Frequency-Aware Energy Optimization for Real-Time Periodic and Aperiodic Tasks

Xiliang Zhong and Cheng-Zhong Xu

Department of Electrical and Computer Engineering
Wayne State University, Detroit, Michigan 48202
{xlzhong, czxu}@wayne.edu

Abstract

Energy efficiency is an important factor in embedded systems design. We consider an embedded system with a dynamic voltage scaling (DVS) capable processor and its system-wide power consumption is dominated by the processor and memory. We present speed assignment polices for a set of periodic/aperiodic tasks that minimize the overall system energy consumption including active and idle power of CPU and other components. A limitation of most DVS-based system-wide energy optimization techniques is that they assume the number of worst-case execution cycles (WCEC) of a task is a constant, independent of CPU frequency. This is not the case when other system components such as memory are taken into account. In this paper, we decompose task execution time into two parts: on-chip inside CPU and off-chip outside the CPU. We propose a frequency-aware system-wide energy minimization approach and establish necessary and sufficient conditions for the optimality. By exploiting properties of the conditions, we derive a bisection algorithm that finds the optimal solution to offline periodic tasks in a linear time complexity. We apply a similar analytical approach to online aperiodic tasks scheduling and devise an iterative speed assignment algorithm in the complexity of $O(n^2)$. We prove it is optimal among all online algorithms without assumptions about future task releases.

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

General Terms Algorithms

Keywords Real-time systems, power-aware scheduling, dynamic power management, dynamic voltage scaling

1. Introduction

Energy saving becomes an increasingly important factor in system designs. It is crucial to battery-powered embedded devices because low power designs can extend their limited battery life. Dynamic voltage and frequency scaling (DVS) is one of the most effective techniques in reducing power consumption because the energy consumption of a CMOS circuit is a strictly convex function of the supply voltage.

Although reducing processor voltage leads to energy savings in the processor, the maximum operating frequency is also reduced with an approximately linear relation. Execution of tasks generally involves system components such as memory in addition to processors. Most of them can operate in more than one power state; for example they can be in active, standby (active but idle), and sleep states. If the components need to be active during the execution, they shall have an increased standby time as a result of processor speed slowdown. The overall system energy consumption is not necessarily reduced with a lower CPU speed. It is necessary to consider system standby power in processor speed assignment. Previous studies focused on processor energy savings, see [16] [14] [20] [17] for examples. Few results are available for system-wide energy optimization. By considering the standby power of other components, recent studies showed the existence of *critical speed* for a task beyond which processor slowdown is no longer beneficial [5, 11, 22, 24].

In real-time systems, the objective of low power designs is to minimize system-wide energy consumption of all tasks without violating their deadlines. In addition, applications with different resource requirements tend to have different power functions and critical speeds. These application-specific characteristics impose extra constraints in lower power designs. Past studies solve a simplified version of the problem with relaxed timing requirement [5], apply heuristics to get approximated solutions [11, 24], or are limited to periodic tasks [2]. A recent study on system-wide energy optimization [19] proves that the energy optimization problem is NP-hard on processors with discrete speed levels and only approximated solutions can be derived. In this paper, we relax the discreteness constraint and assume processor's speed can be scaled continuously. We obtain optimal solutions with a linear running time algorithm. The results can be applied back to the discrete model by mapping continuous voltage levels to discrete ones.

A limitation of most existing studies on DVS-based low power systems design is that they assume processor slowdown has no effect on the worst-case execution cycles (WCEC) of a task. In reality, the number of cycles changes with different frequency settings. As in [17] [5], we distinguish the cycles between on-chip in CPU and off-chip in memory. The number of on-chip cycles scales with CPU frequency, but the off-chip number does not.

In this paper, we consider an embedded system whose power consumption is dominated by a processor and memory and we present a system-wide energy optimization approach that takes into account the impact of frequency scaling on WCEC. This frequency-aware approach can be applied to both periodic and aperiodic task models. For offline periodic tasks, we establish a necessary and sufficient condition for the solution optimality. By exploiting

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

properties of the condition, we propose a bisection algorithm that finds the optimal solution in a linear time complexity.

Real-time aperiodic tasks are effective in modeling applications with unpredictable arrival times and external events such as operator's commands. The irregular releases of aperiodic tasks call for online decision-making, without assuming any timing information before task releases. Most studies on online aperiodic tasks energy savings using DVS were limited to reduction of dynamic energy consumption of processors [16, 18, 13, 20]. For on-line aperiodic tasks, we propose a frequency-aware speed assignment algorithm for system-wide energy optimization, in a way similar to the case of periodic tasks. The algorithm is run in an iterative process with a time complexity of $O(n^2)$, where n is the number of tasks. It is proved to be optimal among all online algorithms without assumptions about future task releases.

The rest of the paper is organized as follows. Section 2 provides task and power models of the system. We analyze the optimal conditions and propose algorithms in finding the optimal solutions for periodic tasks and aperiodic tasks in Sections 3 and 4, respectively. Performance evaluation in terms of energy savings and algorithm running times is presented in Section 5. We review related studies in Section 6. Section 7 concludes the article.

2. Preliminaries

2.1 Task Model

We consider inter-task DVS in a uniprocessor system for a set of n periodic real-time tasks, labeled in integers from 1 to n. We assume the tasks to be independent and preemptive. Task i is characterized by a 3-tuple $\{C_i, T_i, D_i\}$, where T_i is the task period and C_i is the worst case execution time (WCET) under the maximum frequency of the processor. The relative deadline D_i is assumed to be equal to the period T_i. All tasks are scheduled by the Earliest Deadline First (EDF) policy.

Another type of hard real-time tasks under consideration is aperiodic tasks. Each aperiodic task is characterized by its WCET C_i and relative deadline D_i. The release times of an aperiodic task can be arbitrary with irregular intervals. We therefore consider online scheduling in which no knowledge of future task releases is available at the time of speed assignment; parameters of an aperiodic task become known only after its release.

As the execution of a task may be stalled for memory access, not all execution cycles scale with the processor speed. We can distinguish the execution time between on-chip in CPU and off-chip in memory in a similar way to [5] [17]. The on-chip time, denoted by C_i^{on}, scales with the CPU frequency and the off-chip time, denoted by C_i^{off}, is a constant independent of the frequency. Let S_i denote the normalized CPU speed for task i with respect to the maximum CPU speed. The WCET of task i under speed S_i can be expressed as:

$$C_i = \frac{C_i^{on}}{S_i} + C_i^{off}. \tag{1}$$

2.2 Processor Energy Model

We consider a DVS processor that can scale its voltage and speed continuously within its operating range. Whenever the processor speed is scaled, the supply voltage is scaled following a roughly linear relation. Practically, processors only have discrete speed levels. In general, there are two approaches dealing with the discreteness. The first is to formulate and solve the energy optimization problem based on the discrete processor model. The problem is NP-hard and only approximated solutions can be obtained, even if only processor energy is considered [14]. The other approach is to formulate the optimization problem under continuous speed levels, which is tractable. The obtained speed can be mapped to discrete speed by using its two neighboring speed [10], [12]. With an extra speed switch during task execution, the latter approach can lead to more energy savings. DVS can effectively reduce the dynamic power consumption because of the quadratic dependence on voltage level V_{dd}. Meanwhile, static power consumption of processors is increasing rapidly with the scales of the devices and may be comparable to the dynamic power dissipation. Aware of the static power, we do not assume any form of the power function as long as it is differentiable and energy per cycle is strictly convex.

2.3 System Energy Model

We consider an embedded computing system whose power consumption is dominated by a DVS-enabled processor and a memory subsystem. Both the processor and the memory have multiple power states such as active, standby (active but idle), and sleep. During on-chip execution for task i in the processor, the overall power is a sum of the processor power in the active state, the memory power in a standby state, and the system-dependent power to keep basic circuits on. We represent the overall power consumption during on-chip execution as $P_{i,sys}^{on}(S_i)$. In contrast, the power during off-chip execution includes the active power of the memory, the standby power of the processor, and the system-dependent power. We denote it as $P_{i,sys}^{off}(S_i)$. Since processor power in idle state increases with processor speed, the power consumption during off-chip execution is also a function of the speed. The total energy consumption for task i under speed S_i is then

$$E_i(S_i) = P_{i,sys}^{on}(S_i)\frac{C_i^{on}}{S_i} + P_{i,sys}^{off}(S_i)C_i^{off}. \tag{2}$$

The energy function (2) is generally a strictly convex function of the normalized speed S_i. It is not difficult to find the speed that minimizes $E_i(S_i)$ by solving $E_i'(S_i) = 0$. We define the speed as the critical speed for task i and denote it as S_i^{crt}. This is the speed beyond which it is no longer energy-efficient. If the speed is lower than the minimal processor speed, we define the critical speed as the minimal speed.

2.4 Assumptions

We assume tasks execute up to their WCETs. In practice, the actual execution times of tasks can be much smaller than their WCETs. The resultant slacks can be reclaimed by other task instances dynamically for possibly further slowdown. Existing slack management techniques can be easily adapted to consider the variation of actual task execution times; examples include [23] [24] [2] [15]. We do not consider the impact of actual task execution times in this work. In addition, we note that the decomposition of tasks execution time into separate on-chip and off-chip times can be best used for processors with in-order single-issue pipelines. For processors with out-of-order execution and superscalar designs, on-chip execution can continue in case of off-chip data/instruction stalls. The execution time breakdown is not accurate due to the overlapped on-chip/off-chip execution. Finally, the major system components we consider are processor and memory. This is because memory is the mostly widely used component other than processor in an embedded system and power consumption of the memory can take up to half of the system power [5]. This is not applicable for systems with a wireless transceiver, which is another major source of power expenditure [21].

3. Energy Optimization for Periodic Tasks

3.1 Problem Formulation

For a periodic task set, we consider a hyper-period T, which is the least common multiple of T_1, ..., T_n. As the schedule repeats every T time units, our objective is to minimize the overall energy

consumption during the hyper-period. According to studies in [3], for any task i, it is optimal for all its task instances to run at the same processor speed. The problem of Frequency-Aware SysTem-wide EneRgy optimization for Periodic tasks (FASTER-P) then becomes to find the set of speeds $S = [S_1, S_2, ..., S_n]$ that

$$\text{minimize } \sum_{i=1}^{n} \frac{T}{T_i} E_i(S_i) \tag{3}$$

$$\text{subject to } \sum_{i=1}^{n} \frac{T}{T_i}(\frac{C_i^{on}}{S_i} + C_i^{off}) \leq T \tag{4}$$

$$S_i^{crt} \leq S_i \leq 1,\ 1 \leq i \leq n. \tag{5}$$

The objective function (3) is the total energy consumption in a hyper-period. The constraint (4) requires that the total execution time in the hyper-period not exceed T. The condition ensures the feasibility under EDF. As T is a constant, we will omit T in the two functions. The processor utilization, $\sum_{i=1}^{n} \frac{C_i^{on}}{T_i S_i}$, determines the overall on-chip workload. It cannot exceed the maximum on-chip utilization, as defined by $1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$. This optimization problem with inequality constraints can be solved by the method of Lagrange multipliers [7]. In the method, we derive the Kuhn-Tucker conditions as (4) and (5) together with:

$$\frac{1}{T_i} E_i'(S_i) - \lambda \frac{C_i^{on}}{T_i S_i^2} - \mu_i + \nu_i = 0 \tag{6}$$

$$\lambda(\sum_{i=1}^{n}(\frac{C_i^{on}}{T_i S_i} + \frac{C_i^{off}}{T_i}) - 1) = 0 \tag{7}$$

$$\mu_i(S_i^{crt} - S_i) = 0 \tag{8}$$

$$\nu_i(S_i - 1) = 0, \tag{9}$$

where λ, μ_i, ν_i are non-negative Lagrange multipliers for the optimality of solutions to the problem.

As the target function is strictly convex, the conditions are necessary and sufficient [7]. The optimal solution $S^* = [S_1^*, S_2^*, ..., S_n^*]$ can be obtained by solving these equations. However, it is time-consuming to get the solution when the number of tasks is large. We next provide a more efficient solution by exploiting properties of the optimal solution.

3.2 Analysis of the Optimal Solution

Regarding condition (7) in the case of optimal solution, we have the following two cases:

1) Processor utilization $\sum_{i=1}^{n} \frac{C_i^{on}}{T_i S_i^*} < 1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$. In this case, the Lagrange multiplier λ must be zero. Condition (6) becomes $\frac{1}{T_i} E_i'(S_i^*) - \mu_i + \nu_i = 0$. For any $i \in [1, n]$, there are three subcases regarding the boundary condition (5):

- $S_i^{crt} < S_i^* < 1$. We must have $\mu_i = \nu_i = 0$ according to (8) and (9). Consequently, $E_i'(S_i^*) = 0$, which implies S_i^* must equal S_i^{crt}. It follows that $S_i^{crt} < S_i^{crt}$. This is in contradiction to the definition of S_i^{crt}.
- $S_i^* = 1$. We have $\mu_i = 0$ according to (8). Condition (6) becomes $\frac{1}{T_i} E_i'(1) + \nu_i = 0$. As ν_i is non-negative, we get $E_i'(1) \leq 0$. However, as $E_i(S_i)$ is convex, $S_i = 1$ and $S_i^{crt} \leq 1$ imply $E_i'(1) \geq 0$. Consequently, we get $E_i'(1) = 0$ or equivalently, $S_i^{crt} = S_i^{crt} = 1$.
- $S_i^* = S_i^{crt}$. We have $\nu_i = 0$ according to (9). It follows that $\frac{1}{T_i} E_i'(S_i^{crt}) = \mu_i \geq 0$, which can be satisfied as $E_i(S_i)$ is convex and $S_i^{crt} \geq S_i^{crt}$.

In summary, when processor utilization under the optimal speed assignment is smaller than the maximum on-chip utilization, all tasks run at their critical speed S_i^{crt}.

2) Processor utilization $\sum_{i=1}^{n} \frac{C_i^{on}}{T_i S_i^*} = 1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$. In this case, condition (7) is always satisfied. We first solve the problem under the constraint of feasible condition (4) only, without the boundary condition (5). Condition (6) is then reduced to $\frac{1}{T_i} E_i'(S_i) - \lambda \frac{C_i^{on}}{T_i S_i^2} = 0$. It follows that for all i, $1 \leq i \leq n$,

$$\lambda = E_i'(S_i) \frac{S_i^2}{C_i^{on}}$$

$$= (P_{i,sys}^{on}(S_i))' S_i - P_{i,sys}^{on}(S_i) + (P_{i,sys}^{off}(S_i))' S_i^2 \frac{C_i^{off}}{C_i^{on}}. \tag{10}$$

The right-hand side of (10) represents the rate of power change of task i at speed S_i. We call it the *power rate* of a task. Condition (10) indicates that all tasks should have the same power rate. For notational convenience, we let $G_i(S_i)$ represent the power rate of task i at speed S_i. As $E_i(S_i)$ is a strictly convex function, the derivative of $G_i(S_i)$ is always positive. This means for any speed S_i that satisfies $S_i > S_i^{crt}$, $G_i(S_i)$ is strictly increasing with S_i.

The speed for task i can be expressed as $S_i = G_i^{-1}(\lambda)$. λ and S_i are hence determined by solving the constraint $\sum_{i=1}^{n} \frac{C_i^{on}}{T_i S_i} = 1 - \sum_{i=1}^{n} C_i^{off}$. If the resultant speed for any task i is in the range $[S_i^{crt}, 1]$, then conditions (4)-(9) are satisfied and we get the optimal solution to FASTER-P. Otherwise, we adjust the speeds according to the boundary condition (5). That is, if $S_i > 1$, we set S_i to 1; if $S_i < S_i^{crt}$, we set it to S_i^{crt}.

PROPOSITION 3.1. *The processor utilization under slowdown, $\sum_{i=1}^{n} \frac{C_i^{on}}{T_i S_i}$, is non-increasing with λ, when $\lambda_{min} \leq \lambda \leq \lambda_{max}$.*

Proof: To prove that the processor utilization is non-increasing with λ, we only need to show that no speed reduction is possible as λ increases. For any λ_1 and λ_2, $\lambda_{min} \leq \lambda_1 < \lambda_2 \leq \lambda_{max}$, we get $G_i^{-1}(\lambda_1) < G_i^{-1}(\lambda_2)$ for any task i. If both speeds are in the range $[S_i^{crt}, 1]$, no speed adjustment is necessary and speed of task i increases.

If either speed violates its bounds, we need to adjust it according to (5). We first consider the case when the lower bound S_i^{crt} is violated. There are two subcases

- $G_i^{-1}(\lambda_1) < G_i^{-1}(\lambda_2) < S_i^{crt}$. Both speeds are set to S_i^{crt}.
- $G_i^{-1}(\lambda_1) < S_i^{crt} \leq G_i^{-1}(\lambda_2)$. The speed under λ_1 is set to S_i^{crt}. The speed under λ_2 is assigned to the smaller value of $G_i^{-1}(\lambda_2)$ and 1; both of them are no smaller than S_i^{crt}.

Similar analysis can be conducted for the violation of the upper bound. That is, if both $G_i^{-1}(\lambda_1)$ and $G_i^{-1}(\lambda_2)$ exceed 1, they are set to 1; if only $G_i^{-1}(\lambda_2)$ is larger than 1, it is set to the maximum speed, which is guaranteed to be no smaller than the speed under λ_1.

In all of the cases, the resultant speeds are non-decreasing and the processor utilization is non-increasing with λ. This completes the proof □

3.3 A Bisection Search Algorithm

According to Proposition 3.1, we can use bisection to find the λ that minimizes energy consumption of FASTER-P in Algorithm 1. The procedure GET_SPEED returns the set of speeds with a given λ. The main procedure returns the optimal speed setting given a task set and a threshold ϵ. The basic idea is if the processor utilization under mid exceeds the maximum on-chip utilization, we decrease

the utilization by restricting the search range to $(mid, high]$; otherwise, we increase the utilization with the search range $[low, mid)$. We need to first check whether the processor is still under-utilized when each task takes its critical speed in lines 2-4. In such case, we let all tasks run at their critical speeds. Otherwise, the algorithm continues and terminates in two cases. The first condition at the **while** loop ensures the utilization is smaller than the maximum utilization and their difference is smaller than the threshold ϵ. In practice, with a sufficiently small ϵ the computed set of speeds can be used effectively as the optimal speed setting. In the second condition, we use δ to denote the computation precision of a real number. Line 6 means the precision is reached and the value of mid cannot be distinguished from low and $high$.

Algorithm 1 Finding the optimal speed assignment for periodic tasks (FASTER-P) using bisection.

1: $util = 0, low = \lambda_{min}, high = \lambda_{max}$
2: **if** $\sum_{i=1}^{n} \frac{C_i^{on}}{T_i S_i^{crt}} \leq 1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$ **then**
3: return $S_i = S_i^{crt},\ 1 \leq i \leq n$
4: **end if**
5: **while** $(1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i} - util) > \epsilon$ or $util > 1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$ **do**
6: **if** $(high - low) \leq \delta$ **then** ▷ precision reached
7: **return** $[S_1, ..., S_n] =$ GET_SPEED$(high)$
8: **end if**
9: $mid = (low + high)/2$
10: $[S_1, ..., S_n] =$ GET_SPEED(mid)
11: $util = \sum_{i=1}^{n} \frac{C_i^{on}}{S_i T_i}$
12: **if** $util > 1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$ **then**
13: $low = mid$
14: **else if** $util < 1 - \sum_{i=1}^{n} \frac{C_i^{off}}{T_i}$ **then**
15: $high = mid$
16: **end if**
17: **end while**
18: **return** $[S_1, ..., S_n]$

19: **procedure** GET_SPEED(mid)
20: **for** $i = 1$ to n **do**
21: $S_i = G_i^{-1}(mid)$
22: $S_i = \min(S_i, 1)$
23: $S_i = \max(S_i, S_i^{crt})$
24: **end for**
25: **return** $[S_1, ..., S_n]$
26: **end procedure**

In general, the running time for a bisection is characterized by a linear order of convergence. We consider the worst case iteration steps of Algorithm 1. The length of the search space in the **while** loop is reduced in half in each iteration. In the worst case, the loop is terminated if $(high - low) \leq \delta$, in which the search space cannot be further divided. The maximum number of iterations needed is $log((\lambda_{max} - \lambda_{min})/\delta)$. Because each iteration takes $O(n)$ time due to the speed computation in GET_SPEED, the overall running time of the algorithm is $O(n \cdot log\frac{\lambda_{max} - \lambda_{min}}{\delta})$. As the values of λ_{max} and λ_{min} can be bounded in a system with limited components, the running time is linear in the number of tasks. If the loop terminates according to the condition in line 5, less running time is required.

4. Energy Optimization for Online Aperiodic Tasks

Due to the irregular releases of aperiodic tasks, we consider online scheduling with task timing parameters known only after task releases. Similar to the way of handling aperiodic tasks online in [6] [9] [20], we first consider a set of tasks released at time zero. In the general case when all tasks are released at different times, we can easily adapt the formulation by considering only ready tasks and by changing C_i, D_i to residual execution times and deadlines respectively. When there is a task release, we first determine whether the task should be admitted by an acceptance test extended from [9]. The basic idea is to find if the tasks are feasible under the maximum processor speed. Suppose there are n ready tasks sorted in a non-decreasing order of deadlines. The maximum instantaneous utilization of all tasks U_{max} is defined as $U_{max} = \max_{1 \leq i \leq n} U_i$, where $U_i = \sum_{j=1}^{i} C_j^{on}/D_j$. The task is admitted if $U_{max} \leq 1 - \sum_{i=1}^{n} C_i^{off}$.

Once a new task is accepted, we re-calculate the speed settings of all ready tasks so as to minimize their total energy consumption. The Frequency-Aware SysTem-wide EneRgy minimization for online Aperiodic tasks (FASTER-A), can be formulated as

$$\text{minimize} \sum_{i=1}^{n} E_i(S_i) \tag{11}$$

$$\text{subject to} \sum_{i=1}^{j} (\frac{C_i^{on}}{S_i} + C_i^{off}) \leq D_j, 1 \leq j \leq n-1 \tag{12}$$

$$\sum_{i=1}^{n} (\frac{C_i^{on}}{S_i} + C_i^{off}) = D_n \tag{13}$$

$$S_i^{crt} \leq S_i \leq 1,\ 1 \leq i \leq n. \tag{14}$$

The sets of constraints (12) and (13) ensure the feasibility of the tasks. That is, no task misses its deadline. In (13), we let task n finish at its deadline for maximized energy savings.

FASTER-A is an optimization problem with one equality constraint and $3n - 1$ inequality constraints. In general, with a large number of constraints, it is time-consuming to solve the problem and also it is not necessarily convergent. We will show that an efficient solution is possible by exploiting properties of the optimal solution. In the next two subsections, we will first solve a simplified version of FASTER-A considering only feasibility constraints and use it as a basis to solve the problem.

4.1 Optimization with Only Feasible Conditions

We let FASTER-AF represent optimization problem FASTER-A without the constraints of lower and upper bounds of speed in (14). The problem can be solved by forming the Lagrangian as

$$L(S, \lambda_1, ..., \lambda_n) = \sum_{i=1}^{n} E_i(S_i) + \sum_{j=1}^{n} \lambda_j (\sum_{i=1}^{j} (\frac{C_i^{on}}{S_i} + C_i^{off}) - D_j),$$

where λ_j, $1 \leq j \leq n$, are the non-negative Lagrange multipliers. Accordingly, we obtain Kuhn-Tucker conditions as (12)-(14) and

$$E_i'(S_i)\frac{S_i^2}{C_i} - \sum_{j=i}^{n} \lambda_j = 0, 1 \leq i \leq n \tag{15}$$

$$\lambda_j (\sum_{i=1}^{j} (\frac{C_i^{on}}{S_i} + C_i^{off}) - D_j) = 0, \lambda_j \geq 0, 1 \leq j \leq n-1. \tag{16}$$

Because of the strict convexity of the objective function in (11), the conditions are necessary and sufficient for the optimal solution $[S_1^*, S_2^*, ..., S_n^*]$.

4.1.1 Analysis of the Optimal Solution

In a similar way to the case of periodic tasks, we define the power rate of a task at speed S_i in (15) as $G_i(S_i) = E_i'(S_i)S_i^2/C_i^{on}$.

We will omit the parameter S_i of the function for brevity in the following if confusion will not result. The completion time of task j, $\sum_{i=1}^{j}(C_i^{on}/S_i+C_i^{off})$ in (16), can be either equal to or smaller than its deadline D_j (task j finishes at or before its deadline). The two conditions (15) (16) yield the following propositions regarding the power rates of tasks.

LEMMA 4.1. *The power rate of task i, G_i, is non-increasing with i. That is, for any i_1 and i_2, $1 \leq i_1 < i_2 \leq n$, we have $G_{i_1} \geq G_{i_2}$.*

Proof: It follows from (15) that $G_{i_2} = \sum_{j=i_2}^{n} \lambda_j$ and $G_{i_1} = \sum_{j=i_1}^{n} \lambda_j = \sum_{j=i_1}^{i_2-1} \lambda_j + G_{i_2}$. As λ_j is non-negative, we get $G_{i_1} \geq G_{i_2}$. □

LEMMA 4.2. *If all tasks between i_1 and i_2, $1 \leq i_1 < i_2 < n$, finish before their deadlines, then they have the same power rate. That is $G_i = G_{i_2+1}$, for all i, $i_1 \leq i \leq i_2$.*

Proof: For any task i, $i_1 \leq i \leq i_2$, if it finishes before its deadline D_i, i.e., $\sum_{j=1}^{i}(C_j^{on}/S_j + C_j^{off}) < D_i$, we get $\lambda_i = 0$ according to (16). Combining it with (15) leads to $G_i = \sum_{j=i}^{n} \lambda_j = \sum_{j=i_2+1}^{n} \lambda_j = G_{i_2+1}$. □

Given a set of n tasks released at time 0 in a non-decreasing order of their deadlines, we determine the speed assignment by first solving subproblems with fewer tasks. Define optimization problems for the first i tasks as $\mathcal{J}_i$, where $1 \leq i \leq n$. It is trivial to solve $\mathcal{J}_1$ since there is only one task to be scheduled. The targeted problem FASTER-AF is equivalent to $\mathcal{J}_n$. Let $S_j^{(i)}$ denote the speed of task j for problem $\mathcal{J}_i$, where $1 \leq j \leq i$. The optimal solution S_j^* equals $S_j^{(n)}$. Before we show how to solve $\mathcal{J}_{i+1}$ with a solution to $\mathcal{J}_i$, we first present the following property.

LEMMA 4.3. *Let $\mathcal{L}$ be the set of tasks that have their speeds changed during the ith iteration adding task $i+1$. All tasks $j \in \mathcal{L}$ have their speeds increased and have the same power rate after the iteration.*

Proof: We prove the lemma in three steps. First, we note that there must be at least one task in $\mathcal{L}$ with an increase of speed. This is because $\sum_{j=1}^{i}(C_j^{on}/S_j^{(i)} + C_j^{off}) = D_i$ in $\mathcal{J}_i$. Suppose to the contrary, some of the speeds are reduced while others remain the same. Then the summed execution time exceeds D_i, which is not allowed.

Let l be the first task in $\mathcal{L}$ with an increase of its speed after the inclusion of task $i+1$, i.e., $S_l^{(i+1)} > S_l^{(i)}$. In the second step, we prove no speed reduction is possible for all tasks that finish before l by contradiction. Suppose there is such a task. The completion time of task $l-1$ in $\mathcal{J}_{i+1}$ is then larger than that in $\mathcal{J}_i$. That is, $\sum_{j=1}^{l-1}(C_j^{on}/S_j^{(i+1)} + C_j^{off}) > \sum_{j=1}^{l-1}(C_j/S_j^{(i)} + C_j^{off})$. Since the schedule after adding task $i+1$ must be feasible, we have $\sum_{j=1}^{l-1}(C_j^{on}/S_j^{(i+1)}+C_j^{off}) \leq D_{l-1}$. This means task $l-1$ before the iteration adding task $i+1$ finishes before its deadline D_{l-1}, i.e., $\sum_{j=1}^{l-1}(C_j/S_j^{(i)} + +C_j^{off}) < D_{l-1}$. According to Lemma 4.2, we have

$$G_{l-1}^{(i)} = G_l^{(i)}, \tag{17}$$

where we denote $G_l(S_l^{(i)})$ as $G_l^{(i)}$ for brevity. After adding task $i+1$, since task l has a speed increase, we get

$$G_l^{(i+1)} > G_l^{(i)}. \tag{18}$$

The speed of task $l-1$ can be either reduced or unchanged depending whether task $l-1$ belongs to $\mathcal{L}$. In another word,

$$G_{l-1}^{(i)} \geq G_{l-1}^{(i+1)}. \tag{19}$$

Equations (17)-(19) yield $G_{l-1}^{(i+1)} < G_l^{(i+1)}$, which means task $l-1$ has a smaller power rate than that of task l. This is in contradiction to Lemma 4.1. Therefore, no speed can be reduced for all tasks that finish before l

Finally, we prove that all tasks starting from l have their speed increased. Because task l is the first with a speed change (increase), it is completed ahead of the scheduled time due to the solution to $\mathcal{J}_i$. It means task l finishes before its deadline after the inclusion of task $i+1$. As a result, $G_l^{(i+1)} = G_{l+1}^{(i+1)}$ according to Lemma 4.2. Combining the equation with (18), we get $G_{l+1}^{(i+1)} > G_l^{(i)}$. Recall $G_l^{(i)} \geq G_{l+1}^{(i)}$ by Lemma 4.1. It follows that $G_{l+1}^{(i+1)} > G_{l+1}^{(i)}$ and he speed of task $l+1$ increases, i.e., $S_{l+1}^{(i+1)} > S_{l+1}^{(i)}$. Similar analysis can be applied to any task from $l+2$ to i and hence all tasks between l and i have their speeds increased. These tasks finish before their deadlines and have the same power rate by Lemma 4.2. This completes the proof. □

We can get the solution to $\mathcal{J}_{i+1}$ according to that of $\mathcal{J}_i$ by Theorem 4.1.

THEOREM 4.1. *Let l be the first task with an increased speed as a result of adding task $i+1$. Given a speed assignment of the first i tasks, we get the optimal solution to the first $i+1$ tasks according to:*

$$\sum_{j=l}^{i+1} \frac{C_j^{on}}{G_j^{-1}(\lambda^{(i+1)})} = \sum_{j=l}^{i} \frac{C_j^{on}}{S_j^{(i)}} + (D_{i+1} - D_i - C_{i+1}^{off}), \tag{20}$$

where $\lambda^{(i+1)}$ equals $\lambda_{i+1}^{(i+1)}$ and is the value of the Lagrange multipliers for all tasks in $\mathcal{L}$.

Proof: The overall execution time of tasks from l to i before adding task $i+1$ is $\sum_{j=l}^{i}(C_j^{on}/S_j^{(i)} + C_j^{off})$. The time added by the interval $D_{i+1} - D_i$ is equal to the overall execution time of tasks from l to $i+1$ in $\mathcal{L}_{i+1}$, as can also be observed from Figure 1. Formally, it is:

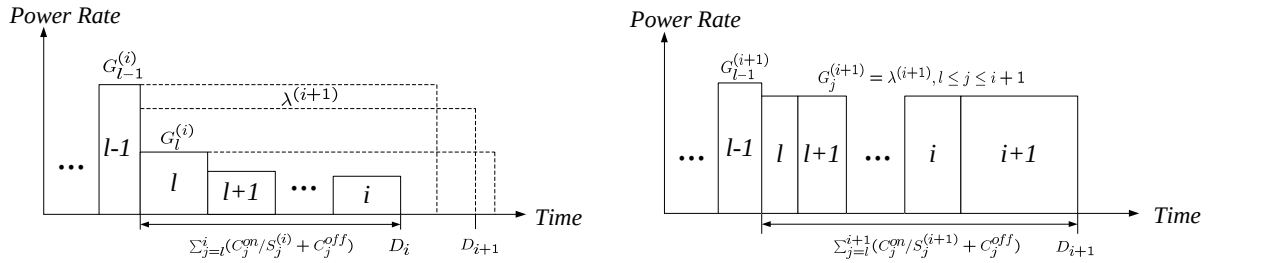
$$\sum_{j=l}^{i+1}(\frac{C_j^{on}}{S_j^{(i+1)}} + C_j^{off}) = \sum_{j=l}^{i}(\frac{C_j^{on}}{S_j^{(i)}} + C_j^{off}) + (D_{i+1} - D_i). \tag{21}$$

From Lemmas 4.2 and 4.3, we know that all tasks from l to i have their speeds increased after adding task $i+1$ and have the same power rate. That is, there exists a $\lambda_{i+1}^{(i+1)}$ such that $G_j(S_j^{(i+1)}) = \lambda_{i+1}^{(i+1)}$ for any j, $l \leq j \leq i+1$. Substituting $S_j^{(i+1)}$ as $G_j^{-1}(\lambda_{i+1}^{(i+1)})$ in (21) completes the proof. □

4.1.2 An Iterative Speed Assignment Algorithm

This subsection develops an iterative algorithm to find the optimal speed assignments. According to Theorem 4.1, the optimal solution is to obtain l and $\lambda^{(i+1)}$ such that (20) holds. A straight-forward approach is to try every task from 1 to i and solve for $\lambda^{(i+1)}$; but this is not guaranteed to have a solution. If we increase $\lambda^{(i+1)}$ up to its next power rate $G_{l-1}^{(i)}$, the overall execution time of tasks from l to $i+1$ becomes smaller than D_{i+1}, as shown in Figure 1(a) by the dotted line started from task $l-1$. On the other hand, the time exceeds D_{i+1} if $\lambda^{(i+1)}$ takes the value $G_l^{(i)}$. Under the desired $\lambda^{(i+1)}$, task $i+1$ finishes at D_{i+1}, leading to the schedule in Figure 1(b).

Based on this observation, we present an iterative procedure in Algorithm 2, which determines speed settings of n aperiodic tasks with only feasible conditions. In the ith iteration, we have i power rates in a non-increasing order and try to add task $i+1$. For notational convenience, we assume all speeds initialized as 0, $G(0) = 0$, and $1/0 = \infty$. We find the minimum index l using a



(a) Schedule of the first i tasks

(b) Schedule after adding task $i+1$

Figure 1. Getting speed schedule of the first $i+1$ tasks based on the schedule of the first i tasks

binary search such that speeds of tasks from l to i increase. The value of $\lambda^{(i+1)}$ is then computed according to Theorem 4.1 and used to determine the speed settings.

Algorithm 2 Frequency-aware system-wide energy minimization with feasible constraints (FASTER-AF).

1: $S_1^{(1)} = C_1^{on}/D_1$
2: **for** $i = 1$ to $n-1$ **do** ▷ adding task $i+1$ at the ith iteration
3: $l = 1$ ▷ set l to 1 if binary search does not return a value
4: Using a binary search, find $l \in [2, i+1]$ such that
$\sum_{j=l-1}^{i+1} \frac{C_j^{on}}{G_j^{-1}(G_{l-1}(S_{l-1}^{(i)}))} - \sum_{j=l-1}^{i} \frac{C_j^{on}}{S_j^{(i)}} \le D_{i+1} - D_i - C_{i+1}^{off} <$
$\sum_{j=l}^{i+1} \frac{C_j^{on}}{G_j^{-1}(G_l(S_l^{(i)}))} - \sum_{j=l}^{i} \frac{C_j^{on}}{S_j^{(i)}}$
5: Solve for $\lambda^{(i+1)}$ in
$\sum_{j=l}^{i+1} \frac{C_j^{on}}{G_j^{-1}(\lambda^{(i+1)})} = \sum_{j=l}^{i} \frac{C_j^{on}}{S_j^{(i)}} + (D_{i+1} - D_i - C_{i+1}^{off})$
6: $S_j^{(i+1)} = G_j^{-1}(\lambda^{(i+1)}), l+1 \le j \le i+1$
$S_j^{(i+1)} = S_j^{(i)}, 1 \le j \le l$
7: **end for**

In Algorithm 2, computing sums in the binary search takes $O(n)$ time. Since there are $O(logn)$ searches, each iteration takes $O(nlogn)$. The running time of the algorithm is then $O(n^2 logn)$. The time complexity can be improved by noting that the sums inside the binary search only need to be done once. We can move the sum out of the search loop and maintain an array of sums, which reduce the complexity of the binary search loop to $O(logn)$. The modified algorithm is listed in lines 1-15 of Algorithm 3. The computation of the sums and other statement during each iteration can be finished in $O(n)$ time. The overall algorithm takes $O(n^2)$. As the algorithm actually returns speeds of n tasks, the average time needed for one task is bounded by $O(n)$.

4.2 Optimization with Upper and Lower Bounds

The solution to FASTER-AF presented in Section 4.1 only considers the feasible conditions (12) and (13). The optimal solution to FASTER-A needs to satisfy the boundary condition (14) as well. If the solution FASTER-AF satisfies the condition, then it is optimal; otherwise, we need to adjust the solution accordingly.

A tempting idea is to find the tasks whose speeds according to FASTER-AF violate the boundary conditions and set them to the corresponding violated bounds. However, the resultant solution does not necessarily satisfy the feasible constraints (12) and (13). A general approach is to deal with the upper and lower bounds separately. For example, Aydin *et al.* proposed an algorithm to reward maximization for periodic tasks with both service time upper and lower bounds [3]. It can find solutions under either bound in n iterations. Each iteration adjusts the speed of one task that violates its bound most. We can adapt their approach to take into account boundary conditions in FASTER-A; but this will increase the running time by a factor of n^2 with an overall complexity of $O(n^4)$. By exploiting inherent properties of online aperiodic tasks scheduling, we can develop an algorithm that satisfies the boundary conditions more efficiently.

We note that feasibility of a task is not affected by tasks that finish later. Consider task i for example. As long as the overall execution time of the first i tasks is no larger than its deadline D_i, the schedule is feasible for task i no matter when and under what speeds tasks from $i+1$ to n are executed. This enables us to guarantee the boundary conditions (14) one at each iteration. If speed of the first task returned from FASTER-AF is out of bound (upper or lower), we set its speed to the bound, remove the task from the task set, and reinvoke Algorithm 2 for the remaining tasks. Because feasibility of tasks that complete earlier is not affected in each step, we get a schedule satisfying both the feasible and boundary conditions after all iterations. In the worst case, no speed assigned from FASTER-AF satisfies the boundary conditions and Algorithm 2 are invoked for $n-1$ times.

We can further improve the average running time. Note that the presented algorithm returns a schedule for all n tasks and the algorithm is online without assuming future task releases at the time of speed assignment decision. If there is a new task release during execution of the n tasks, the schedule needs to be computed again including the new task. In this case, it is not necessary to schedule all tasks at first and reschedule them in a short time. Instead, it is enough to determine the speed of the first task. If there are no new task releases after the first one finishes, the speed assignment algorithm is applied again to determine speed of the next task. This does not change the worst case computation time in which the algorithm is invoked for $n-1$ times for n tasks. However, the partial speed assignment can reduce the actual scheduling complexity when there are many task releases. The time complexity of determining the speed of the first task among n tasks is $O(n^2)$, with the pseudo code listed in Algorithm 3. For completeness, we include the precomputation of execution time sums to reduce time complexity as described in Section 4.1.

5. Experimental Results

We evaluated the effectiveness of the proposed algorithms in energy savings for both periodic and aperiodic tasks. The evaluation was conducted based on the ADS BitsyXb platform [1]. The platform contains an Intel XScale PXA270 microprocessor and a number of other functional modules, including memory, networking interface, and display. The processor can operate in frequencies ranging from 13MHz to 512MHz. We ran five embedded benchmark programs from [8], including JPEG, CRC32, MAD, FFT, SHA, as well as a common Unix utility gzip.

Algorithm 3 Optimal frequency-aware system-wide energy minimization for online aperiodic tasks with feasible / boundary conditions (FASTER-A).

1: $S_1^{(1)} = C_1^{on}/D_1, A_1 = D_1, B_1 = 0$
2: **for** $i = 1$ to $n-1$ **do** ▷ adding task $i+1$ at the ith iteration
3: $A_{i+1} = \infty, B_{i+1} = 0$
4: **for** $l = i$ downto 1 **do**
5: $A_l = A_l + \frac{C_{i+1}^{on}}{G_{i+1}^{-1}(G_l(S_l^{(i)}))}, B_l = B_{l+1} + \frac{C_l^{on}}{S_l^{(i)}}$
6: **end for**
7: Using a binary search, find $l \in [2, i+1]$ such that
$A_{l-1} - B_{l-1} \leq D_{i+1} - D_i - C_{i+1}^{off} < A_l - B_l$
8: **if** $A_1 - B_1 > D_{i+1} - D_i - C_{i+1}^{off}$ **then** $l = 1$ **end if**
9: Solve for $\lambda^{(i+1)}$ in $\sum_{j=l}^{i+1} \frac{C_j^{on}}{G_j^{-1}(\lambda^{(i+1)})} = B_l + (D_{i+1} - D_i - C_{i+1}^{off})$
10: $S_j^{(i+1)} = G_j^{-1}(\lambda^{(i+1)}), l+1 \leq j \leq i+1$
$S_j^{(i+1)} = S_j^{(i)}, 1 \leq j < l$
11: $A_{i+1} = C_{i+1}^{on}/S_{i+1}^{(i+1)}$
12: **for** $j = i$ downto l **do**
13: $A_j = A_{j+1} + C_j^{on}/S_j^{(i+1)}$
14: **end for**
15: **end for**
16: **if** $S_1^{(n)} < S_1^{crt}$ **then** ▷ return speed of the first task
17: $S_1^{(n)} = S_1^{crt}$
18: **else if** $S_1^{(n)} > 1$ **then**
19: $S_1^{(n)} = 1$
20: **end if**

5.1 Measurement of On-chip/Off-chip Workload

We first measured on-chip and off-chip execution times of the applications. We utilized hardware performance counters provided by the PXA270 processor as indicators of on-chip and off-chip running times. The hardware counters collect data from the CPU clock cycles and 15 performance events. The number of off-chip cycles are caused by both data and instruction cache misses. The number of cycles can be captured by the following two event counters:

- CYCLES_DATA_STALL, which is a result of data dependency. This is a major source of CPU stall. Among all the applications we tested, 72%-93% of all CPU stall cycles are caused by cache misses due to data dependency.
- CYCLES_IFU_MEM_STALL, which occurs when instruction fetch pipe is stalled.

The counters are read and kept in memory before and after each benchmark application run. Their differences are returned and the sum of the two stall cycles are the total number of off-chip cycles. Meanwhile, the number of on-chip cycles can be obtained by reading the CPU cycle counter.

We list the number of on-chip and off-chip execution cycles of different applications in Table 1. Each reported value is an average of 6 runs. The ratios of off-chip and on-chip execution cycles are also included to show the application characteristics. From the column, we can see that SHA is the most CPU-intensive because of its smallest ratio; in contrast, gzip is the most I/O-intensive.

5.2 Measurement of Power Consumption

We measured power consumption of the system under different frequency settings. The PXA270 processor has 6 different voltage and frequency levels. (There are 7 voltage and frequency levels for a PXA270 processor. But the particular processor in the BitsyXb product only supports 6 levels.) In addition to a processor, the BitsyXb platform consists other modules such as memory, flash, NIC, UART, backlight inverter, etc. In our measurement, we detached or shut down all unused peripherals and only kept CPU, memory, flash, and UART on. We connected to the platform through serial port using UART and ran applications stored in on-board flash. We used a standard digital multimeter to measure the input voltage, which was relatively stable at 12.65 Volt. It was not feasible to employ multimeters for measuring current since their precision is low and coarse-grained. We used a sense resistor of low resistance to determine the current level of the system. The voltage drop was measured across the sense resistor and sent into a Data Acquisition Card. A National Instruments PCI-6024E DAQ card was used to sample data at a sampling rate of 1000/sec.

The active and standby system power in (2) can be obtained by running CPU-bounded applications under different frequencies. It is hard to measure the active power of the components other than CPU because it needs full run time knowledge of all the components, such as the exact time a module is accessed. We note that only the derivative (for example in the power rate function $G_i(S_i)$) of the energy function (2) is used in the speed assignments. As the active power of a module is independent of CPU speed, the active power becomes zero in the derivative. We therefore do not need to consider active power of the modules other than CPU.

Figures 2 and 3 show the system active and standby power consumed by the BitsyXb platform under different speed settings. We used the least square curve fitting to find approximated power functions. It is surprising to see both power consumption can be well fitted by cubic functions, as shown in the figures. Based on the two power functions, we computed the energy consumption during on-chip and off-chip execution and the overall system energy consumption. Figure 4 presents an example of the energy consumed in running the gzip application. We can observe the speed 0.38 leads to the minimum energy consumption. This is the critical speed for the system to run the gzip application. Critical speeds of other programs were obtained similarly by computing derivatives of their energy functions. We listed them in the last column of Table 1.

For the implementation of algorithms, there is still a need to compute the power rate function $G_i(S_i)$ and its inverse function $G_i^{(-1)}(\lambda)$ of an application. The power rate function is readily available with known measured application on-chip/off-chip execution times and system active/standby power. After that, its inverse function can be obtained analytically.

5.3 Simulation Results

We developed a simulator based on data measured in the BitsyXb platform with hard real-time scheduling using EDF. We first evaluated the effectiveness of the proposed algorithms for periodic tasks. We varied the system utilization from 0.1 to 1 each with 50 task sets. Every task set contains 10 periodic tasks randomly chosen from the benchmarks. Task periods were randomly chosen under the total utilization. We evaluate performance of the following algorithms:

- No-DVS, in which the processor is always run at the maximum speed during task execution.
- FAST, the scheme by Seth *et al.* [17] for processor energy optimization where tasks were run at the minimum feasible speed $\frac{\sum_{i=1}^{n} C_i^{on}/T_i}{1-\sum_{i=1}^{n} C_i^{off}/T_i}$.
- duSys, a heuristic algorithm for system-wide energy optimization by Zhuo and Chakrabarti [24], which sets the CPU speed to the larger value of system utilization and critical speed.

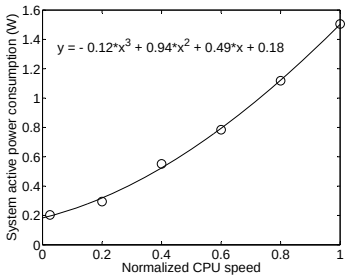


Figure 2. A cubic curve-fitting of system-wide active power consumption of the BitsyXb platform.

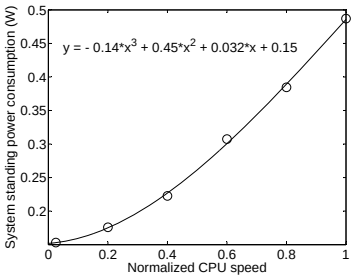


Figure 3. A cubic curve-fitting of system-wide standby power consumption of the BitsyXb platform.

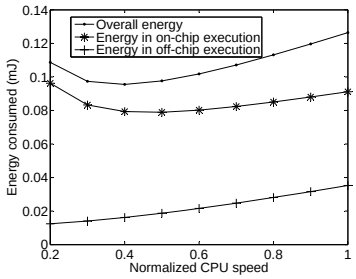


Figure 4. Energy consumption (on-chip and off-chip) in running gzip program.

Table 1. Execution cycles ($\times 10^6$) and critical speeds of benchmark applications.

Benchmarks	on-chip cycles	data stall	instruction stall	off-chip cycles	off-chip/on-chip	critical speed
gzip	3.13	2.65	1.02	3.67	1.17	0.38
JPEG	2.81	1.50	0.11	1.61	0.57	0.47
MAD	2.83	1.17	0.32	1.49	0.53	0.48
CRC32	3.08	0.92	0.13	1.05	0.34	0.56
FFT	4.49	1.11	0.12	1.23	0.27	0.60
SHA	4.65	0.69	0.05	0.74	0.17	0.68

- FASTER-P, the proposed algorithm for frequency-aware system-wide optimization for periodic tasks.

We simulated execution of the generated task sets and collected average energy consumption taken in a hyper-period. Figure 5 shows the normalized energy consumption due to different algorithms with respect to No-DVS. We can observe that FAST consumes much more energy with low utilization values. This is because it only considers processor energy minimization and runs the CPU at a speed lower than the critical speed of an application. Both duSys and FASTER-P operate the CPU at a speed no lower than the critical speed for an application and can cut the energy consumption of FAST almost in half. With higher system utilization, FAST outperforms duSys because it explicitly considers the impact of off-chip execution. FASTER-P consistently performs the best in all cases because it simultaneously considers the existence of critical speed and off-chip workload. It is more efficient than duSys (up to 14%) by using the optimal speed settings. duSys uses a uniform speed for all tasks, which is only a special case of FASTER-P when all tasks have the same power functions. i.e., all tasks have the same resource usage and the same off-chip/on-chip ratios. All the algorithms lead to similar energy consumption when the system becomes heavy utilized.

We also evaluated the proposed algorithm for online aperiodic tasks. We compared performance of the following algorithms:

- No-DVS, in which the processor is always run at the maximum speed during task execution.
- DVSST, an algorithm for online aperiodic tasks by Qadi *et al.* [16] which minimizes CPU energy consumption when only speed scaling (no voltage scaling) is available.
- TVDVS, an adaptive optimal algorithm for online aperiodic tasks by Zhong and Xu [20] where task assignment was based on CPU energy consumption.
- FASTER-A, the proposed algorithm on frequency-aware system energy minimization for online aperiodic tasks.

We experimented with the benchmark applications with different task inter-arrival times. The minimal interarrival time was set to 60ms. To avoid trivial infeasible cases even under No-DVS, we randomly set the maximum utilization of each task under the constraint that the overall utilization of all tasks at their maximum release rates does not exceed one. Interarrival times of a task were chosen from an exponential distribution with averages ranging from 0.1s to 1s. Figure 6 shows normalized system energy consumption of the algorithms. As the task releases are irregular, the system cannot be simply characterized by utilization. We present the energy with respect to interarrival times instead of utilization. When the task interarrival time is large, FASTER-A can save more than 40% energy than TVDVS and DVSST due to the consideration of system standby power. The performance gap reduces with more frequent task releases. But FASTER-A can still perform up to 22% better by taking into account task off-chip workload and different power characteristics of applications.

We measured the time complexity of the algorithms with an increased number of tasks in Figures 7 and 8. The values were taken in a Windows machine with a 2GHz Pentium 4 processor. We get each value by averaging over 50 consecutive runs. We set the termination threshold and computation precision as 10^{-10} for FASTER-P. With all the task numbers tested, FASTER-P terminates very fast with less than 25 iterations under 18ms, as shown in Figure 7. In fact, the scaled processor utilization becomes less than 0.1% that of the final value after 8 iteration steps. Running time of the algorithm shows a roughly linear relation with the number of tasks. This makes the algorithm especially useful for task set with a large problem size. Running times of FAST and duSys were also recorded and presented in Figure 7. As the time is too small to measure, we ran the algorithms 50 times and took their averages. Both algorithms finish quickly within 1ms for all the task numbers.

We also compared the running times of the online algorithms for aperiodic tasks. Figure 8 presents the running times in generating speed schedules of a 10 minutes run with an average 200ms interarrival time. Tasks were randomly chosen from the benchmark applications given each task number. Because TVDVS and DVSST

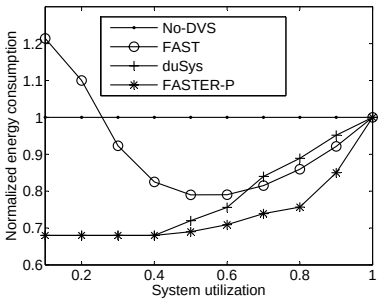


Figure 5. System energy consumption of periodic tasks.

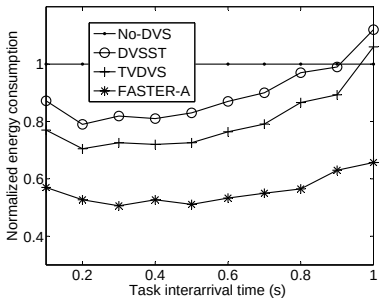


Figure 6. System energy consumption of aperiodic tasks.

have a linear time complexity, their running times are consistently smaller than FASTER-A. The running time of FASTER-A grows faster. However, the overall running time for the algorithm is still small compared to the task execution time. For example, the algorithm running time with 100 tasks is 2.34s, only 0.4% of task execution time. The simulation results suggest that both FASTER-P and FASTER-A provide a better trade-off between energy and time complexity than existing approaches.

6. Related Work

Extensive studies on energy savings have been conducted for periodic tasks using DVS. An emphasis was on reducing dynamic portion of processor energy consumption. There are several initial studies on system-wide energy optimization [5, 11, 19, 4, 24]. Choi *et al.* proposed an interval based frequency setting policy that would minimize system-wide energy consumption of a program subject to a constraint on performance loss in terms of increased execution time [5]. Their approach can be best applied to applications with soft deadlines.

Jejurikar and Gupta considered periodic tasks on a processor with discrete speed levels [11]. Energy optimization under discrete speeds is NP-hard in general [14] and no polynomial time algorithm exists to get the optimal solutions. The authors proposed a heuristic algorithm to approximate the system-wide energy optimization problem. The approach was recently extended by Zhong and Xu with an optimal and approximated solutions in [19]. Cheng and Goddard studied system-wide energy savings of periodic tasks with a focus on the impact of non-preemptive shared resources [4].

Although most commercially available processors have a limited number of speed levels, researchers have shown that given any speed, it can be mapped to its two neighbors [10], [12]. This allows the processor to run at any speed. Zhuo and Chakrabarti studied system-wide energy optimization on a processor with continuous speed levels and proposed a static speed setting policy, with extensions to online slack distribution and preemption control [24]. In their static policy, they started from a uniform speed setting for all tasks and adjusted it for each task according to its critical speed. Although the algorithm is more efficient than traditional DVS, it remains unclear how good the algorithm is and whether there exists a solution with more energy savings. In this paper, we proposed an algorithm to get the optimal solution on a processor with continuous speed levels.

More recently, Aydin *et al.* studied system-wide energy minimization for periodic tasks on a processor with continuous speed levels [2]. They separated task execution into on-chip/off-chip times, which is best applicable for CPU and memory. System energy was represented in the form of $af^m + b$, where a, b, and m ($m \geq 2$) are constants. They proposed an algorithm with a time complexity of $O(n^3)$ where n is the number of tasks. By contrast, we restrict the analysis to CPU and memory only and do not assume any specific forms of power function and propose a more efficient algorithm with $O(n)$ running time. In addition to periodic tasks, we also proposed optimal solution for frequency-aware online scheduling of real-time aperiodic tasks. The algorithm is optimal among all online speed assignments without assumed task information before task releases.

DVS algorithms in support of online aperiodic tasks are another research focus [9, 16, 18, 13, 20]. For examples, Sharma *et al.* investigated the use of DVS in web servers [18], Qadi *et al.* exploited energy saving potentials in processor with only frequency scaling support [16], Lee and Shin focused on online slack management [13]. However, they all targeted at dynamic energy consumption of a processor. A related study on system-wide energy optimization for sporadic tasks (A special type of aperiodic tasks in which there is a minimum inter-releases time between two consecutive release of a task.) proved that the online minimization is NP-Hard in the strong sense, which means any algorithms for the optimal solution have strictly exponential running time [19]. For tractability, we consider system-wide energy minimization for aperiodic tasks by considering a processor with continuous levels. Furthermore, most DVS algorithms assume worst-case execution cycles of a task does not change with processor speed. This is not accurate since applications involve operations outside the processor, which is processor speed independent. By contrast, we take into account the impact of speed scaling on task execution cycles.

7. Conclusion

We develop a system-wide energy minimization approach to both periodic and aperiodic tasks in an embedded system whose power consumption is dominated by CPU and memory. It is frequency-aware in the sense it explicitly considers the impact of frequency scaling on the number of task execution cycles. It distinguishes the cycles between on-chip in CPU and off-chip in memory. The energy minimizations are formulated as constrained optimization problems and conditions are established for the optimal solutions. While it is time consuming to find the solutions satisfying all the conditions, we analyze properties of the conditions and propose efficient algorithms to satisfy the conditions. For periodic tasks, we develop a bisection algorithm to find the optimal solution. Experimental results demonstrates the applicability of the proposed approach in a practical embedded system. Simulation results based on data measured in the platform show that the algorithm can cut the energy consumption in half compared to processor energy optimization by FAST [17]. It improves up to 14% over a recent system-wide approach *duSys* [24]. The bisection algorithm has a linear converging speed in the number of tasks. It is especially useful for tasks with large problem size. For aperiodic tasks, we propose an iterative algorithm that assigns speed online to a task

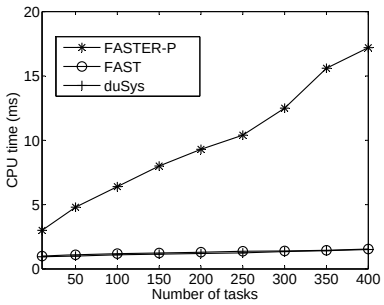


Figure 7. Algorithms running time for periodic tasks.

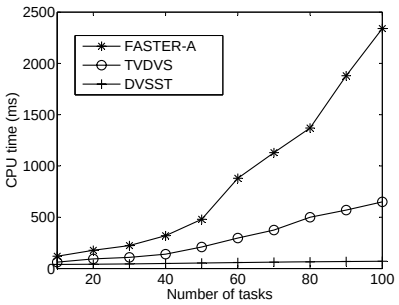


Figure 8. Algorithms running time for aperiodic tasks.

according to unfinished tasks. We prove it is online optimal in the sense it assumes no timing information about future task releases. The time complexity of determining the speed of the first task among n tasks is $O(n^2)$. The task number n is much lower than the number of total tasks in the systems because only active tasks are considered in each speed assignment. Evaluation results show that considering the impact of system standby power and off-chip workload can improve over recent algorithms on dynamic energy conservation of processors (TVDVS [20] and DVSST [16]) by as much as 40%.

Our on-going work is to implement the proposed algorithms in the BitsyXb system. We are trying to extend the proposed algorithm in this paper to consider some practical issues such as voltage switching overhead both in energy and time.

Acknowledgments

We would like to thank our shepherd, Frank Mueller, and the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by U.S. NSF grants ACI-0203592, CCF-0611750, and NASA grant 03-OBPR-01-0049.

References

[1] ADS. *ADS BitsyXb Platform*. http://www.applieddata.net/.

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

[3] H. Aydin, R. G. Melhem, D. Mossé, and P. Mejía-Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Trans. Computers*, 50(2):111–130, 2001.

[4] H. Cheng and S. Goddard. Integrated device scheduling and processor voltage scaling for system-wide energy conservation. In *Proc. of the Int'l Workshop on Power-aware Real-time Computing*, pages 24-29, 2005.

[5] K. Choi, W. Lee, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. In *Proc. of the Int'l Conf. on Computer-Aided Design*, pages 29–34, 2004.

[6] J. K. Dey, J. F. Kurose, D. F. Towsley, C. M. Krishna, and M. Girkar. Efficient on-line processor scheduling for a class of iris (increasing reward with increasing service.) real-time tasks. In *Proc. of the ACM SIGMETRICS Conference*, 1993.

[7] L. R. Foulds. *Optimization Techniques: An Introduction*. Springer-Verlag, 1981.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE Annual Workshop on Workload Characterization*, 2001.

[9] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage voltage processor. In *Proc. of the Int'l Conf. on Computer-Aided Design*, pages 653–656, 1998.

[10] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of the Int'l Symp. on Low-Power Electronics and Design*, 1998.

[11] R. Jejurikar and R. K. Gupta. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *Proc. of the Int'l Symp. on Low-Power Electronics and Design*, pages 78–81, 2004.

[12] W.-C. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Trans. on Embedded Computing Sys.*, 4(1):211–230, 2005.

[13] C.-H. Lee and K. G. Shin. On-line dynamic voltage scaling for hard real-time systems using the edf algorithm. In *Proc. of the IEEE Int'l Real-Time Syst. Symp.*, pages 319–327, 2004.

[14] P. Mejía-Alvarez, E. Levner, and D. Mossé. Adaptive scheduling server for power-aware real-time tasks. *ACM Trans. Embedded Comput. Syst.*, 3(2):284–306, 2004.

[15] J. Pouwelse, K. Langendoen, and H. J. Sips. Application-directed voltage scaling. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(5):812–826, 2003.

[16] A. Qadi, S. Goddard, and S. Farritor. A dynamic voltage scaling algorithm for sporadic tasks. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 52–62, 2003.

[17] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. *ACM Trans. Embedded Comput. Syst.*, 5(1):200–224, 2006.

[18] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware qos management in web servers. In *Proc. of the IEEE Real-Time Systems Symp.*, pages 63–72, 2003.

[19] X. Zhong and C.-Z. Xu. System-wide energy minimization for real-time tasks: Lower bound and approximation. In *Proc. of the Int'l Conf. on Computer-Aided Design*, pages 516–521, 2006.

[20] X. Zhong and C.-Z. Xu. Energy-aware modeling and scheduling for dynamic voltage scaling with statistical real-time guarantee. *IEEE Trans. Computers*, 56(3):358–372, 2007.

[21] X. Zhong and C.-Z. Xu. Energy-efficient wireless packet scheduling with quality of service control. *IEEE Trans. Mobile Computing (Accepted)*, 2007.

[22] D. Zhu, R. G. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of the Int'l Conf. on Computer-Aided Design*, pages 35–40, 2004.

[23] Y. Zhu and F. Mueller. Feedback edf scheduling exploiting hardware-assisted asynchronous dynamic voltage scaling. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 203–212, 2005.

[24] J. Zhuo and C. Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Proc. of the Design Auto. Conf.*, 2005.