Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems*

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

In recent years, there has been a rapid and wide spread of non-traditional computing platforms, especially mobile and portable computing devices. As applications become increasingly sophisticated and processing power increases, the most serious limitation on these devices is the available battery life. Dynamic Voltage Scaling (DVS) has been a key technique in exploiting the hardware characteristics of processors to reduce energy dissipation by lowering the supply voltage and operating frequency. The DVS algorithms are shown to be able to make dramatic energy savings while providing the necessary peak computation power in general-purpose systems. However, for a large class of applications in embedded real-time systems like cellular phones and camcorders, the variable operating frequency interferes with their deadline guarantee mechanisms, and DVS in this context, despite its growing importance, is largely overlooked/under-developed. To provide real-time guarantees, DVS must consider deadlines and periodicity of real-time tasks, requiring integration with the real-time scheduler. In this paper, we present a class of novel algorithms called *real-time DVS* (RT-DVS) that modify the OS's real-time scheduler and task management service to provide significant energy savings while maintaining real-time deadline guarantees. We show through simulations and a working prototype implementation that these RT-DVS algorithms closely approach the theoretical lower bound on energy consumption, and can easily reduce energy consumption 20% to 40% in an embedded real-time system.

1. INTRODUCTION

Computation and communication have been steadily moving toward mobile and portable platforms/devices. This is very evident in the growth of laptop computers and PDAs, but is also occurring in the embedded world. With continued miniaturization and increasing computation power, we see ever growing use of powerful microprocessors running sophisticated, intelligent control software in a vast array of devices including digital camcorders, cellular phones, and portable medical devices.

Unfortunately, there is an inherent conflict in the design goals behind these devices: as mobile systems, they should be designed to maximize battery life, but as intelligent devices, they need powerful processors, which consume more energy than those in simpler devices, thus reducing battery life. In spite of continuous advances in semiconductor and battery technologies that allow microprocessors to provide much greater computation per unit of energy and longer total battery life, the fundamental tradeoff between performance and battery life remains critically important.

Recently, significant research and development efforts have been made on *Dynamic Voltage Scaling* (DVS) [2, 4, 7, 8, 12, 19, 21, 22, 23, 24, 25, 26, 28, 30]. DVS tries to address the tradeoff between performance and battery life by taking into account two important characteristics of most current computer systems: (1) the peak computing rate needed is much higher than the average throughput that must be sustained; and (2) the processors are based on CMOS logic. The first characteristic effectively means that high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance, low-power processor would suffice. We can achieve the low performance by simply lowering the operating frequency of the processor when the full speed is not needed. DVS goes beyond this and scales the operating voltage of the processor along with the frequency. This is possible because static CMOS logic, used in the vast majority of microprocessors today, has a voltage-dependent maximum operating frequency, so when used at a reduced frequency, the processor can operate at a lower supply voltage. Since the energy dissipated per cycle with CMOS circuitry scales quadratically to the supply voltage ($E \propto V^2$) [2], DVS can potentially provide a very large net energy savings through frequency and voltage scaling.

In time-constrained applications, often found in embedded systems like cellular phones and digital video cameras, DVS presents a serious problem. In these real-time embedded systems, one cannot directly apply most DVS algorithms known to date, since changing the operating frequency of the processor will affect the execution time of the tasks and may violate some of the timeliness guarantees. In this paper, we present several novel algorithms that incorporate DVS into the OS scheduler and task management services of a real-time embedded system, providing the energy savings of DVS while preserving deadline guarantees. This is in sharp

*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.

This paper will appear in SOSP'01

contrast with the average throughput-based mechanisms typical of many current DVS algorithms. In addition to detailed simulations that show the energy-conserving benefits of our algorithms, we also present an actual implementation of our mechanisms, demonstrating them with measurements on a working system. To the best of our knowledge, this is one of the first working implementations of DVS, and the first implementation of *Real-Time DVS* (RT-DVS).

In the next section, we present details of DVS, real-time scheduling, and our new RT-DVS algorithms. Section 3 presents the simulation results and provides insight into the system parameters that most influence the energy-savings potential of RT-DVS. Section 4 describes our implementation of RT-DVS mechanisms in a working system and some measurements obtained. Section 5 presents related work and puts our work in a larger perspective before we close with our conclusions and future directions in Section 6.

2. RT-DVS

To provide energy-saving DVS capability in a system requiring real-time deadline guarantees, we have developed a class of RT-DVS algorithms. In this section, we first consider DVS in general, and then discuss the restrictions imposed in embedded real-time systems. We then present RT-DVS algorithms that we have developed for this time-constrained environment.

2.1 Why DVS?

Power requirements are one of the most critical constraints in mobile computing applications, limiting devices through restricted power dissipation, shortened battery life, or increased size and weight. The design of portable or mobile computing devices involves a tradeoff between these characteristics. For example, given a fixed size or weight for a handheld computation device/platform, one could design a system using a low-speed, low-power processor that provides long battery life, but poor performance, or a system with a (literally) more powerful processor that can handle all computational loads, but requires frequent battery recharging. This simply reflects the cost of increasing performance — for a given technology, the faster the processor, the higher the energy costs (both overall *and* per unit of computation).

The discussion in this paper will generally focus on the energy consumption of the processor in a portable computation device for two main reasons. First, the practical size and weight of the device are generally fixed, so for a given battery technology, the available energy is also fixed. This means that only power consumption affects the battery life of the device. Secondly, we focus particularly on the processor because in most applications, the processor is the most energy-consuming component of the system. This is definitely true on small handheld devices like PDAs [3], which have very few components, but also on large laptop computers [20] that have many components including large displays with backlighting. Table 1 shows measured power consumption of a typical laptop computer. When it is idle, the display backlighting accounts for a large fraction of dissipated power, but at maximum computational load, the processor subsystem dominates, accounting for nearly 60% of the energy consumed. As a result, the design problem generally boils down to a tradeoff between the computational power of the processor and the system's battery life.

One can avoid this problem by taking advantage of a feature very common in most computing applications: the average computational throughput is often much lower than the peak computational capacity needed for adequate performance. Ideally, the processor would be "sized" to meet the average computational demands, and would have low energy costs per unit of computation, thus providing good battery life. During the (relatively rare) times when peak computational load is imposed, the higher computational throughput of a more sophisticated processor would somehow be "configured" to meet the high performance requirement, but at a higher energy cost per unit of computation. Since the high-cost cycles are applied for only some, rather than all, of the computation, the energy consumption will be lower than if the more powerful processor were used all of the time, but the performance requirements are still met.

Screen	CPU subsystem	Disk	Power
On	Idle	Spinning	13.5 W
On	Idle	Standby	13.0 W
Off	Idle	Standby	7.1 W
Off	Max. Load	Standby	27.3 W

Table 1: Power consumption measured on Hewlett-Packard N3350 laptop computer

One promising mechanism that provides the best of both low-power and high-performance processors in the same system is DVS [30]. DVS relies on special hardware, in particular, a programmable DC-DC switching voltage regulator, a programmable clock generator, and a high-performance processor with wide operating ranges, to provide this best-of-both-worlds capability. In order to meet peak computational loads, the processor is operated at its normal voltage and frequency (which is also its maximum frequency). When the load is lower, the operating frequency is reduced to meet the computational requirements. In CMOS technology, used in virtually all microprocessors today, the maximum operating frequency increases (within certain limits) with increased operating voltage, so when the processor is run slower, a reduced operating voltage suffices [2]. A second important characteristic is that the energy consumed by the processor per clock cycle scales quadratically with the operating voltage ($E \propto V^2$) [2], so even a small change in voltage can have a significant impact on energy consumption. By dynamically scaling both voltage and frequency of the processor based on computation load, DVS can provide the performance to meet peak computational demands, while on average, providing the reduced power consumption (including energy per unit computation) benefits typically available on low-performance processors.

2.2 Real-time issues

For time-critical applications, however, the scaling of processor frequency could be detrimental. Particularly in real-time embedded systems like portable medical devices and cellular phones, where tasks must be completed by some specified deadlines, most algorithms for DVS known to date cannot be applied. These DVS algorithms do not consider real-time constraints and are based on solely average computational throughput [7, 23, 30]. Typically, they use a simple feedback mechanism, such as detecting the amount of idle time on the processor over a period of time, and then adjust the frequency and voltage to just handle the computational load. This is very simple and follows the load characteristics closely, but cannot provide any timeliness guarantees and tasks may miss their execution deadlines. As an example, in an embedded camcorder controller, suppose there is a program that must react to a change in a sensor reading within a 5 ms deadline, and that it requires up to 3 ms of computation time with the processor running at the maximum operating frequency. With a DVS algorithm that reacts only

EDF_test (α):
    if ($C_1/P_1 + \cdots + C_n/P_n \leq \alpha$) return true;
    else return false;

RM_test (α):
    if ($\forall T_i \in \{T_1, \ldots, T_n | P_1 \leq \cdots \leq P_n\}$
        $\lceil P_i/P_1 \rceil * C_1 + \cdots + \lceil P_i/P_i \rceil * C_i \leq \alpha * P_i$)
    return true;
    else return false;

select_frequency:
    use lowest frequency $f_i \in \{f_1, \ldots, f_m | f_1 < \cdots < f_m\}$
    such that RM_test(f_i/f_m) or EDF_test(f_i/f_m) is true.

Figure 1: Static voltage scaling algorithm for EDF and RM schedulers

to average throughput, if the total load on the system is low, the processor would be set to operate at a low frequency, say half of the maximum, and the task, now requiring 6 ms of processor time, cannot meet its 5 ms deadline. In general, none of the average throughput-based DVS algorithms found in literature can provide real-time deadline guarantees.

In order to realize the reduced energy-consumption benefits of DVS in a real-time embedded system, we need new DVS algorithms that are tightly-coupled with the actual real-time scheduler of the operating system. In the classic model of a real-time system, there is a set of tasks that need to be executed periodically. Each task, T_i, has an associated period, P_i, and a worst-case computation time, C_i.[1]. The task is *released* (put in a runnable state) periodically once every P_i time units (actual units can be seconds or processor cycles or any other meaningful quanta), and it can begin execution. The task needs to complete its execution by its deadline, typically defined as the end of the period [18], i.e., by the next release of the task. As long as each task T_i uses no more than C_i cycles in each invocation, a real-time scheduler can guarantee that the tasks will always receive enough processor cycles to complete each invocation in time. Of course, to provide such guarantees, there are some conditions placed on allowed task sets, often expressed in the form of *schedulability tests*. A real-time scheduler guarantees that tasks will meet their deadlines given that:

C1. the task set is *schedulable* (passes schedulability test), and

C2. no task exceeds its specified worst-case computation bound.

DVS, when applied in a real-time system, must ensure that both of these conditions hold.

In this paper, we develop algorithms to integrate DVS mechanisms into the two most-studied real-time schedulers, *Rate Monotonic* (RM) and *Earliest-Deadline-First* (EDF) schedulers [13, 14, 17, 18, 27]. RM is a static priority scheduler, and assigns task priority according to period — it always selects first the task with the shortest period that is ready to run (released for execution). EDF is a dynamic priority scheduler that sorts tasks by deadlines and always gives the highest priority to the released task with the most imminent deadline. In the classical treatments of these schedulers [18], both assume that the task deadline equals the period (i.e., the task must complete before its next invocation), that scheduling and preemption overheads are negligible,[2] and that the tasks are independent (no task will block waiting for another task). In our design of DVS to real-time systems, we maintain the same assumptions, since our primary goal is to reduce energy consumption, rather than to derive general scheduling mechanisms.

[1]Although not explicit in the model, aperiodic and sporadic tasks can be handled by a periodic or deferred server [16] For non-real-time tasks, too, we can provision processor time using a similar periodic server approach.

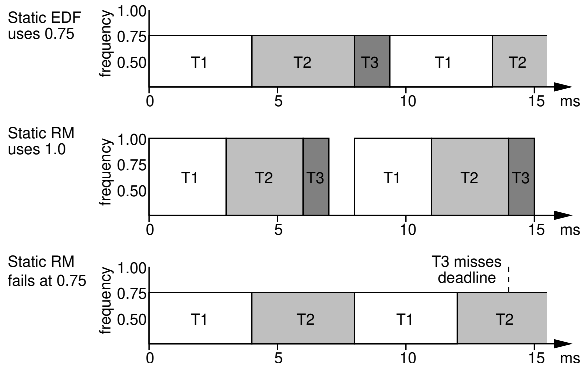

Figure 2: Static voltage scaling example

Task	Computing Time	Period
1	3 ms	8 ms
2	3 ms	10 ms
3	1 ms	14 ms

Table 2: Example task set, where computing times are specified at the maximum processor frequency

In the rest of this section, we present our algorithms that perform DVS in time-constrained systems without compromising deadline guarantees of real-time schedulers.

2.3 Static voltage scaling

We first propose a very simple mechanism for providing voltage scaling while maintaining real-time schedulability. In this mechanism we select the lowest possible operating frequency that will allow the RM or EDF scheduler to meet all the deadlines for a given task set. This frequency is set statically, and will not be changed unless the task set is changed.

To select the appropriate frequency, we first observe that scaling the operating frequency by a factor α ($0 < \alpha \leq 1$) effectively results in the worst-case computation time needed by a task to be scaled by a factor $1/\alpha$, while the desired period (and deadline) remains unaffected. We can take the well-known schedulability tests for EDF and RM schedulers from the real-time systems literature, and by using the scaled values for worst-case computation needs of the tasks, can test for schedulability at a particular frequency. The necessary and sufficient schedulability test for a task set under ideal EDF scheduling requires that the sum of the worst-case *utilizations* (computation time divided by period) be less than one, i.e., $C_1/P_1 + \cdots + C_n/P_n \leq 1$ [18]. Using the scaled computation time values, we obtain the EDF schedulability test with frequency

[2]We note that one could account for preemption overheads by computing the worst-case preemption sequences for each task and adding this overhead to its worst-case computation time.

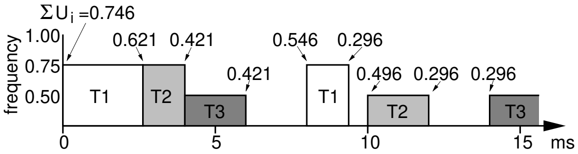

Figure 3: Example of cycle-conserving EDF

Task	Invocation 1	Invocation 2
1	2 ms	1 ms
2	1 ms	1 ms
3	1 ms	1 ms

Table 3: Actual computation requirements of the example task set (assuming execution at max. frequency)

```
select_frequency():
    use lowest freq. f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
    such that U_1 + ··· + U_n ≤ f_i/f_m

upon task_release(T_i):
    set U_i to C_i/P_i;
    select_frequency();

upon task_completion(T_i):
    set U_i to cc_i/P_i;
        /* cc_i is the actual cycles used this invocation */
    select_frequency();
```

Figure 4: Cycle-conserving DVS for EDF schedulers

scaling factor α:

$$C_1/P_1 + \cdots + C_n/P_n \leq \alpha$$

Similarly, we start with the sufficient (but not necessary) condition for schedulability under RM scheduling [13] and obtain the test for a scaled frequency (see Figure 1). The operating frequency selected is the lowest one for which the modified schedulability test succeeds. The voltage, of course, is changed to match the operating frequency. Assume that the operating frequencies and the corresponding voltage settings available on the particular hardware platform are specified in a table provided to the software. Figure 1 summarizes the static voltage scaling for EDF and RM scheduling, where there are m operating frequencies $f_1, \ldots, f_m$ such that $f_1 < f_2 < \cdots < f_m$.

Figure 2 illustrates these mechanisms, showing sample worst-case execution traces under statically-scaled EDF and RM scheduling. The example uses the task set in Table 2, which indicates each task's period and worst-case computation time, and assumes that three normalized, discrete frequencies are available (0.5, 0.75, and 1.0). The figure also illustrates the difference between EDF and RM (i.e., deadline vs. rate for priority), and shows that statically-scaled RM cannot reduce frequency (and therefore reduce voltage and conserve energy) as aggressively as the EDF version.

As long as for some available frequency, the task set passes the schedulability test, and as long as the tasks use no more than their scaled computation time, this simple mechanism will ensure that frequency and voltage scaling will not compromise timely execution of tasks by their deadlines. The frequency and voltage setting selected are static with respect to a particular task set, and are changed only when the task set itself changes. As a result, this mechanism need not be tightly-coupled with the task management functions of the real-time operating system, simplifying implementation. On the other hand, this algorithm may not realize the full potential of energy savings through frequency and voltage scaling. In particular, the static voltage scaling algorithm does not deal with situations where a task uses less than its worst-case requirement of processor cycles, which is usually the case. To deal with this common situation, we need more sophisticated, RT-DVS mechanisms.

2.4 Cycle-conserving RT-DVS

Although real-time tasks are specified with worst-case computation requirements, they generally use much less than the worst case on most invocations. To take best advantage of this, a DVS mechanism could reduce the operating frequency and voltage when tasks use less than their worst-case time allotment, and increase frequency to meet the worst-case needs. When a task is released for its next invocation, we cannot know how much computation it will actually require, so we must make the conservative assumption that it will need its specified worst-case processor time. When the task completes, we can compare the actual processor cycles used to the worst-case specification. Any unused cycles that were allotted to the task would normally (or eventually) be wasted, idling the processor. Instead of idling for extra processor cycles, we can devise DVS algorithms that avoid wasting cycles (hence "cycle-conserving") by reducing the operating frequency. This is somewhat similar to slack time stealing [15], except surplus time is used to run other remaining tasks at a lower CPU frequency rather than accomplish more work. These algorithms are tightly-coupled with the operating system's task management services, since they may need to reduce frequency on each task completion, and increase frequency on each task release. The main challenge in designing such algorithms is to ensure that deadline guarantees are not violated when the operating frequencies are reduced.

For EDF scheduling, as mentioned earlier, we have a very simple schedulability test: as long as the sum of the worst-case task utilizations is less than α, the task set is schedulable when operating at the maximum frequency scaled by factor α. If a task completes earlier than its worst-case computation time, we can reclaim the excess time by recomputing utilization using the actual computing time consumed by the task. This reduced value is used until the task is released again for its next invocation. We illustrate this in Figure 3, using the same task set and available frequencies as before, but using actual execution times from Table 3. Here, each invocation of the tasks may use less than the specified worst-case times, but the actual value cannot be known to the system until after the task completes execution. Therefore, at each scheduling point (task release or completion) the utilization is recomputed using the actual time for completed tasks and the specified worst case for the others, and the frequency is set appropriately. The numerical values in the figure show the total task utilizations computed using the information available at each point.

The algorithm itself (Figure 4) is simple and works as follows. Suppose a task T_i completes its current invocation after using cc_i cycles which are usually much smaller than its worst-case computation time C_i. Since task T_i uses no more than cc_i cycles in its current invocation, we treat the task as if its worst-case computation bound were cc_i. With the reduced utilization specified for this task, we can now potentially find a smaller scaling factor α (i.e., lower operating frequency) for which the task set remains schedulable. Trivially,

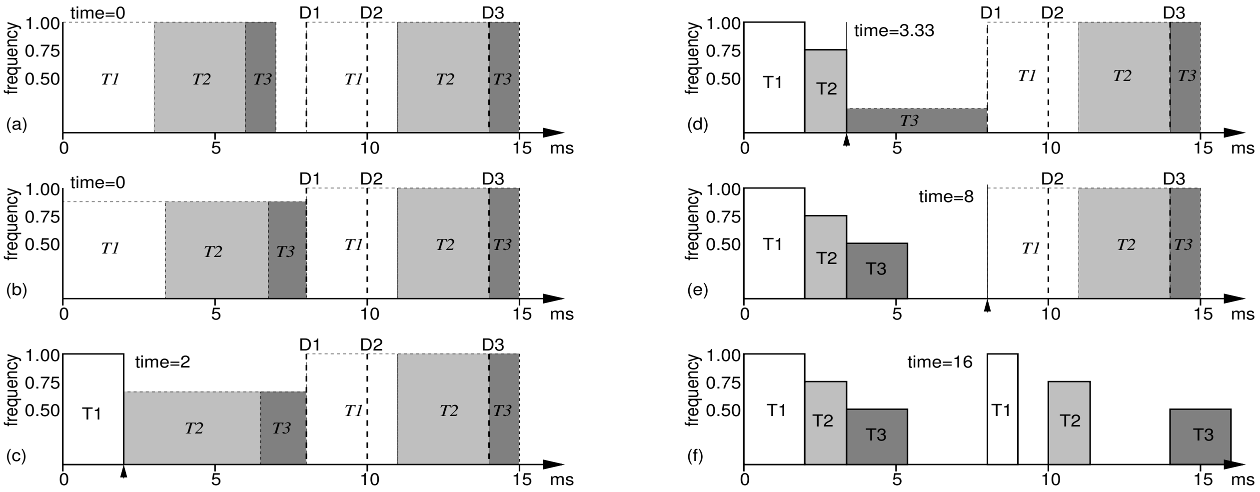

Figure 5: Example of cycle-conserving RM: (a) Initially use statically-scaled, worst-case RM schedule as target; (b) Determine minimum frequency so as to complete the same work by D1; rounding up to the closest discrete setting requires frequency 1.0; (c) After T1 completes (early), recompute the required frequency as 0.75; (d) Once T2 completes, a very low frequency (0.5) suffices to complete the remaining work by D1; (e) T1 is re-released, and now, try to match the work that should be done by D2; (f) Execution trace through time 16 ms.

---

assume f_j is frequency set by static scaling algorithm

select_frequency():
  set s_m = max_cycles_until_next_deadline();
  use lowest freq. $f_i \in \{f_1, \ldots, f_m | f_1 < \cdots < f_m\}$
  such that $(d_1 + \cdots + d_n)/s_m \leq f_i/f_m$

upon task_release(T_i):
  set *c_left*$_i$ = C_i;
  set s_m = max_cycles_until_next_deadline();
  set s_j = $s_m * f_j/f_m$;
  allocate_cycles (s_j);
  select_frequency();

upon task_completion(T_i):
  set *c_left*$_i$ = 0;
  set d_i = 0;
  select_frequency();

during task_execution(T_i):
  decrement *c_left*$_i$ and d_i;

allocate_cycles(k):
  for i = 1 to n, $T_i \in \{T_1, \ldots, T_n | P_1 \leq \cdots \leq P_n\}$
      /* tasks sorted by period */
    if (*c_left*$_i$ $< k$)
      set d_i = *c_left*$_i$;
      set k = k - *c_left*$_i$;
    else
      set d_i = k;
      set k = 0;

Figure 6: Cycle-conserving DVS for RM schedulers

---

given that the task set prior to this change was schedulable, the EDF schedulability test will continue to hold, and T_i (which has completed execution) will not violate its lowered maximum computing bound for the remainder of time until its deadline. Therefore, the task set continues to meet both conditions C1 and C2 imposed by the real-time scheduler to guarantee timely execution, and as a result, deadline guarantees provided by EDF scheduling will continue to hold *at least* until T_i is released for its next invocation. At this point, we must restore its computation bound to C_i to ensure that it will not violate the temporarily-lowered bound and compromise the deadline guarantees. At this time, it may be necessary to increase the operating frequency. At first glance, this algorithm does not appear to significantly reduce frequencies, voltages, and energy expenditure. However, since multiple tasks may be simultaneously in the reduced-utilization state, the total savings can be significant.

We could use the same schedulability-test-based approach to designing a cycle-conserving DVS algorithm for RM scheduling, but as the RM schedulability test is significantly more complex ($O(n^2)$, where n is the number of tasks to be scheduled), we will take a different approach here. We observe that even assuming tasks always require their worst-case computation times, the statically-scaled RM mechanism discussed earlier can maintain real-time deadline guarantees. We assert that as long as equal or better progress for all tasks is made here than in the worst case under the statically-scaled RM algorithm, deadlines can be met here as well, regardless of the actual operating frequencies. We will also try to avoid getting ahead of the worst-case execution pattern; this way, any reduction in the execution cycles used by the tasks can be applied to reducing operating frequency and voltage. Using the same example as before, Figure 5 illustrates how this can be accomplished. We initially start with worst-case schedule based on static-scaling (a), which for this example, uses the maximum CPU frequency. To keep things simple, we do not look beyond the next deadline in the system. We then try to spread out the work that should be accomplished before this deadline over the entire interval from the current time to the deadline (b). This provides a minimum operating frequency value, but since the frequency settings are discrete, we round up to the closest available setting, frequency=1.0. After executing T_1, we

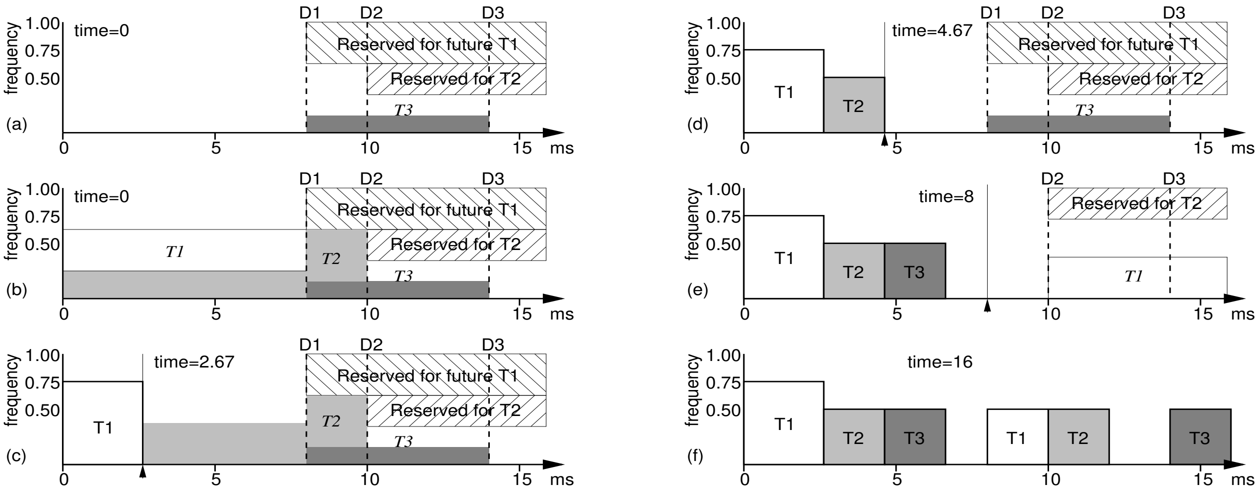

Figure 7: Example of look-ahead EDF: (a) At time 0, plan to defer T3's execution until after D1 (but by its deadline D3, and likewise, try to fit T2 between D1 and D2; (b) T1 and the portion of T2 that did not fit must execute before D1, requiring use of frequency 0.75; (c) After T1 completes, repeat calculations to find the new frequency setting, 0.5; (d) Repeating the calculation after T2 completes indicates that we do not need to execute anything by D1, but EDF is work-conserving, so T3 executes at the minimum frequency; (e) This occurs again when T1's next invocation is released; (f) Execution trace through time 16 ms.

repeat the exercise of spreading out the remaining work over the remaining time until the next deadline (c), which results in a lower operating frequency since T_1 completed earlier than its worst-case specified computing time. Repeating this at each scheduling point results in the final execution trace (f).

Although conceptually simple, the actual algorithm (Figure 6) for this is somewhat complex due to a number of counters that must be maintained. In this algorithm, we need to keep track of the worst-case remaining cycles of computation, c_left_i, for each task T_i. When task T_i is released, c_left_i is set to C_i. We then determine the progress that the static voltage scaling RM mechanism would make in the worst case by the earliest deadline for *any* task in the system. We obtain s_j and s_m, the number of cycles to this next deadline, assuming operation at the statically-scaled and the maximum frequencies, respectively. The s_j cycles are allocated to the tasks according to RM priority order, with each task T_i receiving an allocation $d_i \leq c\_left_i$ corresponding to the number of cycles that it would execute under the statically-scaled RM scenario over this interval. As long as we execute at least d_i cycles for each task T_i (or if T_i completes) by the next task deadline, we are keeping pace with the worst-case scenario, so we set execution speed using the sum of the d values. As tasks execute, their c_left and d values are decremented. When a task T_i completes, c_left_i and d_i are both set to 0, and the frequency and voltage are changed. Because we use this pacing criteria to select the operating frequency, this algorithm guarantees that at any task deadline, all tasks that would have completed execution in the worst-case statically-scaled RM schedule would also have completed here, hence meeting all deadlines.

These algorithms dynamically adjust frequency and voltage, reacting to the actual computational requirements of the real-time tasks. At most, they require 2 frequency/voltage switches per task per invocation (once each at release and completion), so any overheads for hardware voltage change can be accounted in the worst-case computation time allocations of the tasks. As we will see later, the algorithms can achieve significant energy savings without affecting real-time guarantees.

2.5 Look-Ahead RT-DVS

The final (and most aggressive) RT-DVS algorithm that we introduce in this paper attempts to achieve even better energy savings using a look-ahead technique to determine future computation need and defer task execution. The cycle-conserving approaches discussed above assume the worst case initially and execute at a high frequency until some tasks complete, and only then reduce operating frequency and voltage. In contrast, the look-ahead scheme tries to defer as much work as possible, and sets the operating frequency to meet the minimum work that must be done now to ensure all future deadlines are met. Of course, this may require that we will be forced to run at high frequencies later in order to complete all of the deferred work in time. On the other hand, if tasks tend to use much less than their worst-case computing time allocations, the peak execution rates for deferred work may never be needed, and this heuristic will allow the system to continue operating at a low frequency and voltage while completing all tasks by their deadlines.

Continuing with the example used earlier, we illustrate how a look-ahead RT-DVS EDF algorithm works in Figure 7. The goal is to defer work beyond the earliest deadline in the system (D_1) so that we can operate at a low frequency now. We allocate time in the schedule for the worst-case execution of each task, starting with the task with the latest deadline, T_3. We spread out T_3's work between D_1 and its own deadline, D_3, subject to a constraint reserving capacity for future invocations of the other tasks (a). We repeat this step for T_2, which cannot entirely fit between D_1 and D_2 after allocating T_3 and reserving capacity for future invocations of T_1. Additional work for T_2 and all of T_1 are allotted before D_1 (b). We note that more of T_2 could be deferred beyond D_1 if we moved all of T_3 after D_2, but for simplicity, this is not considered. We use the work allocated before D_1 to determine the operating frequency. Once T_1 has completed, using less than its specified worst-case execution cycles, we repeat this and find a lower operating frequency (c). Continuing this method of trying to defer work beyond the next deadline in the system ultimately results in the execution trace shown in (f).

```
select_frequency(x):
    use lowest freq. f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
    such that x ≤ f_i/f_m

upon task_release(T_i):
    set c_left_i = C_i;
    defer();

upon task_completion(T_i):
    set c_left_i = 0;
    defer();

during task_execution(T_i):
    decrement c_left_i;

defer():
    set U = C_1/P_1 + ··· + C_n/P_n;
    set s = 0;
    for i = 1 to n,   T_i ∈ {T_1, ..., T_n | D_1 ≥ ··· ≥ D_n}
                    /* Note: reverse EDF order of tasks */
        set U = U − C_i/P_i;
        set x = max(0, c_left_i − (1 − U)(D_i − D_n));
        set U = U + (c_left_i − x)/(D_i − D_n);
        set s = s + x;
    select_frequency (s/(D_n − current_time));
```

Figure 8: Look-Ahead DVS for EDF schedulers

The actual algorithm for look-ahead RT-DVS with EDF scheduling is shown in Figure 8. As in the cycle-conserving RT-DVS algorithm for RM, we keep track of the worst-case remaining computation c_left_i for the current invocation of task T_i. This is set to C_i on task release, decremented as the task executes, and set to 0 on completion. The major step in this algorithm is the deferral function. Here, we look at the interval until the next task deadline, try to push as much work as we can beyond the deadline, and compute the minimum number of cycles, s, that we must execute during this interval in order to meet all future deadlines. The operating frequency is set just fast enough to execute s cycles over this interval. To calculate s, we look at the tasks in reverse EDF order (i.e., latest deadline first). Assuming worst-case utilization by tasks with earlier deadlines (effectively reserving time for their future invocations), we calculate the minimum number of cycles, x, that the task must execute before the closest deadline, D_n, in order for it to complete by its own deadline. A cumulative utilization U is adjusted to reflect the actual utilization of the task for the time after D_n. This calculation is repeated for all of the tasks, using assumed worst-case utilization values for earlier-deadline tasks and the computed values for the later-deadline ones. s is simply the sum of the x values calculated for all of the tasks, and therefore reflects the total number of cycles that must execute by D_n in order for all tasks to meet their deadlines. Although this algorithm very aggressively reduces processor frequency and voltage, it ensures that there are sufficient cycles available for each task to meet its deadline after reserving worst-case requirements for higher-priority (earlier deadline) tasks.

2.6 Summary of RT-DVS algorithms

All of the RT-DVS algorithms we presented thus far should be fairly easy to incorporate into a real-time operating system, and do not require significant processing costs. The dynamic schemes all require $O(n)$ computation (assuming the scheduler provides an EDF sorted task list), and should not require significant processing over the scheduler. The most significant overheads may come from the hardware voltage switching times. However, in all of our algorithms, no more than two switches can occur per task per invocation period, so these overheads can easily be accounted for, and added to, the worst-case task computation times.

RT-DVS method	energy used
none (plain EDF)	1.0
statically-scaled RM	1.0
statically-scaled EDF	0.64
cycle-conserving EDF	0.52
cycle-conserving RM	0.71
look-ahead EDF	0.44

Table 4: Normalized energy consumption for the example traces

To conclude our series of examples, Table 4 shows the normalized energy dissipated in the example task (Table 2) set for the first 16 ms, using the actual execution times from Table 3. We assume that the 0.5, 0.75 and 1.0 frequency settings need 3, 4, and 5 volts, respectively, and that idle cycles consume no energy. More general evaluation of our algorithms will be done in the next section.

3. SIMULATIONS

We have developed a simulator to evaluate the potential energy savings from voltage scaling in a real-time scheduled system. The following subsection describes our simulator and the assumptions made in its design. Later, we show some simulation results and provide insight into the most significant system parameters affecting RT-DVS energy savings.

3.1 Simulation Methodology

Using C++, we developed a simulator for the operation of hardware capable of voltage and frequency scaling with real-time scheduling. 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 we have developed. EDF and RM schedulers without any DVS support are also simulated for comparison.[3] Parameters supplied to the simulator include the machine specification (a list of the frequencies and corresponding voltages available on the simulated platform) and a specification for the actual fraction of the worst-case execution cycles that the tasks will require for each invocation. 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 that a constant amount of energy is required for each cycle of operation at a given voltage. This quantum is scaled by the square of the operating voltage, consistent with energy dissipation in CMOS circuits ($E \propto V^2$). Only the energy consumed by the processor is computed, and variations due to differ-

[3] Without DVS, energy consumption is the same for both EDF and RM, so EDF numbers alone would suffice. However, since some task sets are schedulable under EDF, but not under RM, we simulate both to verify that all task sets that are schedulable under RM are also schedulable when using the RM-based RT-DVS mechanisms.

ent types of instructions executed are not taken into account. With this simplification, the task execution modeling can be reduced to counting cycles of execution, and execution traces are not needed. The software-controlled halt feature, available on some processors and used for reducing energy expenditure during idle, is simulated by specifying an idle level parameter. This value gives the ratio between energy consumed during a cycle while halted and that during a cycle of normal operation (e.g., a value of 0.5 indicates a cycle spent idling dissipates one half the energy of a cycle of computation). For simplicity, only task execution and idle (halt) cycles are considered. In particular, this does not consider preemption and task switch overheads or the time required to switch operating frequency or voltages. There is no loss of generality from these simplifications. The preemption and task switch overheads are the same with or without DVS, so they have no effect on relative power dissipation. The voltage switching overheads incur a time penalty, which may affect the schedulability of some task sets, but incur almost no energy costs, as the processor does not operate during the switching interval.

The real-time task sets are specified using a pair of numbers for each task, indicating its period and worst-case computation time. The task sets are generated randomly as follows. Each task has an equal probability of having a short (1–10ms), medium (10–100ms), or long (100–1000ms) 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 [31]. 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 worst-case utilization of a task set.

3.2 Simulation Results

We have performed extensive simulations of the RT-DVS algorithms to determine the most important and interesting system parameters that affect energy consumption. Unless specified otherwise, we assume a DVS-capable platform that provides 3 relative operating frequencies (0.5, 0.75, and 1.0) and corresponding voltages (3, 4, and 5, respectively).

In the following simulations, we compare our RT-DVS algorithms to each other and to a non-DVS system. We also include a theoretical lower bound for energy dissipation. This lower bound reflects execution throughput only, and does not consider any timing issues (e.g., whether any task is active or not). It is computed by taking the total number of task computation cycles in the simulation, and determining the absolute minimum energy with which these can be executed over the simulation time duration with the given platform frequency and voltage specification. No real algorithms can do better than this theoretical lower bound, but it is interesting to see how close our mechanisms approach this bound.

Number of tasks:

In our first set of simulations, we determine the effects of varying the number of tasks in the task sets. Figure 9 shows the energy consumption for task sets with 5, 10, and 15 tasks for all of our RT-DVS algorithms as well as unmodified EDF. All of these simulations assume that the processor provides a perfect software-controlled halt function (so idling the processor will consume no energy), thus showing scheduling without any energy conserving features in the most favorable light. In addition, we assume that tasks do consume their worst-case computation requirements during each invocation. With these extremes, there is no difference between the statically-scaled and cycle-conserving EDF algorithms.

We notice immediately that the RT-DVS algorithms show potential for large energy savings, particularly for task sets with mid-range worst-case processor utilization values. The look-ahead RT-DVS mechanism, in particular, seems able to follow the theoretical lower bound very closely. Although the total utilization greatly affects energy consumption, the number of tasks has very little effect. Neither the relative nor the absolute positions of the curves for the different algorithms shift significantly when the number of tasks is varied. Since varying the number of tasks has little effect, for all further simulations, we will use a single value.

Varying idle level:

The preceding simulations assumed that a perfect software-controlled halt feature is provided by the processor, so idle time consumes no energy. To see how an imperfect halt feature affects power consumption, we performed several simulations varying the idle level factor, which is the ratio of energy consumed in a cycle while the processor is halted, to the energy consumed in a normal execution cycle. Figure 10 shows the results for idle level factors 0.01, 0.1, and 1.0. Since the absolute energy consumed will obviously increase as the idle state energy consumption increases, it is more insightful to look at the *relative* energy consumption by plotting the values normalized with respect to the unmodified EDF energy consumption.

The most significant result is that even with a perfect halt feature (i.e., idle level is 0), where the non-energy conserving schedulers are shown in the best light, there is still a very large percentage improvement with the RT-DVS algorithms. Obviously, as the idle level increases to 1 (same energy consumption as in normal operation), the percentage savings with voltage scaling improves. The relative performance among the energy-aware schedulers is not significantly affected by changing the idle power consumption level, although the dynamic algorithms benefit more than the statically-scaled ones. This is evident as the cycle-conserving EDF mechanism results diverge from the statically-scaled EDF results. This is easily explained by the fact that the dynamic algorithms switch to the lowest frequency and voltage during idle, while the static ones do not; with perfect idle, this makes no difference, but as idle cycle energy consumption approaches that of execution cycles, the dynamic algorithms perform relatively better. For the remainder of the simulations, we assume an idle level of 0.

Varying machine specifications:

All of the previous simulations used only one set of available frequency and voltage scaling settings. We now investigate the effects of varying the simulated machine specifications. The following summarizes the hardware voltage and frequency settings, where each tuple consists of the relative frequency and the corresponding processor voltage:

machine 0: { (0.5, 3), (0.75, 4), (1.0, 5) }
machine 1: { (0.5, 3), (0.75, 4), (0.83, 4.5), (1.0, 5) }
machine 2: { (0.36, 1.4), (0.55, 1.5), (0.64, 1.6),
(0.73, 1.7), (0.82, 1.8), (0.91, 1.9), (1.0, 2.0) }

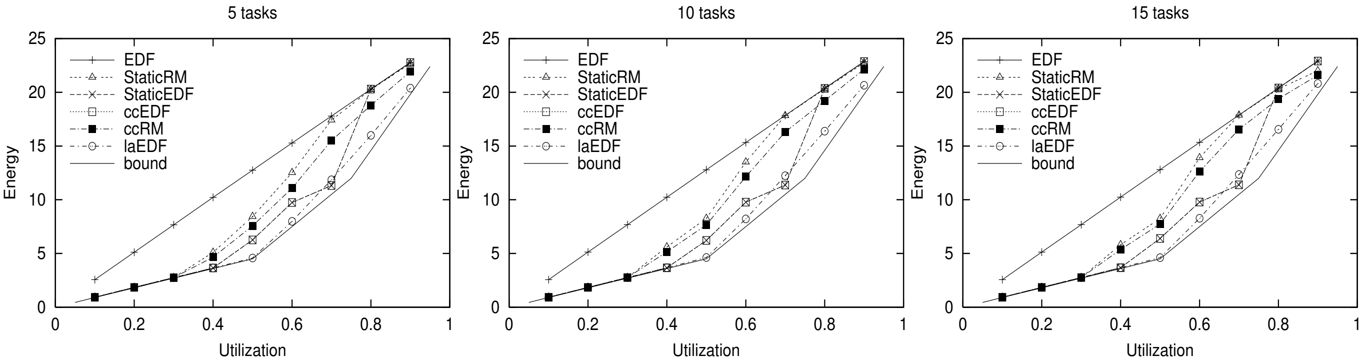

Figure 9: Energy consumption with 5, 10, and 15 tasks

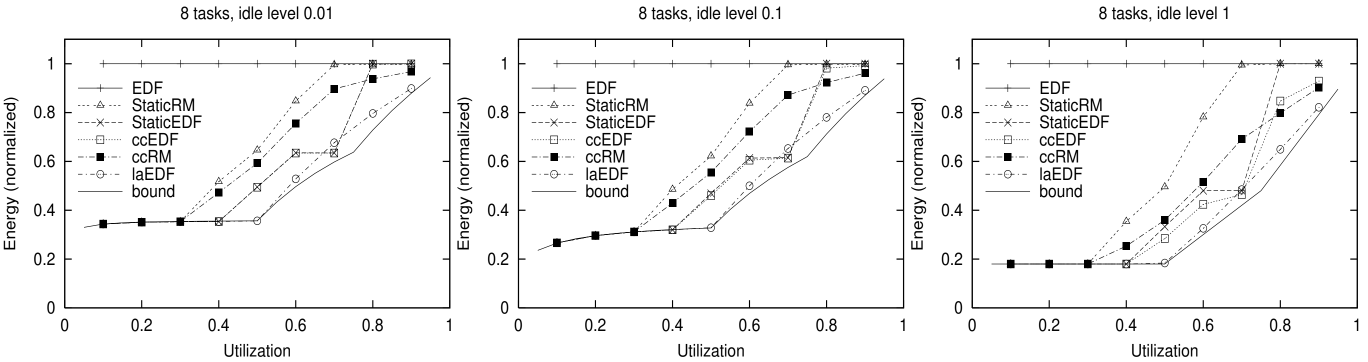

Figure 10: Normalized energy consumption with idle level factors 0.01, 0.1, and 1.0

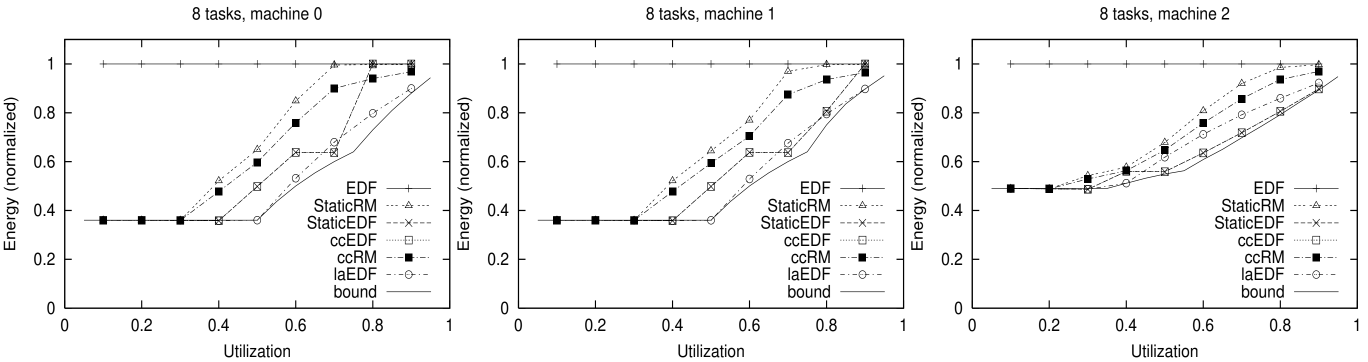

Figure 11: Normalized energy consumption with machine 0, 1, and 2

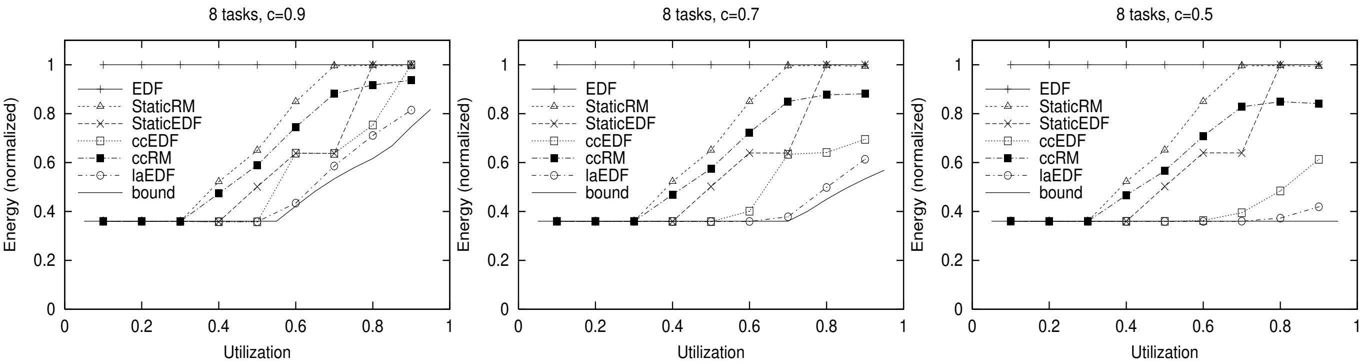

Figure 12: Normalized energy consumption with computation set to fixed fraction of worst-case allocation

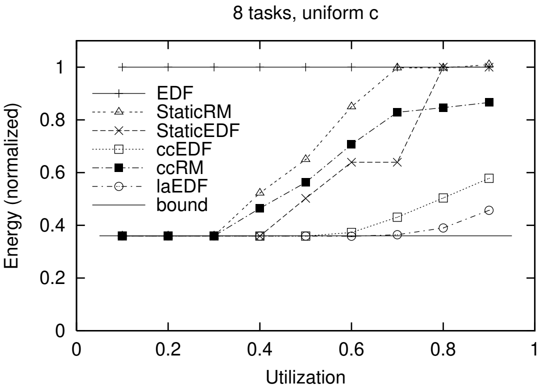

Figure 13: Normalized energy consumption with uniform distribution for computation

Figure 11 shows the simulation results for machines 0, 1, and 2. Machine 0, used in all of the previous simulations, has frequency settings that can be expected on a standard PC motherboard, although the corresponding voltage levels were arbitrarily selected. Machine 1 differs from this in that it has the additional frequency setting, 0.83. With this small change, we expect only slight differences in the simulation results with these specifications. The most significant change is seen with cycle-conserving EDF (and statically-scaled EDF, since the two are identical here). With the extra operating point in the region near the cross-over point between ccEDF and ccRM, the ccEDF algorithm benefits, shifting the cross-over point closer to full utilization.

Machine 2 is very different from the other two, and reflects the settings that may be available on a platform incorporating an AMD K6 processor with AMD's PowerNow! mechanism [1]. Again, the voltage levels are only speculated here. As it has many more settings to select from, the plotted curves tend to be smoother. Also, since the relative voltage range is smaller with this specification, the maximum savings is not as good as with the other two machine specifications. More significant is the fact that the cycle-conserving EDF outperforms the look-ahead EDF algorithm. ccEDF and staticEDF benefit from the large number of settings, since this allows them to more closely match the task set and reduce energy expenditure. In fact, they very closely approximate the theoretical lower bound over the entire range of utilizations. On the other hand, laEDF sets the frequency trying to defer processing (which, in the worst case, would require running at full speed later). With more settings, the low frequency setting is closely matched, requiring high-voltage, high-frequency processing later, hurting performance. With fewer settings, the frequency selected would be somewhat higher, so less processing is deferred, lessening the likelihood of needing higher voltage and frequency settings later, thus improving performance. In a sense, the error due to a limited number of frequency steps is detrimental in the ccEDF scheme, but beneficial with laEDF. These results, therefore, indicate that the energy savings from the various RT-DVS algorithms depend greatly on the available voltage and frequency settings of the platform.

Varying computation time:

In this set of experiments, we vary the distribution of the actual computation required by the tasks during each invocation to see how well the RT-DVS mechanisms take advantage of task sets that do not consume their worst-case computation times. In the preceding simulations, we assumed that the tasks always require their worst-case computation allocation. Figure 12 shows simulation results for tasks that require a constant 90%, 70%, and 50% of their worst-case execution cycles for each invocation. We observe that the statically-scaled mechanisms are not affected, since they scale voltage and frequency based solely on the worst-case computation times specified for the tasks. The results for the cycle-conserving RM algorithm do not show significant change, indicating that it does not do a very good job of adapting to tasks that use less than their specified worst-case computation times. On the other hand, both the cycle-conserving and look-ahead EDF schemes show great reductions in relative energy consumption as the actual computation performed decreases.

Figure 13 shows the simulation results using tasks with a uniform distribution between 0 and their worst-case computation. Despite the randomness introduced, the results appear identical to setting computation to a constant one half of the specified value for each invocation of a task. This makes sense, since the average execution with the uniform distribution is 0.5 times the worst-case for each task. From this, it seems that the actual distribution of computation per invocation is not the critical factor for energy conservation performance. Instead, for the dynamic mechanisms, it is the average utilization that determines relative energy consumption, while for the static scaling methods, the worst-case utilization is the determining factor. The exception is the ccRM algorithm, which, albeit dynamic, has results that primarily reflect the worst-case utilization of the task set.

4. IMPLEMENTATION

This section describes our implementation of a real-time scheduled system incorporating the proposed DVS algorithms. We will discuss the architecture of the system and present measurements of the actual energy savings with our RT-DVS algorithms.

4.1 Hardware Platform

Although we developed the RT-DVS mechanisms primarily for embedded real-time devices, our prototype system is implemented on the PC architecture. The platform is a Hewlett-Packard N3350 notebook computer, which has an AMD K6-2+ [1] processor with a maximum operating frequency of 550 MHz. Some of the power consumption numbers measured on this laptop were shown earlier in Table 1. This processor features *PowerNow!*, AMD's extensions that allow the processor's clock frequency and voltage to be adjusted dynamically under software control. We have also looked into a similar offering from Intel called *SpeedStep* [10], but this controls voltage and frequency through hardware external to the processor. Although it is possible to adjust the settings under software control, we were not able to determine the proper output sequences needed to control the external hardware. We do not have experience with the Transmeta Crusoe processor [29] or with various embedded processors (such as Intel XScale [9]) that are now supporting DVS.

The K6-2+ processor specification allows system software to select one of eight different frequencies from its built-in PLL clock generator (200 to 600 MHz in 50 MHz increments, skipping 250), limited by the maximum processor clock rate (550 MHz here). The processor has 5 control pins that can be used to set the voltage through an external regulator. Although 32 settings are possible, there is only one that is explicitly specified (the default 2.0V setting); the rest are left up to the individual manufacturers. HP chose to incorporate only 2 voltage settings: 1.4V and 2.0V. The volt-

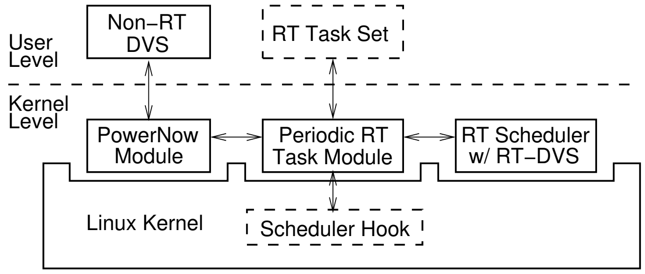

Figure 14: Software architecture for RT-DVS implementation

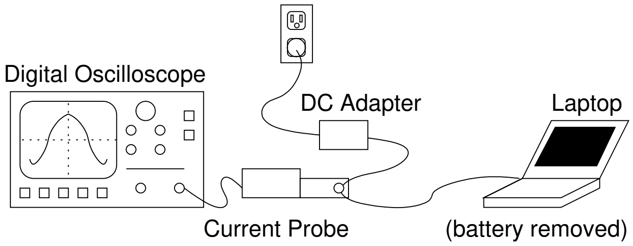

Figure 15: Power measurement on laptop implementation

age and frequency selectors are independently set, so we need a function to map each frequency to the appropriate available voltage level. As there are no such specifications publicly available, we determined this experimentally. The processor was stable up to 450 MHz at 1.4 V, and needed the 2.0 V setting for higher frequencies. Stability was checked using a set of CPU-intensive benchmarks (`mpg123` in a loop and Linux kernel compile), and verifying proper behavior. We note that this empirical study used a sample size of one, so the actual frequency to voltage mappings may vary for other machines, even of the same model.

The processor has a mandatory stop interval associated with every change of the voltage or frequency transition, during which the processor halts execution. This mandatory halt duration is meant to ensure that the voltage supply and clock have time to stabilize before the processor continues execution. As the characteristics of different hardware implementations of the external voltage regulators can vary greatly, this stop duration is programmable in multiples of 41 μs (4096 cycles of the 100 MHz system bus clock). According to our experience, it takes negligible time for frequency changes to occur. Using the CPU time-stamp register (basically a cycle counter), which continues to increment during the halt duration, we observed that around 8200 cycles occur during any transition to 200 MHz, and around 22500 cycles for a transition to 550 MHz, both with the minimum interval of 41 μs. This indicates that the frequency of the CPU clock changes quickly and that most of the halt time is spent at the target frequency. We do not know the actual time required for voltage transition to occur, but in our experiments using a halt duration value of 10 (approximately 0.4 ms) resulted in no observable instability. The switching overheads in our system, therefore, are 0.4 ms when voltage changes, and 41 μs when only frequency changes. As mentioned earlier, we can account for this switching overhead in the computation requirements of the tasks, since at most only two transitions are attributable to each task in each invocation.

4.2 Software Architecture

We implemented our algorithms as extension modules to the Linux 2.2.16 kernel. Although it is not a real-time operating system, Linux is easily extended through modules and provides a robust development environment familiar to us. The high-level view of our software architecture is shown in Figure 14. The approach taken for this implementation was to maximize flexibility and ease of use, rather than optimize for performance. As such, this implementation serves as a good proof-of-concept, rather than the ideal model. By implementing our kernel-level code as Linux kernel modules, we avoided any code changes to the Linux kernel, and these modules should be able to plug into unmodified 2.2.x kernels.

The central module in our implementation provides support for periodic real-time tasks in Linux. This is done by attaching call-back functions to hooks inside the Linux scheduler and timer tick handlers. This mechanism allows our modules to provide tight timing control as well as override the default Unix scheduling policy for our real-time tasks. Note that this module does not actually define the real-time scheduling policy or the DVS algorithm. Instead, we use separate modules that provide the real-time scheduling policy and the RT-DVS algorithms. One such RT scheduler/DVS module can be loaded on the system at a time. By separating the underlying periodic RT support from the scheduling and DVS policies, this architecture allows dynamic switching of these latter policies without shutting down the system or the running RT tasks. (Of course, during the switch-over time between these policy modules, a real-time scheduler is not defined, and the timeliness constraints of any running RT tasks may not be met). The last kernel module in our implementation handles the access to the PowerNow! mechanism to adjust clock speed and voltage. This provides a clean, high-level interface for setting the appropriate bits of the processor's special feature register for any desired frequency and voltage level.

The modules provide an interface to user-level programs through the Linux `/procfs` filesystem. Tasks can use ordinary file read and write mechanisms to interact with our modules. In particular, a task can write its required period and maximum computing bound to our module, and it will be made into a periodic real-time task that will be released periodically, scheduled according to the currently-loaded policy module, and will receive static priority over non-RT tasks on the system. The task also uses writes to indicate the completion of each invocation, at which time it will be blocked until the next release time. As long as the task keeps the file handle open, it will be registered as a real-time task with our kernel extensions. Although this high-level, filesystem interface is not as efficient as direct system calls, it is convenient in this prototype implementation, since we can simply use `cat` to read from our modules and obtain status information in a human readable form. The PowerNow! module also provides a `/procfs` interface. This will allow for a user-level, non-RT DVS demon, implementing algorithms found in other DVS literature, or to manually deal with operating frequency and voltage through simple Unix shell commands.

4.3 Measurements and Observations

We performed several experiments with our RT-DVS implementation and measured the actual energy consumption of the system. Figure 15 shows the setup used to measure energy consumption. The laptop battery is removed and the system is run using the external DC power adapter. Using a special current probe, a digital oscilloscope is used to measure the power consumed by the laptop as the product of the current and voltage supplied. This basic methodology is very similar to the one used in the PowerScope [6],

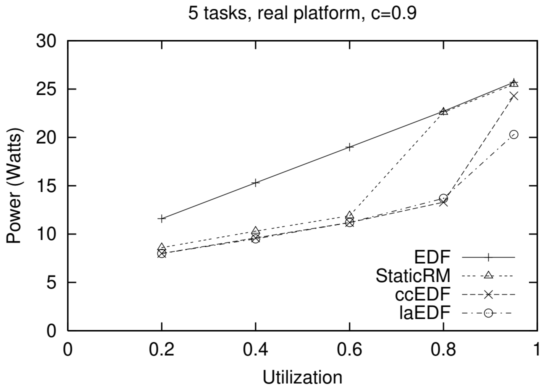

Figure 16: Power consumption on actual platform

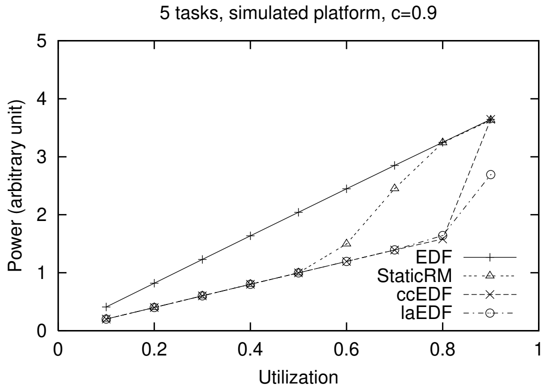

Figure 17: Power consumption on simulated platform

but instead of a slow digital multimeter, we use an oscilloscope that can show the transient behavior and also provide the true average power consumption over long intervals. Using the long duration acquisition capability of the digital oscilloscope, our power measurements are averaged over 15 to 30 second intervals.

Figure 16 shows the actual power consumption measured for our RT-DVS algorithms while varying worst-case CPU utilization for a set of 5 tasks which always consume 90% of their worst-case computation allocated for each invocation. The measurements reflect the total system power, not just the CPU energy dissipation. As a result, there is a constant, irreducible power drain from the system board consumption (the display backlighting was turned off for these measurements; with this on, there would have been an additional constant 6 W to each measurement). Even with this overhead, our RT-DVS mechanisms show a significant 20% to 40% reduction in power consumption, while still providing the deadline guarantees of a real-time system.

Figure 17 shows a simulation with identical parameters (including the 2 voltage-level machine specification) to these measurements. The simulation only reflects the processor's energy consumption, so does not include any energy overheads from the rest of the system. It is clear that, except for the addition of constant overheads in the actual measurements, the results are nearly identical. This validates our simulation results, showing that the results we have seen earlier really hold in real systems, despite the simplifying assumptions in the simulator. The simulations are accurate and may be useful for predicting the performance of RT-DVS implementations.

We also note two interesting phenomena that should be considered when implementing a system with RT-DVS. First, we noticed that the very first invocation of a task may overrun its specified computing time bound. This occurs only on the first invocation, and is caused by "cold" processor and operating system state. In particular, when the task begins execution, many cache misses, translation-look-aside buffer (TLB) misses, and page faults occur in the system (the last may be due to the copy-on-write page allocation mechanism used by Linux). These processing overheads count against the task's execution time, and may cause it to exceed its bound. On subsequent invocations, the state is "warm," and this problem disappears. This is due to the large difference between worst-case and average-case performance on general-purpose platforms, and explains why real-time systems tend to use specialized platforms to decrease or eliminate such variations in performance.

The second important observation is that the dynamic addition of a task to the task set may cause transient missed deadlines unless one is very careful. Particularly with the more aggressive RT-DVS schemes, the system may be so closely matched to the current task set load that there may not be sufficient processor cycles remaining before the task deadlines to also handle the new task. One solution to this problem is to immediately insert the task into task set, so DVS decisions are based on the new system characteristics, but defer the initial release of the new task until the current invocations of all existing tasks have completed. This ensures that the effects of past DVS decisions, based on the old task set, will have expired by the time the new task is released.

5. RELATED WORK

Recently, there have been a large number of publications describing DVS techniques. Most of them present algorithms that are very loosely-coupled with the underlying OS scheduling and task management systems, relying on average processor utilization to perform voltage and frequency scaling [7, 23, 30]. They are basically matching the operating frequency to some weighted average of the current processor load (or conversely, idle time) using a simple feedback mechanism. Although these mechanisms result in close adaptation to the workload and large energy savings, they are unsuitable for real-time systems. More recent DVS research [4, 19] have shown methods of maintaining good interactive performance for general-purpose applications with voltage and frequency scaling. This is done through prediction of episodic interaction [4] or by applying soft deadlines and estimating task work distributions [19]. These methods show good results for maintaining short response times in human-interactive and multimedia applications, but are not intended for the stricter timeliness constraints of real-time systems.

Some DVS work has produced algorithms closely tied to the scheduler [22, 24, 26], with some claiming real-time capability. These, however, take a very simplistic view of real-time tasks — only taking into account a single execution and deadline for a task. As such, they can only handle sporadic tasks that execute just once. They cannot handle the most important, canonical model of real-time systems, which uses periodic real-time tasks. Furthermore, it is not clear how new tasks entering the system can be handled in a timely manner, especially since all of the tasks are single-shot, and since the system may not have sufficient computing resources after having adapted so closely to the current task set.

To the best of our knowledge, there are only four recent papers [12, 28, 21, 8] that deal with DVS in a true real-time system's perspective. The first paper [12] uses a combined offline and online scheduling technique. A worst-case execution time (WCET) schedule, which provides the ideal operating frequency and voltage schedule assuming that tasks require their worst-case computation time, is calculated offline. The online scheduler further reduces frequency and voltage when tasks use less than their requested computing quota, but can still provide deadline guarantees by ensuring all invocations complete no later than in the WCET schedule. This is much more complicated than the algorithms we have presented, yet cannot deal effectively with dynamic task sets.

The second, a work-in-progress paper [28], presents two mechanisms for RT-DVS. One mechanism attempts to calculate the best feasible schedule; this is a computationally-expensive process and can only be done offline. The other is a heuristic based around EDF that tests schedulability at each scheduling point. The details of this online mechanism were not presented in [28]. Moreover, the assumption of a common period for all of the tasks is somewhat unrealistic — even if a polynomial transformation is used to produce common periods, they may need to schedule over an arbitrarily long planning cycle in their algorithm.

The third paper [21] looks at DVS from the application side. It presents a mechanism by which the application monitors the progress of its own execution, compares it to the profiled worst-case execution, and adjusts the processor frequency and voltage accordingly. The compiler inserts this monitoring mechanism at various points in the application. It is not clear how to determine the locations of these points in a task/application, nor how this mechanism will scale to systems with multiple concurrent tasks/applications.

The most recent [8] RT-DVS paper combines offline analysis with online slack-time stealing [15] and dynamic, probability-based voltage scaling. Offline analysis provides minimum operating rates for each task based on worst-case execution time. This is used in conjunction with a probability distribution for actual computation time to change frequency and voltage without violating deadlines. Excess time is used to run remaining tasks at lower CPU frequencies.

These papers, and indeed almost all papers dealing with DVS, only present simulations of algorithms. In contrast, we present fairly simple, online mechanisms for RT-DVS that work within the common models, assumptions, and contexts of real-time systems. We implemented and demonstrated RT-DVS in a real, working system. A recent paper [25] also describes a working DVS implementation, using a modified StrongARM embedded system board, which is used to evaluate a DVS scheduler in [26].

In addition to DVS, there has been much research regarding other energy-conserving issues, including work on application adaptation [5] and communication-oriented energy conservation [11]. These issues are orthogonal to DVS and complementary to our RT-DVS mechanisms.

6. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we have presented several novel algorithms for real-time dynamic voltage scaling that, when coupled with the underlying OS task management mechanisms and real-time scheduler, can achieve significant energy savings, while simultaneously preserving timeliness guarantees made by real-time scheduling. We first presented extensive simulation results, showing the most significant parameters affecting energy conservation through RT-DVS mechanisms, and the extent to which CPU power dissipation can be reduced. In particular, we have shown that the number of tasks and the energy efficiency of idle cycles do not greatly affect the relative savings of the RT-DVS mechanisms, while the voltage and frequency settings available on the underlying hardware and the task set CPU utilizations profoundly affect the performance of our algorithms. Furthermore, our look-ahead and cycle-conserving RT-DVS mechanisms can achieve close to the theoretical lower bound on energy. We have also implemented our algorithms and, using actual measurements, have validated that significant energy savings can be realized through RT-DVS in real systems. Additionally, our simulations do predict accurately the energy consumption characteristics of real systems. Our measurements indicate that 20% to 40% energy savings can be achieved, even including irreducible system energy overheads and using task sets with high values for both worst- and average- case utilizations.

In the future, we would like to expand this work beyond the deterministic/absolute real-time paradigm presented here. In particular, we will investigate DVS with probabilistic or statistical deadline guarantees. We will also explore integration with other energy-conserving mechanisms, including application energy adaptation and energy-adaptive communication (both real-time and best-effort).

Additionally, although developed for portable devices, RT-DVS is applicable widely in general real-time systems. The energy savings works well for extending battery life in portable applications, but can also reduce the heat generated by the real-time embedded controllers in various factory or home automation products, or even reduce cooling requirements and costs in large-scale, multiprocessor supercomputers.

7. REFERENCES

[1] ADVANCED MICRO DEVICES CORPORATION. *Mobile AMD-K6-2+ Processor Data Sheet*, June 2000. Publication # 23446.

[2] 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.

[3] ELLIS, C. S. The case for higher-level power management. In *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Rio Rico, AZ, Mar. 1999), pp. 162–167.

[4] FLAUTNER, K., REINHARDT, S., AND MUDGE, T. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01* (Rome, Italy, July 2001).

[5] FLINN, J., AND SATYANARAYANAN, M. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating System Principles* (Kiawah Island, SC, Dec. 1999), ACM Press, pp. 48–63.

[6] FLINN, J., AND SATYANARAYANAN, M. PowerScope: a tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile*

Computing Systems and Applications (New Orleans, LA, Feb. 1999), pp. 2–10.

[7] 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).

[8] 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).

[9] INTEL CORPORATION. http://developer.intel.com/design/intelxscal/.

[10] INTEL CORPORATION. *Mobile Intel Pentium III Processor in BGA2 and MicroPGA2 Packages*, 2000. Order Number 245483-003.

[11] KRAVETS, R., AND KRISHNAN, P. Power management techniques for mobile communication. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98)* (New York, Oct. 1998), ACM Press, pp. 157–168.

[12] KRISHNA, C. M., AND LEE, Y.-H. Voltage-clock-scaling techniques for low power in hard real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium* (Washington, D.C., May 2000), pp. 156–165.

[13] KRISHNA, C. M., AND SHIN, K. G. *Real-Time Systems*. McGraw-Hill, 1997.

[14] LEHOCZKY, J., SHA, L., AND DING, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium* (1989), pp. 166–171.

[15] 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).

[16] LEHOCZKY, J. P., SHA, L., AND STROSNIDER, J. K. Enhanced aperiodic responsiveness in hard real-time environments. In *Proc. of the 8th IEEE Real-Time Systems Symposium* (Los Alamitos, CA, Dec. 1987), pp. 261–270.

[17] LEUNG, J. Y.-T., AND WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation 2*, 4 (Dec. 1982), 237–250.

[18] 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.

[19] LORCH, J., AND SMITH, A. J. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference* (Cambridge, MA, June 2001), pp. 50–61.

[20] LORCH, J. R., AND SMITH, A. J. Apple Macintosh's energy consumption. *IEEE Micro 18*, 6 (Nov. 1998), 54–63.

[21] MOSSE, D., AYDIN, H., CHILDERS, B., AND MELHEM, R. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)* (Philadelphia, PA, Oct. 2000).

[22] 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).

[23] PERING, T., AND BRODERSEN, R. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'98* (Monterey, CA, Aug. 1998), pp. 76–81.

[24] PERING, T., BURD, T., AND BRODERSEN, R. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'00* (Rapallo, Italy, July 2000).

[25] POUWELSE, J., LANGENDOEN, K., AND SIPS, H. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01* (Rome, Italy, July 2001).

[26] POUWELSE, J., LANGENDOEN, K., AND SIPS, H. Energy priority scheduling for variable voltage processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01* (Huntington Beach, CA, Aug. 2001).

[27] STANKOVIC, J., ET AL. *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 1998.

[28] SWAMINATHAN, V., AND CHAKRABARTY, K. Real-time task scheduling for energy-aware embedded systems. In *Proceedings of the IEEE Real-Time Systems Symp. (Work-in-Progress Session)* (Orlando, FL, Nov. 2000).

[29] TRANSMETA CORPORATION. http://www.transmeta.com/.

[30] 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.

[31] 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.