Sprint-and-Halt Scheduling for Energy Reduction in Real-Time Systems with Software Power-Down [*]

Padmanabhan Pillai and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122, U.S.A.
{pillai, kgshin}@eecs.umich.edu

Abstract

Mobile computing platforms are performing increasingly complex and computationally intensive tasks. To help lengthen useful battery life, these platforms often incorporate some form of hardware power-down that is controlled by the system software. Unfortunately, these often incur substantial transition latencies when switching between power-down and active states, making them difficult to use in time-critical embedded systems.

This paper introduces a class of *sprint-and-halt* schedulers that attempt to maximize the energy savings of software-controlled power-down mechanisms, while simultaneously maintaining hard real-time deadline guarantees. Several different algorithms are proposed to reclaim unused processing time, defer processing, and extend power-down intervals while respecting task deadlines. Sprint-and-halt schedulers are shown to reduce energy consumption by 40–70% over typical operating parameters. For very large or small state transition latencies, simple approaches work very close to theoretical limits, but over a critical range of latencies, advanced schedulers show 10–20% energy reduction over simpler methods.

[*]The work reported in this paper is supported in part by the U.S. Airforce Office of Scientific Research under Grant AFOSR F49620-01-1-0120.

1 Introduction

In recent years, there has been a significant shift toward mobile computation and communication platforms and devices. This shift has occurred in both the realm of general-purpose computing with the increase in use of laptop computers and PDAs, and in the embedded computing realm with an increasing number of digital cameras, cellular phones, and portable medical devices running complex applications and operating systems on embedded microprocessors. Critical to these systems is the limited stored energy available in a portable form factor. There is a fundamental trade-off between the weight and size of the device, the processing speed of the processor, and the useful battery life of device.

Unrelenting market pressures have created increasingly-sophisticated applications in increasingly-compact devices, such as multimedia and web capabilities on cell phones and gaming on PDAs. These demanding applications require the use of more powerful processors to provide the user a responsive experience. This has made the need for power management to minimize energy waste in such systems critical.

There has been recent interest and significant research on *Dynamic Voltage Scaling* (DVS) techniques [3, 5, 15] that attempt to trade off performance and battery life by adjusting the operating frequency and voltage of the processor to match the computational load on the system. As processors are composed mostly of CMOS logic gates, the energy expended is proportional to the charge on the gate capacitances, and thus, a quadratic improvement in energy is attained when voltage is reduced. However, DVS requires software adjustable voltage regulators and clock generators that may not be available on many platforms.

More generally available is the much simpler concept of a *software-controlled power-down* mechanism. This may take a variety of forms. One simple form is a processor `halt` instruction that will effectively stop the CPU core, and keep it in a low-power standby state until a subsequent interrupt. This is a low-overhead, fast operation that can simply be invoked in place of an idle-loop to reduce wasted energy by the processor. More generally, there may be some mechanism to place various system components into a standby state, incurring a finite time overhead to power down and up the system.

This time overhead of switching hardware power states adds complexity to managing power in embedded devices. In particular, these devices often require strict timeliness guarantees for executing their resident tasks. In such systems, any adjustment of hardware power states must ensure task deadlines are not violated while maximizing energy savings.

In this paper, we propose and evaluate a class of *sprint-and-halt* scheduling algorithms that provide real-time task scheduling, while maximizing the benefits of software-controlled power-down mechanisms. The rest of this paper is organized as follows. We will first present our general model of software-controlled power-down hardware, which is followed by a detailed description of several algorithms for sprint-and-halt scheduling of real-time systems. We then evaluate these algorithms with respect to energy savings, before ending with conclusions and a discussion of future work.

2 Background

As power dissipation becomes an increasingly critical limitation in mobile systems, various mechanisms have been introduced to help conserve and reduce wasted energy. The most general type of power conservation mechanism is based on changing the power state of hardware components, placing them in a low-power or standby state when not actively used. For general-purpose systems, mechanisms such as APM and ACPI [1] provide interfaces for software-controlled power-down of the system when not actively used. When explicitly notified by the user, such as when closing the lid on a laptop computer, or after some timeout interval without user input, the system enters a low-power state, and computation is halted until a subsequent wake-up event occurs. This works well in laptops and PDAs, which are usually idle when a user is not directly interacting with the system. However, most real-time applications are not considered interactive, and generally need to run continuously over extended periods of time. Battery-operated embedded systems cannot take advantage of simple timeout-based power-down to conserve energy.

Instead, for such systems, we need to take advantage of power-down mechanisms at much finer time-scales, halting operations between executions of periodic tasks. One hardware power-down mechanism that works well here is processor halt operation. Here, a special `halt` instruction puts the processor to a sleep mode, turning off the execution pipeline and disabling further computation. Although power is still supplied to the processor, along with a clock signal, much of the CPU core is deactivated and power dissipation is very low. A subsequent event, generally a hardware interrupt, will resume the processor core. Using the halt instruction in place of a more traditional idle-loop can greatly reduce the wasted energy executing empty spin loops. As the overhead of executing the halt operation and resuming on interrupt is very low, on the order of a few processor cycles, this mechanism may be safely employed without significantly affecting execution times or deadlines.

The halt instruction, if available on the processor, is effective for conserving energy, but only within the processing core. The rest of the system, such as buses, memory, and communication devices, will continue to draw energy at normal rates even when the processor is halted. With more sophisticated hardware, a system can provide an interface that allows a larger subset of system components to be deactivated under the control of software. Timers, memory controllers, and communication ports may be powered down to save considerable energy when not in active use. In particular, turning off the main system clock-generation circuitry will essentially shut off the processor and memory, and often any communications and I/O subsystems as well, saving considerably more energy than with processor halts alone. Taking this to an extreme, APM and ACPI suspend modes essentially turn off the entire system after saving all dynamic processor and system state to persistent storage, dropping power dissipation to zero.

These lower-power modes do not come for free. Unlike the simple processor halt, powering down external subsystems can incur substantial time and processing overheads for entering and leaving the low-power modes. In the extreme case of the ACPI suspend operation, the operating system must iterate through every system device driver, saving the current state, and then copy all memory to disk before powering down the system. Upon resume, this process is reversed to restore the system to the exact state it was previously in. The overheads for such operations is very high, and will typically require on the order of tens of seconds to complete. Although this can greatly reduce power consumption, it cannot be used in the short intervals between task invocations

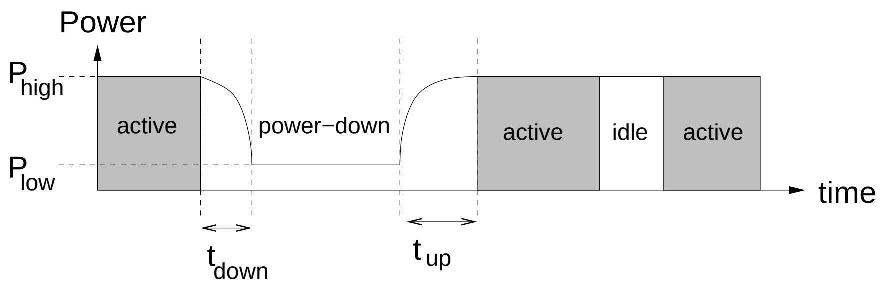


Figure 1: Parameters of system power model.

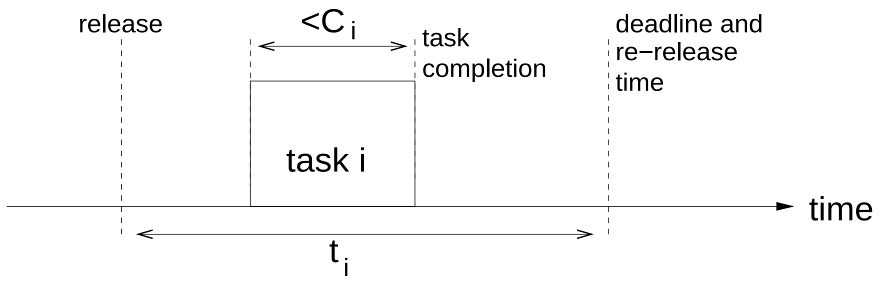


Figure 2: Periodic real-time task model parameters.

in a real-time system. At the opposite extreme, the halt operation incurs negligible to very low overheads, on the order of a microsecond, but may not provide significant energy savings in a system whose energy consumption is dominated by components other than the processor. Trading off power reduction level for improved switching latencies, a moderately aggressive approach may require only a few milliseconds of overhead, such as in waiting for the clock circuitry to stabilize on power-up. Various other software-controlled power-down mechanisms can vary anywhere between these extremes, trading off power reduction for time overheads.

3 System Model

Regardless of the actual software-controlled power-down mechanism available in a particular hardware platform, its use in a real-time system is primarily affected by the time overheads the mechanism incurs and how this would affect the timeliness of task execution. Therefore, we can generalize software-controlled power-down mechanisms and model them as follows. First, for simplicity, we assume that the platform dissipates power in a bimodal manner, consuming a constant P_{high} when in the active state, and P_{low} when in the power-down state. The transition from active to power-down state takes a constant time, t_{down}. We assume that there exists some time trigger, such as an external real-time alarm, that can be programmed to reactivate the system at a specific future time. Once triggered, the transition to active state takes t_{up} time. The average power dissipation during the transitions is P_{trans}, which can be anywhere between P_{low} and P_{high}, but we will assume the worst case of $P_{trans} = P_{high}$ unless noted otherwise. These parameters are illustrated in Figure 1.

We assume that the system follows the canonical periodic real-time task model. Each task i is *released* periodically, becoming ready to execute every t_i time units. The task is also characterized by a *worst-case execution time* (WCET) C_i, which indicates the maximum processing time it needs on each release/invocation. The relative deadline is equal to the period, so each task must complete execution within t_i of its release, i.e., must complete by the time it is re-released for its next invocation. These parameters are illustrated in Figure 2. The tasks are scheduled according to either the *rate-monotonic* (RM) or the *earliest deadline first* (EDF) priority scheduler. These are the most extensively-studied real-time scheduling mechanisms and cover a broad range of actual OS implementations. RM is a preemptive scheduler that assigns fixed priorities among tasks, giving the highest priority to the most-frequently executed task. EDF, on the other hand, assigns dynamic priority based on which task has the most imminent deadline, which varies over time. Assuming preemption and scheduler overheads to be negligible, the latter has a nice schedulability property that allows one to ensure a set of tasks is schedulable and all deadlines met by simply keeping the total worst-case processor utilization of the task set below one, i.e., $\sum C_i/t_i \leq 1$ [9].

Using the system models described above, we will in the next section design real-time scheduling algorithms that attempt to maximize energy savings from powering down the system, while ensuring real-time deadlines are met.

4 Sprint-and-Halt Algorithms

Existing real-time scheduling algorithms were not designed with energy-savings in mind. In particular, they do not consider how to incorporate software-controlled power-down mechanisms in the task schedule, and how to deal with the latencies incurred when switching between power states. In this section, we develop several novel algorithms to take advantage of power-down techniques while ensuring the schedulability of the real-time task set. These algorithms attempt to rapidly complete all work in the system (thus the term “sprint”), and then power down the system as long as possible (thus the term “halt”) to maximize the reduction in energy consumption and amortizing the transition latencies over long power-down intervals.

4.1 Real-Time Schedulers with Power-down

We first consider the standard RM and EDF schedulers and extend them as minimally as possible to incorporate power-down control in the task schedule. The goal of this first design is to ensure schedulability and task deadlines by leaving the actual execution schedule unaltered. Rather, this algorithm incorporates power-down such that all execution timings are left identical to that of plain vanilla EDF or RM scheduling.

This algorithm tries to replace any idle time in the schedule with a power-down event while preserving task timing. However, due to the latency of power-state change, power-down must be applied only when idle periods in the schedule are sufficiently long to cover the transition latencies. Based on the model parameters specified earlier, power-down is triggered only when $t_{idle} \geq t_{down} + t_{up}$, where t_{idle} is the contiguous idle period in the schedule. When power-down is invoked, the system is set to resume execution in $t_{idle} - t_{up}$ time, ensuring that the system is in the active state by the time the idle period expires.

Given the real-time assumptions of a task’s relative deadline equal to its period, and a work-

```
Assume $n$ tasks, sorted by deadline:
    $D_1 \leq D_2 \leq \cdots \leq D_n$
        /* this is needed for EDF */

upon task_release(task $i$):
    set $done_i$ to 0;
    update $D_i$ to $D_i + t_i$;
    resort task list by new deadlines;
        /* schedule by RM/EDF priority */

upon task_completion(task $i$):
    set $done_i$ to 1;
    if (for all $j$, $done_j$=1) then:
        $t_{now}$ = get_current_time();
        if ( $(D_1 - t_{now}) > (t_{down} + t_{up})$ ) then:
            set wakeup timer to $D_1 - t_{now} - t_{up}$;
            start power-down;
        else idle;
    else:
            /* schedule by RM/EDF priority */
```

Figure 3: Real-time scheduling with power-down

conserving RM or EDF scheduler, one can very easily compute t_{idle} online. When some task completes execution and no other tasks have any computation time remaining, an idle period in the schedule begins. This idle ends upon release of the next task, which, since the relative deadlines equal the task periods, will coincide with the earliest deadline among the tasks within the system. Hence, $t_{idle} = D_1 - t_{now}$, where D_1 is the earliest deadline in the system, and t_{now} is the current time when idle would normally start. Figure 3 shows a pseudocode implementation of this algorithm. For EDF scheduling, the set of tasks is already sorted by deadlines, so adding power-down is trivial. For RM scheduling, one needs to add structures to keep track of the deadlines. In practice, it is not necessary to actually sort the task set by the deadlines, as a simple scan to find the earliest deadline is sufficient.

This algorithm is very conservative, avoiding altering any timing from the normal EDF or RM schedule. However, as a result, it can only reduce power consumption under fortunate circumstances when a sufficiently long idle interval occurs in the normal execution of the tasks.

4.2 Work-Idle-Conserving Schedulers

To improve the energy savings of the previous scheduler, one can try to increase the duration of idle periods to allow longer intervals in low-power mode and amortize switching-time costs over longer periods. However, care must be taken to ensure that no task will be delayed and miss its

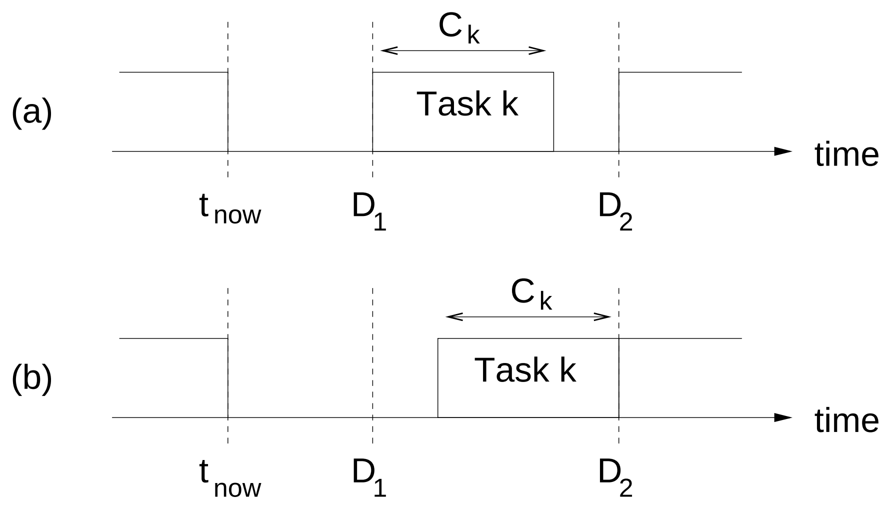


Figure 4: Example of deferral of task execution in work-idle-conserving scheduler. (a) Original execution schedule; (b) After deferral.

deadline.

A class of *work-idle-conserving* schedulers can help increase such idle durations. While there are tasks to execute, these schedulers follow the standard work-conserving RM or EDF scheduling policy. However, once all tasks have completed and the system enters idle, these schedulers becomes "idle-conserving" — they attempt to lengthen the idle period by deferring the next arriving task. This must be done conservatively to ensure future deadlines are not violated.

To this end, a simple algorithm for execution deferral looks ahead to the next arriving task, k, and will limit its effects to just this one task. Task k will arrive at time D_1, the earliest deadline in the system. Between time D_1 and time D_2, the next deadline in the system, task k will execute exclusively. If the WCET of task k, C_k, is less than $D_2 - D_1$, then we can defer the starting time of task k by $D_2 - D_1 - C_k$ without affecting its deadline or the execution of any other task. This is illustrated in Figure 4. Since C_k time is available before D_2, task k still completes all execution by D_2 as with the unaltered schedule. Furthermore, the effects of this deferral are local to the interval (D_1, D_2), so no other task's execution is affected by the deferral of task k.

There are two caveats when implementing this algorithm. First, it is possible that two tasks have coinciding deadlines at time D_1. In this case, two tasks are released simultaneously, and we should not attempt to defer execution based on an algorithm that assumes just one task executes exclusively after D_1. This case is handled by simply using $D_2 = D_1$ in case of coinciding deadlines. Since $C_k > 0$ for either task, no deferral is performed. The second issue is that the second deadline, D_2, may actually be for the invocation of task k released at time D_1. At the time idle begins (before time D_1), this invocation of task k has not yet been released, and its deadline has not yet been added to the system, so this case must be checked when computing $D_2 - D_1$. Figure 5 presents the power-down algorithm for the simple work-idle-conserving RM/EDF scheduler. As before, in the case of RM, it may be necessary to add structures to keep track of task deadlines.

Essentially, this algorithm conservatively extends the previous algorithm to allow the deferral of execution for a single task in an attempt to extend idle intervals. Although this will improve performance over the simple RM/EDF scheduling with power-down described earlier, there is no

```
Assume $n$ tasks, sorted by deadline:
    $D_1 \le D_2 \le \cdots \le D_n$
        /* this is needed for EDF */

upon task_release(task $i$):
    set $done_i$ to 0;
    update $D_i$ to $D_i + t_i$;
    resort task list by new deadlines;
        /* schedule by RM/EDF priority */

upon task_completion(task $i$):
    set $done_i$ to 1;
    if (for all $j$, $done_j$=1) then:
        $t_{now}$ = get_current_time();
        $t_{defer}$ = max{ 0, min{ $D_2 - D_1 - C_1, t_1 - C_1$ } }
        if ( $(D_1 + t_{defer} - t_{now}) > (t_{down} + t_{up})$ ) then:
            set wakeup timer to $D_1 + t_{defer} - t_{now} - t_{up}$;
            start power-down;
        else idle;
    else:
            /* schedule by RM/EDF priority */
```

Figure 5: Work-idle-conserving scheduler

guarantee that the deferral of the next task alone will provide greatly improved power-down time, particularly if tasks use significantly less than their WCETs.

4.3 Slack-Stealing Schedulers for Power-down

When tasks consume less than their WCETs, one would like to use the surplus time, or *slack*, as effectively as possible for power-down. However, with the simple approach of task deferral shown above, the slack is not directly taken into account, so a somewhat conservative mechanism is used to ensure that future deadlines are not violated. If one could accurately track the slack gained due to tasks completing early, then more aggressive deferral of task execution can be employed, while still ensuring that future deadlines are met.

In this next approach, called *slack-stealing scheduling for power-down*, the goal is to maintain an accurate count of the extra computing time (i.e., slack), and use this to generate longer idle intervals in the execution schedule. Existing slack-stealing techniques [8] use slack to provide time to real-time aperiodic tasks, increased execution time to variable runtime tasks, e.g., increasing rewards for increasing service (IRIS) [4], or to execute best-effort, non-real-time tasks. In this case, the computed slack time is used to determine the maximum period over which one can delay the execution of tasks (i.e., stay in an idle-coserving mode) to ensure the execution starting

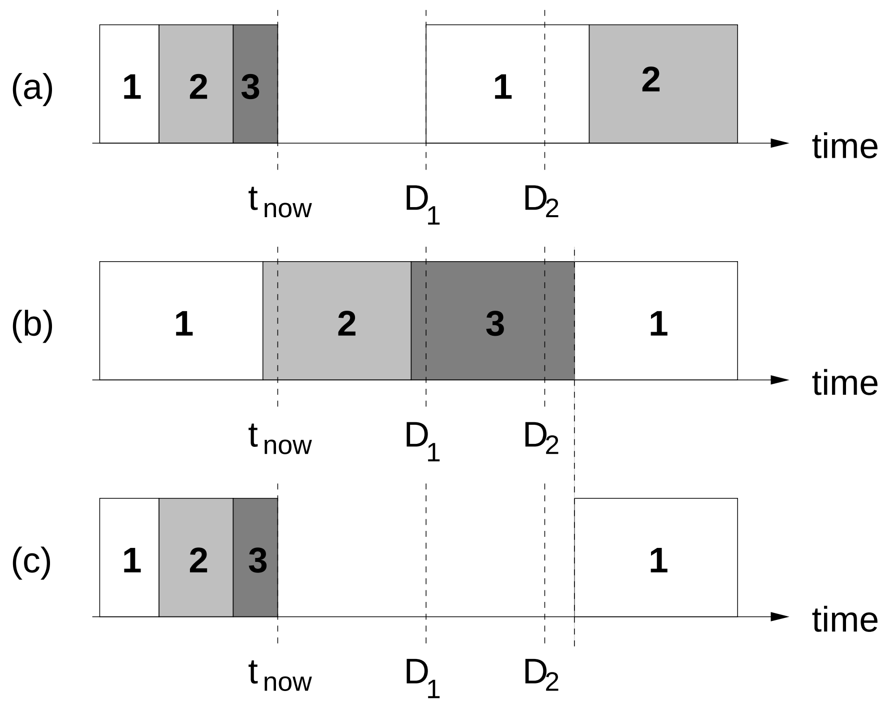


Figure 6: Slack-stealing scheduler example scenario. t_{now} is current time, where system enters idle. (a) Execution schedule for work-conserving scheduler indicates execution resumes at time D_1 when task 1 is released; (b) Canonical schedule assuming tasks always use exactly their WCETs indicates next invocation of task 1 would start after time D_2; (c) Slack-stealing power-down schedulers defer task 1 until the time indicated by the WCET schedule.

time is not delayed beyond that in the EDF or RM execution schedule assuming WCETs for all tasks. As long as the starting time for any part of a task occurs no later than in the EDF or RM WCET schedule, then task completion no later than in the WCET schedule is guaranteed, and, therefore, all deadlines are ensured to be met. The only exception to executing tasks no later than in the WCET schedule is that the single task deferral used in the previous work-idle-conserving schedulers is also applied when possible, which, as discussed earlier, will not cause any task to violate its deadlines.

As in the previous approach, the scheduling first proceeds in a work-conserving fashion. When an idle period is reached, the mode is switched to idle-conserving, and tasks that arrive in the future are deferred in a non-work-conserving manner. Once execution of a delayed task begins, the scheduler resumes work-conserving operation. While tasks are executing, the execution schedule, assuming WCETs for all tasks, is computed, so when idle occurs, the actual slack relative to the WCET schedule can be computed. This is used to determine the maximum duration over which the system may be powered down to ensure arriving tasks will begin no later than they would have in the WCET schedule. This is illustrated in the example in Figure 6.

The actual algorithm for slack-stealing EDF and RM scheduling is outlined in Figure 7. There are two general functions performed in the algorithm. First, a set of data structures is maintained that simulate the execution timing under the EDF or RM scheduler assuming WCETs for all tasks. These structures are updated while tasks are executed in a work-conserving manner. One should

```
Assume $n$ tasks, sorted by deadline:
    $D_1 \leq D_2 \leq \cdots \leq D_n$

upon task_completion(task $i$):
    simulate_execution();
    set $done_i$ to 1;
    if (for all $j$, $done_j$=1) then:
        $t_{now}$ = get_current_time();
        simulate_forward();
        $t_{resume}$ = max{ $t_{sim}$, min{ $D_2 - C_1$, $D_1 + t_1 - C_1$ }}
        if ( $(t_{resume} - t_{now}) > (t_{down} + t_{up})$ ) then:
            set wakeup timer to $t_{resume} - t_{now} - t_{up}$;
            start power-down;
        else idle;
    else:    /* schedule by RM/EDF priority */

simulate_release(task $i$):
    set $done_i$ to 0;
    set $cc_i$ to $C_i$;
    update $D_i$ to $D_i + t_i$;
    resort task list by new deadlines;

simulate_execution():
    $t_{now}$ = get_current_time();
    repeat while $t_{sim} < t_{now}$:
        find task $k$ such that $cc_k \neq 0$ and for all $j$, $D_j < D_k$ implies $cc_j = 0$
            /* for RM, replace $D_j < D_k$ with $t_j < t_k$ in line above */
        if $k$ exists, then:
            $t_{run}$ = min{ $cc_k$, $t_{now}$-$t_{sim}$, $D_1$-$t_{sim}$ };
            set $cc_k$ to $cc_k - t_{run}$;
        else $t_{run}$ = min{ $t_{now}$-$t_{sim}$, $D_1$-$t_{sim}$ };
        set $t_{sim}$ to $t_{sim} + t_{run}$;
        for all $j$ such that $D_j \leq t_{sim}$, simulate_release( task $j$ );

simulate_forward():
    loop:
        find task $k$ such that $cc_k \neq 0$ and for all $j$, $D_j < D_k$ implies $cc_j = 0$
            /* for RM, replace $D_j < D_k$ with $t_j < t_k$ in line above */
        if $k$ exists, then:
            if $done_k = 0$ jump out of loop;
            $t_{run}$ = min{ $cc_k$, $D_1$-$t_{sim}$ };
            set $cc_k$ to $cc_k - t_{run}$;
        else $t_{run}$ = $D_1$-$t_{sim}$;
        set $t_{sim}$ to $t_{sim} + t_{run}$;
        for all $j$ such that $D_j \leq t_{sim}$, simulate_release( task $j$ );
    end of loop;
```

Figure 7: Slack-stealing scheduler for power-down

note that the release times of tasks in the actual execution and the simulated schedule are identical, but in the actual execution, tasks will complete earlier than in the WCET schedule. The second function is triggered when some task completes and no further work is immediately available. Then, the algorithm simulates the continued execution of the WCET schedule forward in time, including the future releases of tasks, and determines the earliest time at which the WCET schedule indicates that a future-released task commences execution. The difference between this time and the current time essentially constitutes the available slack generated by tasks using less than their WCETs. The power-down interval is selected to terminate at this future time, or at the deferred start time computed by the previously-described work-idle-conserving mechanism, whichever is later. The system will then resume work-conserving execution until the next idle interval.

With this deferral technique, the system can guarantee timely execution of tasks (i.e., all tasks complete by their deadlines) by ensuring that execution occurs no later than in the WCET schedule, and that any greater deferral is limited to a single task with local effects that do not extend beyond any deadline (i.e., the work-idle-conserving mechanism described in previous section). Hence, the schedulability of the system and deadline guarantees are identical to the system with ordinary RM or EDF scheduling. This algorithm, although still an example of a bimodal work-idle-conserving scheduler, is more aggressive and has greater time scope than the previous schedulers, as it does permit deferral beyond multiple deadlines, allowing the deferral of multiple ready tasks.

4.4 Improved Slack-Stealing EDF

The slack-stealing EDF scheduler works well when the WCET schedule indicates a greatly deferred start time for tasks released in the future due to continued worst-case execution of currently released tasks. When the system is heavily-loaded (i.e., very little idle time in WCET schedule), this method can help greatly. However, when the system is lightly-loaded, there will be idle periods in the WCET schedule, and, as this simulated schedule is work-conserving, it may greatly limit the deferral time and, consequently, the power-down intervals.

This next approach modifies the slack-stealing EDF scheduler slightly to improve the deferral time and power-down intervals when the system is under-utilized. The goal is to defer task execution farther than the WCET EDF schedule would indicate, but still ensure deadlines of the tasks. This is accomplished by creating a specification of an alternate task set that fully utilizes the system, and using this in the simulation of the WCET schedule. For each actual task i, there is a task i' in the alternate set with an identical period, t_i. The WCET of task i', C_i' is such that $C_i' \geq C_i$. Since the period, and therefore deadlines, of task i' are identical to those of the real task i, and since the WCET is at least as long, any schedule that can guarantee the timely execution of task i' will also suffice for i. As this is true for all tasks, as long as the alternate task set is schedulable, so is the real task set using the same execution schedule.

To create a schedulable alternate task set for an EDF scheduler, assuming negligible preemption and scheduler overheads, one needs to simply ensure that the total utilization does not exceed 1, i.e., $\sum_i C_i/t_i \leq 1$ [9]. Figure 8 shows the algorithm to generate a schedulable alternate task set that fully utilizes the system. First, the worst-case utilization of the given task set is computed as U. For an under-utilized system, this value is strictly bounded, $0 < U < 1$. For a fully-utilized system, $U = 1$, so the alternate task set is constructed to ensure this. Each alternate task is given the same period as its real counterpart, but its WCET is multiplied by a factor of $1/U$. Now, this

```
Assume $n$ tasks
Each task $i$ has period $t_i$ and WCET $C_i$

At startup:
    Compute $U = \sum_i C_i/P_i$
    For each task $i$:
        create alternate task $i'$
        set $t_i'$ to $t_i$
        set $C_i'$ to $C_i/U$

Proceed with slack-stealing EDF scheduling
    but use alternate task set for simulated WCET schedule
```

Figure 8: Improved slack-stealing EDF scheduler for power-down

alternate task set is used for the simulated WCET schedule in the slack-stealing EDF scheduler. As this alternate schedule has the same deadlines and greater execution time available for each task than needed for the given task set, ensuring all tasks execute no later than in this alternate WCET schedule suffices to guarantee task deadlines. Again, the one exception to starting a task no later than in the WCET schedule is when the work-idle-conserving EDF scheduler's single task deferral is applied, but its effects are localized to the single task and do not cross deadlines, so task deadline guarantees are maintained.

By using an alternate task set that fully utilizes the system, has the same deadlines, and has greater execution time required for each task than the given task set as the reference for task start times, greater deferral times, longer power-down intervals, and lower energy consumption can be achieved by the improved slack-stealing EDF algorithm.

4.5 Handling Multiple Power-down States

The sprint-and-halt algorithms as discussed support hardware with a two power states: active and power-down. However, often, there may be multiple power-down states available on a platform. These states will have varying power-consumption rates, as well as latencies to resume active-state operation. For example, a system may have a low-latency processor halt mechanism that moderately reduces power consumption, as well as a system power-down mode that greatly reduces power, at the expense of a longer resume latency.

It is possible to modify all of the algorithms introduced here to support more than one power-down state. First, one needs to determine the power model of the system with two or more power-down states. To keep everything consistent, assume simply that each power state x has its own constant power-consumption rate, $P_{low,x}$. Furthermore, only transitions between each low-power state and the active state are considered, i.e., do not transition from one power-down mode to another directly. Each state x has transition latencies $t_{up,x}$ and $t_{down,x}$ to switch to and from active state. During the transitions, an average of power of $P_{trans,x}$ is dissipated.

Given this model, the algorithms need to choose the power state that requires the lowest energy cost for any power-down interval t_{pd}. The energy consumed by state x, E_x, is expressed as a function of the power-down interval, t_{pd}, as:

$$E_x(t_{pd}) = (t_{down,x} + t_{up,x})P_{trans,x} + (t_{pd} - t_{down,x} - t_{up,x})P_{low,x}$$

Assuming two states, x and y, where $(t_{down,x}+t_{up,x}) < (t_{down,y}+t_{up,y})$ and $P_{low,x} > P_{low,y}$, i.e., y is a lower-power, longer-latency state than x, it is better to switch to state y when $E_x(t_{pd}) > E_y(t_{pd})$. Solving this for t_{pd}, it is better to use state y when:

$$t_{pd} > \frac{(t_{down,y} + t_{up,y})(P_{trans,y} - P_{low,y}) - (t_{down,x} + t_{up,x})(P_{trans,x} - P_{low,x})}{P_{low,x} - P_{low,y}}.$$

The right-hand side of the inequality depends only on system parameters, so it can be computed ahead of time and used as a constant, called $t_{E_x=E_y}$. So, to support two power-down states, the algorithms simply decide based on the power-down interval:

$$\begin{array}{rcl} \text{idle} & : & t_{pd} < (t_{down,x} + t_{up,x}) \\ \text{state } y & : & t_{pd} > \max\{(t_{down,y} + t_{up,y}), t_{E_x=E_y}\} \\ \text{state } x & : & \text{otherwise} \end{array}$$

With a larger number of low-power states, one can similarly use the energy computation above and solve the inequality for t_{pd} to determine the range of t_{pd} for which one state is better than another. Using all such boundary values of t_{pd} for all possible pairs of states, and the minimum t_{pd} that allows the use of each state, one can find a simple static mapping from t_{pd} to the power state that results in the lowest energy dissipation.

5 Evaluation

To evaluate the potential energy savings provided by the various sprint-and-halt scheduling algorithms described so far, one can use a system simulation to predict energy dissipation across a broad range of scenarios. The following subsection describes the simulator developed to evaluate the power-down scheduling techniques and the assumptions made in its design. Following this, some simulation results are presented to provide insight into the system parameters affecting energy savings in a real-time system with software-controlled power-down capabilities.

5.1 Simulation Methodology

The sprint-and-halt algorithms are evaluated using a simulator developed using C++ that models the operation of hardware capable of software-controlled power-down under a wide range of system characteristics. The simulator takes as input a task set, specified with the period and computation requirements of each task, as well as several system parameters, and provides the energy consumption of the system for each of the algorithms presented earlier. Real-time schedulers without any power-down support are also simulated for comparison. Parameters supplied to the simulator include the hardware specification, i.e., P_{high}, P_{low}, P_{trans}, t_{down}, and t_{up}, and a specification of the fraction of the WCET that the tasks should actually consume. This latter parameter can be a constant (e.g., 0.9 indicates that each task will use 90% of its specified worst-case computation cycles during each invocation), or can be a random function (e.g., uniformly-distributed random multiplier for each invocation).

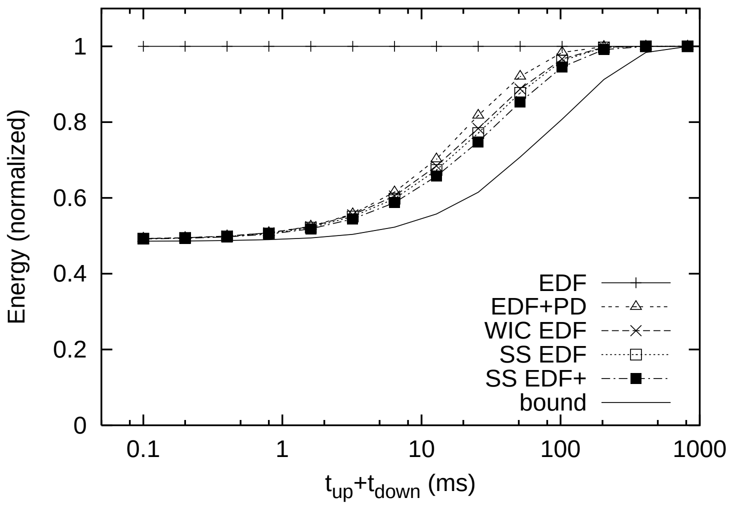
The simulation assumes the bimodal system power model described in Section 3. Each simulated cycle used for task execution or idle consumes a constant energy quantum, derived from P_{high}, while each cycle in the power-down state consumes energy based on P_{low}. Although the simulator can use arbitrary P_{trans} values, in the evaluations, the worst-case situation where $P_{trans} = P_{high}$ is assumed. With this model, variations due to different types of instructions executed are not taken into account. This simplification eliminates the need for actual execution traces, and a simpler cycle counting approach can be used to determine energy consumption. The simulator only considers the time/energy overheads of switching into and out of the power-down state. In particular, it does not consider preemption and task-switch overheads, or the overheads of executing scheduler code. However, these are small relative to the range of power-state switching latencies considered, so there is no loss of generality from these assumptions. Besides, the relative energy performance of different scheduling algorithms will not be affected by this assumption.

The real-time task sets are specified using a pair of numbers for each task, indicating its period and worst-case execution time. The task sets are generated randomly as follows. Each task has an equal probability of having a short (1–10 ms), medium (10–100 ms), or long (100–1000 ms) period. Within each range, task periods are uniformly distributed. This simulates the varied mix of short- and long- period tasks commonly found in real-time systems. The computation requirements of the tasks are assigned randomly using a similar 3-range uniform distribution. Finally, the task computation requirements are scaled by a constant chosen such that the sum of the utilizations of the tasks in the task set reaches a desired value. This method of generating real-time task sets has been used previously in the development and evaluation of a real-time embedded microkernel [16]. Averaged across hundreds of distinct task sets generated for several different total worst-case utilization values, the simulations provide a relationship of energy consumption to the worst-case utilization of the task sets, or to the power-down-power-up latencies.

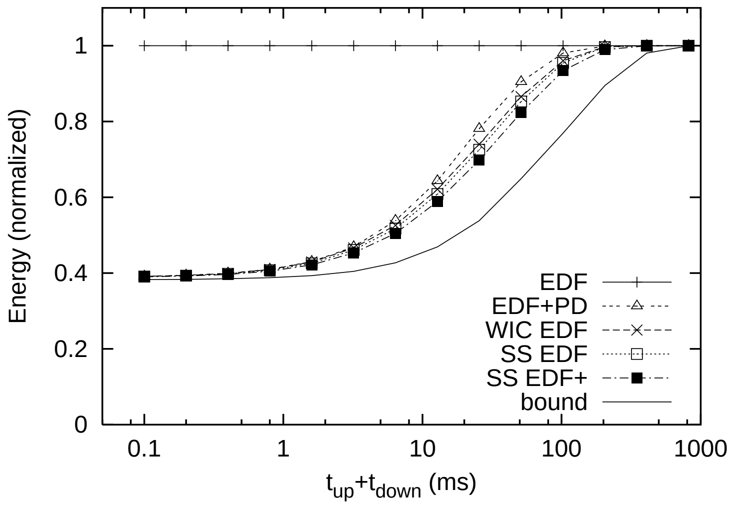
5.2 Results

The simulator described above permits the energy consumption comparison of the sprint-and-halt schedulers to each other as well as against real-time scheduling without power-down support. In addition, the schedulers are also compared to a theoretical lower bound on energy. This lower bound is computed based on the observation that the highest frequency task in the system limits the maximum duration of the power-down state. Considering just this one task, and assuming actual execution times are known, then it is possible to execute this task as late as possible so it completes exactly at its deadline, and then its next invocation is released and executed immediately. The system can power down until sufficiently before the following invocation's deadline to execute the task. As a result, at best, one power-down interval can span at most 2 periods of the highest frequency task. The lower bound energy is computed assuming all of the idle time is lumped together and divided into power-down intervals exactly 2 times the length of the period of the highest frequency task. All of the execution time is likewise lumped together at the start of the

(a)



(b)



(c)

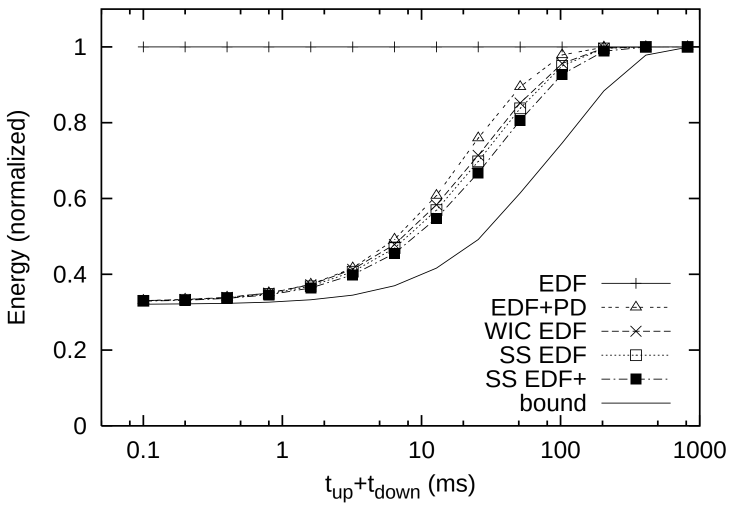


Figure 9: Effects of varying power-down hardware specifications: (a) $P_{high}/P_{low} = 4$; (b) $P_{high}/P_{low} = 10$; (c) $P_{high}/P_{low} = 100$

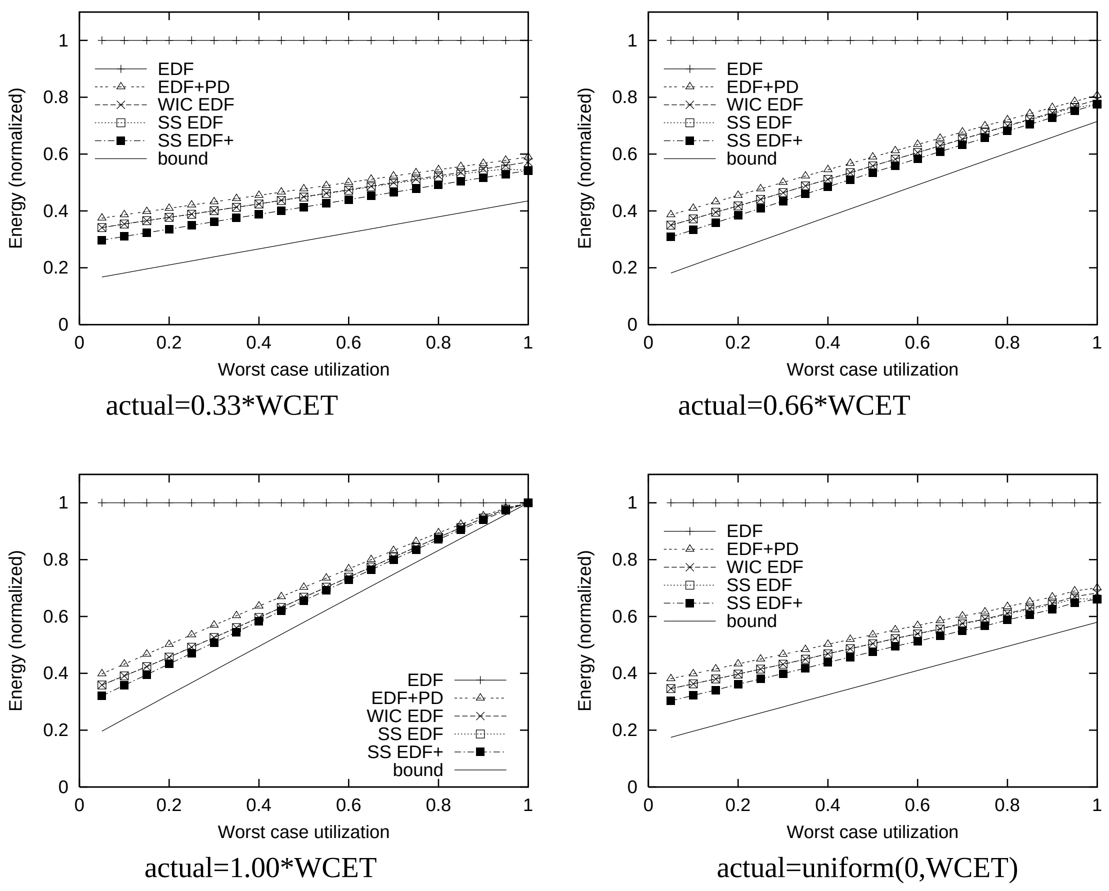


Figure 10: Effects of varying workload parameters

lower-bound simulation. This lower bound is in practice unachievable, but does give some insight into limitations on further energy improvements. We limit the experiments to the EDF versions of the sprint-and-halt algorithms.

Effects of power-down specifications: The first set of experiments determines the effects of varying the specifications of the power-down hardware. Figure 9 shows the energy dissipation for each sprint-and-halt EDF-based algorithm, normalized with respect to EDF scheduling without power-down support. Here, the task sets all contain 8 random tasks as described earlier, such that the total worst-case processor utilization is 0.95. The tasks' actual execution times are fixed to WCET/3. The average energy for the task sets is plotted for varying values of power-down latency, i.e., $t_{down} + t_{up}$, which are shown on a log scale. The three separate plots correspond to different P_{high}/P_{low} ratios.

One should immediately note that all of the algorithms can potentially save significant amounts of energy, particularly when the power-state transition latencies are small. In addition, the actual ratio of P_{high} to P_{low} does not affect the relative performance of the schedulers or the general trend of the curves significantly. Only the maximal achievable savings is affected. Finally, at the extreme range of power-state transistion latencies, the algorithms perfrom very close to the computed lower bound on energy. Between these extremes, the improved slack-stealing EDF (SS EDF+) scheduler

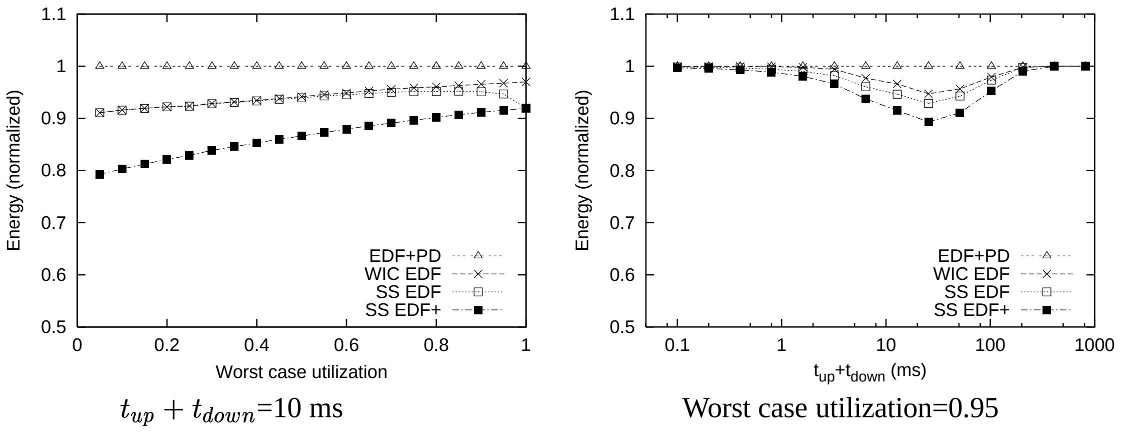


Figure 11: Relative performance of sprint-and-halt algorithms

performs the best, followed by slack-stealing EDF (SS EDF) and work-idle-conserving EDF (WIC EDF) schedulers. Even the simple EDF with power-down (EDF+PD) scheduler performs much better than plain vanilla EDF scheduling.

Effects of workload parameters: The next set of experiments fix the harware specification and vary instead the task set parameters. In Figure 10, P_{high}/P_{low} is set to 20, and $t_{down} + t_{up}$ set to 10 ms. There are still 8 tasks in each random task set, but now the worst-case utilization of the task sets is varied, and resulting average energy across the task sets plotted. The actual execution times for the task is also varied, set to 0.33, 0.66, and 1.0 times the WCET for the first three subplots. The fourth plot uses a unform random distribution for the actual execution times of each task invocation.

Overall, the average energy profiles of the schedulers across multiple random task sets seem to vary fairly linearly with the worst-case utilization of the task sets. The one interesting exception is the simple slack-stealing EDF (SS EDF) scheduler. For most of the range of utilization, it performs nearly identically to the work-idle-conserving (WIC EDF) scheduler. However, at very high worst-case utilizations, it performs better than WIC EDF. This is due to the fact that at high utilizations, WCET EDF schedules have very little idle time, so the algorithm, attempting to pace execution to the WCET schedule, is capable of much longer deferrals. It is this very effect that motivates the improved slack-stealing approach (SS EDF+).

The change in actual execution times affects the slope of the average energy response as worst-case utilization is varied. Using uniformly-distributed random execution times (between 0 and WCET for each task invocation) does not significantly change the average energy curves. As the average execution times are generally much smaller than the WCETs for most real-time task sets, the first plot is closest to what one can deem as typical. Here, with the reasonable assumption of 20:1 P_{high} to P_{low} ratio and 10 ms power-down latency, the sprint-and-halt schedulers achieve 40–70% energy reduction over EDF without power-down support.

Relative performance of power-down schedulers: Although all of the power-down schedulers perform much better than ordinary EDF without power-down support, it is interesting to see how well the more complex techniques perform relative to the simple EDF with power-down added. Figure 11 shows this relationship, assuming actual task execution times are 0.33*WCET. The first plot indicates that with a power-down latency of 10 ms, the improved slack-stealing approach reduces average energy by approximately 10–20% over EDF+PD as the task set worst case processor utilizations vary. In the second plot, the worst-case utilization is fixed to 0.95, and the power-down latency is varied across the log scale. At the extremes, is very little improvement over EDP+PD. However, in the middle of the range, where the power-down latency is comparable to the task periods, the more advanced techniques show up to 10% energy reduction relative to the simple EDF+PD scheduler.

6 Related Work

Reducing power consumption in mobile devices is a very active area of research. The authors of [2, 10] enumerate and survey a wide variety of approaches to energy reduction on mobile platforms from a high-level perspective. A variety of techniques for powering down subsystems, including display backlight, disk drives, and communication channels, and applying circuit tricks to reduce power are cited. However, little work has been done in the context of powering down systems (including the processor) when real-time constraints are present.

Although not intended for real-time systems, the authors of [7] developed techniques for powering down systems that execute event-driven (i.e., user-interactive) applications. Using idle history, future idle durations are predicted. This, unfortunately, cannot provide the timing guarantees needed for real-time systems. This work also proposes a *pre-wake* technique that reactivates the system early to compensate for wake-up latencies and improve responsiveness. A similar mechanism is used in all of the sprint-and-halt algorithms, as this is necessary in order to ensure deadline guarantees.

Focusing on powering down I/O subsystems, the authors of [11] attempted to maximize the effectiveness of power-down by increasing the duration and amortizing switching overheads. This is similar in concept to the sprint-and-halt algorithms, but it is meant for I/O, not the processor. Furthermore, it involves reordering tasks to coalesce common device accesses, which would affect timings of tasks and preclude its use in real-time systems.

Extending this to real-time systems, the authors of [14] presented a device power-scheduling algorithm that preserves real-time guarantees. Deadlines are preserved by keeping the task execution schedule unaltered and fitting power-down events for I/O devices whenever possible. In contrast, sprint-and-halt scheduling does alter the task execution schedule, while preserving deadlines, and can power down the processor.

Finally, there has been much research on dynamic voltage scaling of the CPU to conserve energy since the earliest papers on this topic appeared [3, 5, 15]. These mechanisms have also been extended to work in real-time systems [6, 12, 13]. DVS algorithms try to execute tasks as slowly as possible to spread out work and eliminate idle time, while in contrast, sprint-and-halt techniques try to coalesce work and execute it as fast as possible to allow longer power-down intervals. Hence, these real-time scheduling algorithms approach energy conservation with directly opposite philosophies.

7 Conclusions

This paper has presented a class of sprint-and-halt scheduling algorithms that attempt to make best use of software-controlled power-down to reduce energy expenditure, while meeting hard, real-time constraints. Several algorithms of increasing complexity have been developed to better amortize energy costs due to the transition latencies to and from low-power states. Extensive simulations show that with some typical system parameters, the power-down techniques can save 40–70% of the energy dissipated in an unmodified system, while preserving all real-time deadline guarantees. Sensitivity experiments show that for very large (100's of ms) and very small (100's of μs) power-down latencies, the simplest power-down scheduling techniques suffice, as all of the methods approach a theoretical lower bound. However, for moderate power-down latencies, the advanced techniques provide 10–20% lower energy consumption relative to the simplest sprint-and-halt schedulers.

Future research directions related to this work include extending sprint-and-halt scheduling to less restrictive real-time paradigms. A probabilistic real-time approach may provide greater flexibility in using power-down techniques and allow greater energy savings. Integration of sprint-and-halt with other power-reduction techniques, such as dynamic voltage scaling, may also be possible in a hybrid solution that switches between the two depending on workload characteristics and available idle time.

References

[1] ADVANCED CONFIGURATION AND POWER INTERFACE. http://www.acpi.info/.

[2] BENINI, L., BOGLIOLO, A., AND MICHELI, G. D. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VSLI* (June 2000), 299–316.

[3] BURD, T. D., AND BRODERSEN, R. W. Energy efficient CMOS microprocessor design. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture* (Los Alamitos, CA, USA, Jan. 1995), T. N. Mudge and B. D. Shriver, Eds., IEEE Computer Society Press, pp. 288–297.

[4] DEY, J. K., TOWSLEY, D. F., KRISHNA, C. M., AND GIRKAR, M. Efficient on-line processor scheduling for a class of iris real-time tasks. In *SIGMETRICS* (1993), pp. 217–228.

[5] GOVIL, K., CHAN, E., AND WASSERMANN, H. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95* (Mar. 1995).

[6] GRUIAN, F. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01* (Huntington Beach, CA, Aug. 2001).

[7] HWANG, C., AND WU, A. C.-H. A predictive system shut-down method for energy saving of event-driven computation. In *Proc. Intl. Conf. on Computer-Aided Design* (1997), pp. 28–32.

[8] LEHOCZKY, J., AND THUEL, S. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *Proceedings of the IEEE Real-Time Systems Symposium* (1994).

[9] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM 20*, 1 (Jan. 1973), 46–61.

[10] LORCH, J. R., AND SMITH, A. J. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine 5*, 3 (June 1998), 60–73.

[11] LU, Y.-H., BENINI, L., AND MICHELI, G. D. Low-power task scheduling for multiple devices. In *International Workshop on Hardware/Software Codesign* (May 2000), pp. 39–43.

[12] PERING, T., AND BRODERSEN, R. Energy efficient voltage scheduling for real-time operating systems. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS'98, Work in Progress Session* (Denver, CO, June 1998).

[13] PILLAI, P., AND SHIN, K. G. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Alberta, CA, Oct. 2001), pp. 89–102.

[14] SWAMINATHAN, V., CHAKRABARTY, K., AND IYENGAR, S. S. Dynamic i/o power management for hard real-time systems. In *Proc. Intl. Symposium on Hardware/Software Co-Design (CODES)* (2001), pp. 237–242.

[15] WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, Nov. 1994), pp. 13–23.

[16] ZUBERI, K. M., PILLAI, P., AND SHIN, K. G. EMERALDS: A small-memory real-time microkernel. In *Proceedings of the 17th ACM Symposium on Operating System Principles* (Kiawah Island, SC, Dec. 1999), ACM Press, pp. 277–291.