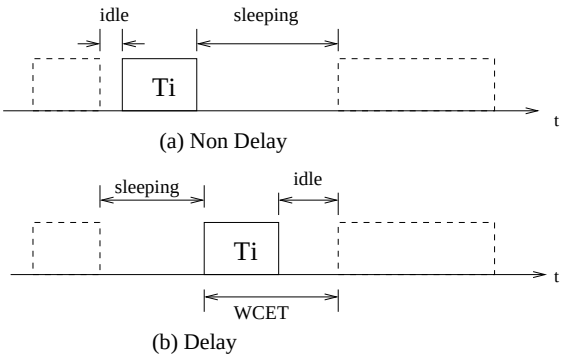


Figure 4. Delay vs. Non-delay

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 standard EDF scheduling algorithm. 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.

We use the following rule 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 $C_i/\alpha_i - c_i < t_r - t$.

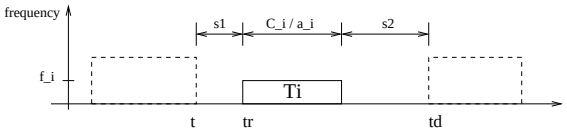


Figure 5. Rules for Task Delaying

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 still 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 (see [18] 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. Experiments

We implemented our leakage-aware DVSleak algorithm in a simulation environment, 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, which is the maximal frequency supported by the processor. We use the same approach as in [7] to compute the corresponding power consumption as $550mW$, $650mW$, $990mW$ and $1480mW$ for frequency levels 25%, 50%, 75% and 100%, respectively. The processor enters an idle frequency whenever none of the tasks are ready. As in previous work [7, 14], the idle power is assumed to be 240mW, E_{sd} is $483\mu J$, and t_{th} is $2ms$. The threshold frequency level is set to 41% of f_m.

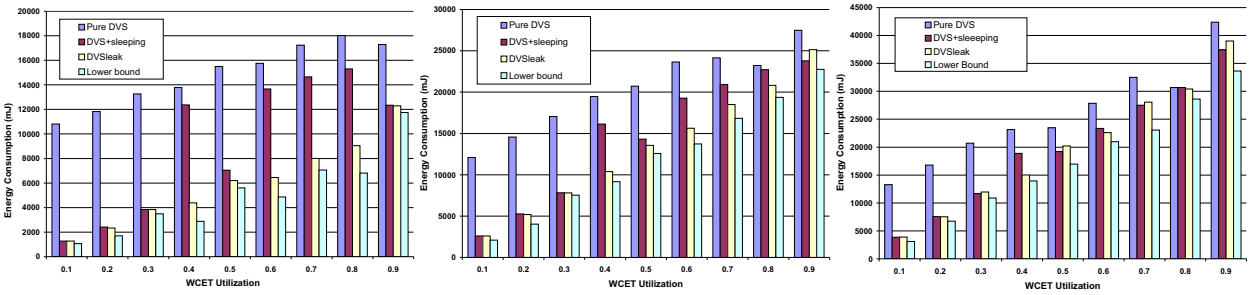
In order to assess the energy saving potential of our combined leakage-aware DVSleak algorithm, three different algorithms are implemented in the simulator.

1. A pure feedback-DVS algorithm without any leakage power saving [18]. The algorithm does not observe trade-offs due to the threshold frequency, *i.e.*, the frequency can be scaled below this threshold.
2. A feedback-DVS algorithm with a sleep policy. This algorithm puts the processor into sleep mode whenever the idle slack in the schedule is longer than the sleeping threshold. The algorithm exploits the threshold frequency in that no tasks will be scaled below that frequency. Hence, a frequency of 25% of f_m will never be used. However, this algorithm does not contain any delay policy to postpone the release of a task.
3. DVSleak, a hybrid feedback-DVS/sleep algorithm with a delay policy, as outlined in the last section. This algorithm is the most aggressive one. It not only puts the processor into a sleep mode. It also delays the release time of tasks according to our delay rules. The delay rules increase the length of the sleep duration, which saves more energy than other algorithms. Our experimental results show that this is mostly (but not always) the case. DVSleak also exploits knowledge about the threshold frequency.

A suite of task sets with synthetic CPU workloads was used in the experiment. Each task set contains three independent periodic tasks whose worst-case execution time varies from 0.1 to 0.9 with an increment of 0.1. The actual execution time of a task follows four different patterns.

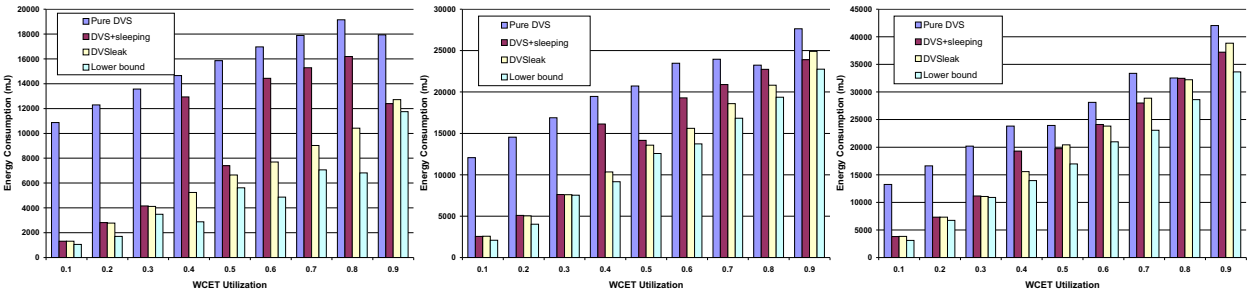
1. In pattern one, a task's actual execution time remains constant between different jobs.
2. In the second pattern, the actual execution time of a job starts at 50% of the task's WCET before spiking to a peak value c_m every 10th job and then receding with a decay of $c_i = 1/2^{(t-c_m)}$ again. This 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 [10,1000]. Each task's period was chosen so that the worst case utiliza-



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

Figure 6. Energy Savings for 3 Tasks, Pattern One under Different Actual Execution Times (Constant) and Utilization



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

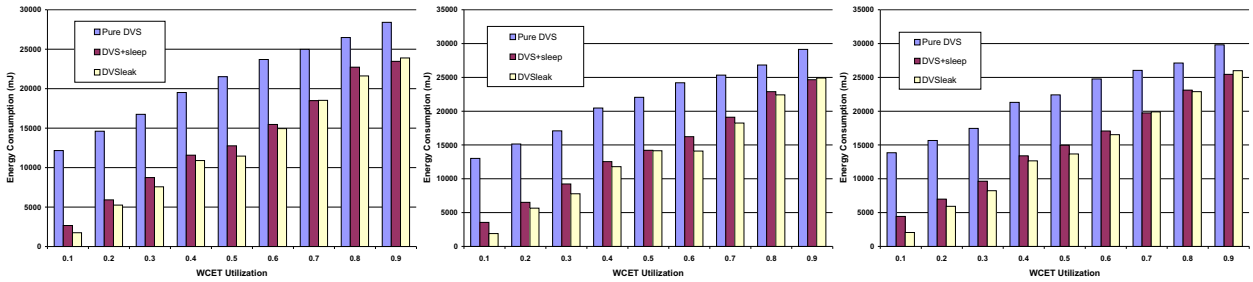
Figure 7. Energy Savings for 3 Tasks, Pattern Two under Different Actual Execution Times (Variable) and Utilization

tion of the task set varies from 0.1 to 1.0 in increments of 0.1.

In order to make a comparison, we also calculate a lower bound on energy for each utilization case. In the schedule for the lower bound, the entire task set runs at either the ideal optimal speed or the critical speed, whichever the greater. The number of times the processor gets into sleep or idle state is also minimized by assigning a longest busy interval for the task set, which equals the maximal response time of the longest period task. Such assumptions make it possible to derive a lower bound on energy overhead for processor state transitions.

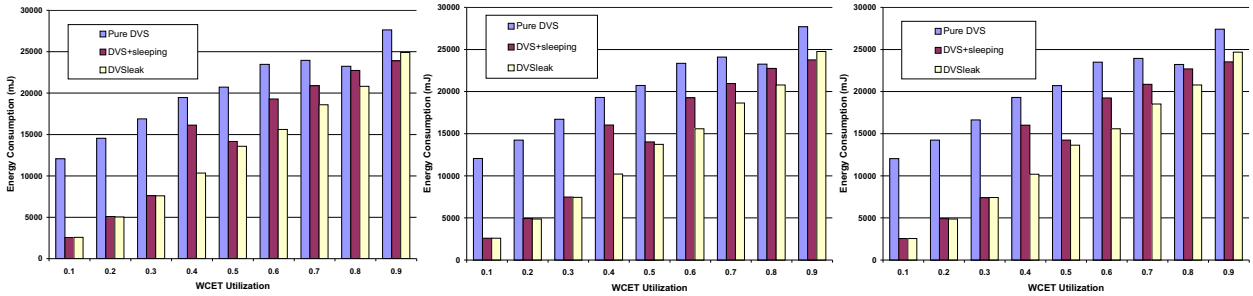
Figure 6 depicts the energy consumption of the three different algorithms with execution pattern one, as well as the lower bound energy consumption for each utilization case. We see significant energy savings at low utilization because of the existence of large amounts of slack. Putting the processor 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 0.6 and larger, the sleep policy alone is not attractive since there is not enough *ad-hoc* slack in the schedule anymore. On average, only 10% more energy is saved over pure DVS. DVSleak with its combined sleep and delay policy, in contrast, shows its strength by saving 50% more energy on average than the pure DVS scheme and 40% more energy on average than the pure sleep policy. It is interesting to note that the delay policy performs at par with the sleep policy for several cases, such as the 0.1 and 0.2 utilization cases. Figure 6(b) and 6(c) also show that the delay policy even costs more energy than the non-delay policy at 0.9 utilization, where already limited slack is further reduced by the delay policy, which results in higher processor frequencies than that of a pure sleep policy.

Figure 7 depicts the energy consumption of these three DVS algorithms under execution pattern two. In contrast to pattern one, execution times vary dynamically among different jobs, which results in higher energy consumption than pattern one in corresponding cases. When we look at the energy savings of the three DVS algorithms, we see that DVSleak again shows its advantage under medium and high utilization. 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 both execution patterns, the energy con-



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

Figure 8. Energy Savings for 10 Tasks, Pattern Two under Different Actual Execution Times (Variable) + Utilization



(a) Pattern 2 (b) Pattern 3 (c) Pattern 4

Figure 9. Energy Savings for 3 Tasks, Dynamic Pattern 2/3/4 when Average Execution Time = 50% WCET

sumption produced by DVSleak is also very close to the lower bound in most of the utilization cases.

We further increase the number of tasks in a task set from 3 to 10. The increase in number of tasks limits the effectiveness of the delay policy. It cannot produce sufficiently long intervals to benefit from sleeping. Nonetheless, Figure 8 illustrates that DVSleak exhibits stable savings in energy irrespective of the number of tasks. It achieves almost the same amount of savings over pure DVS observed for 3 tasks (Figure 7). The energy savings over the pure sleeping algorithm are not as significant as that for 3 tasks. Still, DVSleak saves 10% more energy on average than the sleep policy for 10 tasks. These results clearly show the adaptiveness and stability of DVSleak under different workloads.

Figure 9 compares the performance of different algorithms under three different dynamic patterns when the ratio of average execution time to WCET is fixed. Although the three patterns (patterns 2, 3, and 4) follow different fluctuations in execution time, DVSleak works equally well for all patterns. It saves 15% more energy on average than the sleep policy and 30% more energy on average than the pure DVS algorithm. Overall, the combined sleep and delay algorithm, 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.

These experiments provide a better understanding of the three policies. The pure sleep policy and DVSleak do not show much difference under extremely low or extremely high utilization cases. In the former case, there is always enough slack for turning off the processor, no matter whether we delay the release time of a task or not. In the later case, there is hardly any slack at all, no matter how the release time of a task is delayed. Using the sleep policy alone in such a 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, DVSleak excels due to its combined sleep and delay policy to show its true potential of energy savings.

7. 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 the system energy together with dynamic voltage scaling algorithms. Lee *et al.* [10] proposed greedy methods to locally maximize the duration

of alternating idle and busy periods based on the worst-case execution time [10]. 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 that 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 [14]. 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 [7]. A delay interval is calculated for each task, which only considers static task set information and may result in a pessimistic schedule. Our scheme is integrated with the online scheduler and is able to convert dynamic slack, generated due to the critical speed threshold or the early completion of tasks, into idle or, preferably, sleep time. Their approach also 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.

Zhang *et al.* presented a compiler-supported solution to reduce leakage energy consumption [17]. Data-flow 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.

8. Conclusion

Static power consumption has shown to be a critical design concern in dynamic voltage scaling algorithms. 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 task, the delay policy determines task's release time according to its expected actual execution time, which is provided by a feedback controller. Our experiments show that such a combined algorithm saves 50% more energy on average than a pure DVS algorithm. The combined sleep and delay policy saves 40% more energy on average than a pure sleep policy. We are currently pursuing an implementation on an IBM PowerPC 405LP board under Linux with support for DVS and various sleep modes [19].

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.