*Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*

# Periodic Charging Scheme for Fixed-Priority Real-Time Systems with Renewable Energy

Mario Bambagini[1,2] and Hakan Aydin[3]

[1]Scuola Superiore Sant'Anna, Pisa, Italy e-mail: *m.bambagini@sssup.it*
[2]University of Modena and Reggio Emilia, Italy e-mail: *mario.bambagini@unimore.it*
[3]George Mason University, Fairfax, VA, USA e-mail: *aydin@cs.gmu.edu*

***Abstract*—Energy harvesting systems are gaining increasing importance in the embedded systems domain, as they provide an effective solution to bridge the gap between the energy supply and demand. However, the variable nature of the energy supply rate due to the environmental conditions creates serious challenges for embedded real-time systems targeting predictability.**

**This paper presents a proactive and highly predictable framework, called Periodic Charging Scheme (PCS), for fixed-priority real-time systems with renewable energy. The main idea of the algorithm is to plan in advance for periodic charging and discharging of the battery to avoid energy outage, while still meeting the timing constraints. The algorithm is specifically designed to exploit the low-power states of modern processors, to enable effective power state transitions when the battery is recharged on a periodic basis. We also offer online enhancements to opportunistically extend the duration of charging phases and improve the responsiveness of the potential non-real-time workloads without compromising feasibility. Extensive simulations show that the proposed approach outperforms the state-of-the-art algorithms in terms of the number of task sets that meet the timing constraints under specified energy profiles, when a realistic power model with state transition overheads is assumed.**

## I. Introduction

Resource management for real-time embedded systems, that offer predictability in terms of timing constraints, has always been a prime research and development area. In the last decade, *power-aware* resource management and scheduling for embedded systems has received increasing attention. One of the major factors in this trend is the proliferation of small footprint devices that rely on battery power, which is fairly limited in practice. Hence, several power management techniques to save energy at run-time and extend the lifetime of the system before it runs out of energy were proposed. Among these techniques are *Dynamic Power Management* ([1], [2], [3]) and *Dynamic Voltage and Frequency Scaling* ([4], [5]) approaches that exploit low-power sleep states and low-power/low-performance active states, respectively.

More recently, there has been a growing interest in *energy harvesting* systems that scavenge energy from the environment. Most commonly, the deployed systems use solar panels and piezoelectric units, that exploit solar energy and mechanical energy generated by vibrations, respectively. The harvested energy is stored in a re-chargeable battery or supercapacitor for future use. With energy harvesting capability, energy becomes a resource which can be *replenished* in quasi-continuous manner. The power generation activity is typically predictable (for example, considering the time of the day and season in the case of solar energy); but its rate of supply is not necessarily uniform: the system is not able to harvest solar energy at night time, and energy harvesting rate will vary during the day. This *predictably non-uniform* energy availability is characteristic of energy harvesting systems, and adds a new dimension to the power-aware system design. An additional complexity is related to the limited capacity of the energy storage unit. With energy harvesting, in theory it becomes possible to design and build *energy-neutral* systems [6]: the systems that manage their energy consumption activities in such a way that they can perpetually sustain their operation, subject to the hardware faults/longevity only. In an energy-neutral system, over any time interval $[0, t]$, the consumed energy should not exceed the available energy, which is the harvested energy augmented by the initial energy reserves.

In the real-time embedded systems area as well, researchers have recently started to investigate the impact of adding energy harvesting dimension to the existing frameworks. In those settings, the real-time scheduling objectives have to consider two separate resource supply dimensions simultaneously: *time*, which is available at uniform rate, and *energy* which is supplied by the environment at a time-varying rate. As a result, several solutions have been proposed for both dynamic-priority and fixed-priority systems. In general, energy-harvesting algorithms apply *task procrastination* as the conditions warrant: for instance, due to the *current* low energy level, or as a way to proactively prevent a *future* energy shortage. Based on this distinction, we can broadly divide the existing algorithms into *energy-greedy* and *computation-greedy* classes depending on the actions they take when the energy level is low. Under that condition, the *energy-greedy* algorithms exploit the available slack in the system by procrastinating tasks and charging the battery as much as possible. In contrast, the *computation-greedy* algorithms give priority to execute the pending workload, and charge the battery only when there is no sufficient energy to execute tasks.

For fixed-priority real-time embedded systems which are more common in practice, the two well-known algorithms

This work has been supported by the European Commission under the P-SOCRATES project (FP7-ICT-611016), and in part, by US National Science Foundation awards CNS-1016855 – CNS-1016974

are $PFP_{st}$ (also called $EDeg$) [7] and $PFP_{asap}$ [8], that represent energy-greedy and computation-greedy algorithms, respectively. In particular, $PFP_{asap}$ is shown to be optimal in [8]: any task set that can be feasibly scheduled by any other fixed-priority energy-harvesting scheduling algorithm can be also scheduled by $PFP_{asap}$. On the other hand, the same paper shows that in terms of the preemption numbers and other run-time overhead metrics, $PFP_{st}$ has a clear advantage, while its average feasibility performance lags behind $PFP_{asap}$ by a small margin.

Our work is partly inspired by the observation that, despite their theoretical importance, the existing algorithms assume power models that do not fully comply with existing processors: for instance, it is assumed that the CPU can be switched to/from a low-power (sleep) state instantaneously and without any energy overhead. Moreover, there is only one sleep state and its power consumption is negligible. Contemporary processors have typically multiple low-power states, each with different power/transition overhead characteristics, such as *standby, idle,* and *deep sleep* states. In general, the lower the power consumption in a state, the higher the transition overhead, and there is a minimum idle time interval that must be guaranteed before switching to a low-power state, called *break-even time* [9]: if the system has to switch back to active state sooner, then the overall energy consumption *increases*, negating all the benefits of low-power states. Another objective is to develop a simple, proactive and highly-predictable framework that seamlessly integrate the energy-harvesting capabilities into existing and widely known fixed-priority systems.

**Paper contributions.** This paper proposes a novel and highly-predictable energy management algorithm for fixed-priority real-time systems with renewable energy, called *Periodic Charging Scheme (PCS)*. The main idea consists of planning in advance for periodically alternating charging and discharging phases to avoid battery failures. Specifically, the task execution is suspended periodically and for a pre-determined duration, to allow the system to re-charge the battery. At design time, the algorithm computes the duration of the phases, taking into account the characteristics of tasks and the embedded platform, as well as the break-even times of the existing low-power states. At runtime, the algorithm opportunistically extends the duration of the charging states whenever possible, to further increase the energy level. Moreover, we also provide an enhancement to increase the spare bandwidth that can be used by non real-time aperiodic tasks, if included in the workload, without affecting the overall feasibility.

We perform extensive simulations to compare the performance of $PCS$ against the state-of-the-art techniques. We show that when realistic power parameters are considered, $PCS$ outperforms other techniques in terms of *feasibility ratio*, which is the percentage of the task sets that are feasibly scheduled with the given energy profiles. We also evaluate several other performance indicators, such as the number of preemptions and the length of the average sleep intervals.

**Paper organization.** Section II introduces the power and workload models that are used in this paper. Section III gives the details of the proposed approach. Section IV compares our algorithm with the state-of-the-art techniques, experimentally. In Section V, we overview the closely related work. Finally, Section VI concludes the paper with final remarks.

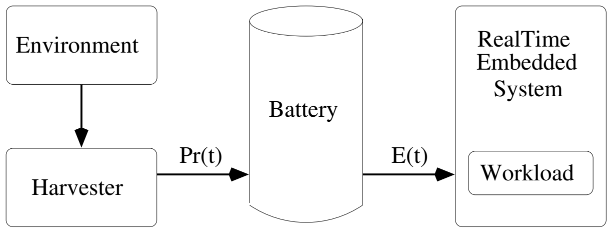


Fig. 1. Components involved in the energy flow.

## II. System model

### A. Power model

As depicted in Figure 1, we assume a system with energy harvesting capability. The *harvester unit* is in charge of scavenging energy from the environment and storing in the *energy storage unit* (which may be a battery or supercapacitor). The energy available to the embedded system at time $t$ is denoted by $E(t)$. This energy level is bounded by $C$, which is called the *battery capacity*. The energy is harvested at the rate of $P_r(t)$. As common in energy harvesting research ([6], [10], [11]), we assume that the operation interval is divided into equal length *time slots* (or, *epochs*): in each time slot (of duration $T_r$) the energy harvesting/replenishment rate can be assumed to be constant. For example, several papers assumed various $T_r$ values, ranging from 15 to 60 minutes.

The power consumption of the processor in the *active* state (denoted as $\sigma_0$) is given by a constant value $P_{CPU}$ that accounts for both dynamic and leakage dissipation. The overall power consumption due to the entire set of remaining system components (e.g., I/O devices/peripherals) is denoted by $P_{dev}$. The total power $P_{\sigma_0}$ consumed by the system in the active state is the sum of processor power and total power consumption of the specific subsets of peripherals in use by the running task; thus, at any time, $P_{CPU} \leq P_{\sigma_0} \leq P_{CPU} + P_{dev}$. As in [7], [8], we do not assume DVFS capability in the system; i.e., task execution takes place at a constant frequency level.

A feature provided by almost all processors consists of a set $\Phi = \{\sigma_1, \ldots, \sigma_m\}$ of *low-power (sleep) states*, in which the task execution is suspended and the system enters a mode with reduced power consumption. Typically, processors support multiple states with different nomenclature, such as *idle, standby* and *deep sleep* states. Moreover, each low-power state $\sigma_k \in \Phi$ is characterized by a power consumption level $P_{\sigma_k}$ and a *break-even time* $\xi_{\sigma_k}$. Specifically, the break-even time is the length of the shortest idle interval that must be available in the schedule to switch to, and later back from, that specific low-power state ([3], [9]). This is necessary to amortize the time and energy overheads associated with that transition. In general, when the CPU is idle, the system should try to switch to the *deepest* low-power state whose break-even time is shorter than the predicted length of the idle interval.

Given the above notation, if the system remains in state $\sigma_k$ (which may be the *active* state $\sigma_0$ as well as one of the *low-power* states $\sigma_i$ $(i \geq 1)$) from time $t_1$ to $t_2$, then the battery energy level at time $t_2$ is expressed by:

$$E(t_2) = \min\left(C, \quad E(t_1) + \int_{t_1}^{t_2} (P_r(t) - P_{\sigma_k})\, dt\right).$$

### B. Task model

The workload consists of a set of $n$ real-time sporadic independent tasks, $\Gamma = \{\tau_1, \tau_2 \ldots, \tau_n\}$. Each real-time task $\tau_i$ consists of a infinite sequence of jobs $(\tau_{i,1}, \tau_{i,2}, \ldots)$ and is characterized by a minimum inter-arrival time $T_i$ (also called the *period*), a relative deadline $D_i$, a worst-case execution time (WCET) $C_i$, and power consumption $P_i$. Each task $\tau_i$ releases a new job $\tau_{i,j}$ sporadically at time $a_{i,j}$, meaning that the interval between two consecutive jobs arrivals is greater than or equal to $T_i$: $a_{i,j+1} \geq a_{i,j} + T_i$. We assume the common *implicit-deadline* systems in which the relative deadline is equal to the minimum inter-arrival time: $\forall \tau_i,\ D_i = T_i$.

$P_i$ represents the power consumed to execute a job of $\tau_i$, including both the processor and the peripherals ($P_{CPU} \leq P_i \leq P_{CPU} + P_{dev}$). The overall worst-case energy required by each job of $\tau_i$ is denoted as $E_i$ and it is computed as $E_i = P_i \times C_i$. Note that two tasks having the same WCET may consume different energy if they use different peripherals. $H$ denotes the *hyperperiod* of the task set, computed as the least common multiple of all the periods: $H = lcm(T_1, \ldots, T_n)$.

Finally, the real-time tasks are executed according to the *Rate Monotonic* scheduling policy. Tasks are indexed in decreasing priority order, so that $\tau_1$ is the highest priority task.

## III. Proposed approach

The proposed approach is based on a periodic scheme which alternates between active and inactive phases of the processor: the first one is in charge of executing the pending workload, while the second one replenishes energy until the next active phase.

The inactive phase is implemented by adding a new hypothetical periodic task ($\tau_s$) that puts the processor in a low-power state for an interval $C_s$ in every period $T_s$, in order to charge the battery continuously, without any interruption by other tasks. To this aim, the highest priority in the system is assigned to $\tau_s$; implying that whenever it is ready, the system will be put in a low-power state and continuous re-charging will be enforced in a predictable and periodic fashion. Moreover, its "execution time" $C_s$ is chosen in such a way that the system will be able to exploit the deepest possible low-power state offered by the platform, by considering the break-even times of the existing states, *while still guaranteeing the deadlines of real-time tasks*. In other words, our periodic charging scheme ($PCS$) provides both a predictable harvesting mechanism and an ability to comply with the requirements of the low-power states of the processor, in terms of the overhead amortization.

According to this framework, the problem can be reformulated as finding a valid pair of $C_s$ and $T_s$ which avoid deadline misses and energy failures while executing $\tau_s$ at the highest-priority level. Note that the assignment of the highest priority to $\tau_s$ is critical to enforce its "non-preemptive" execution, to enable the system to enter a low-power state effectively.

In systems with renewable energy, the concept of feasibility is extended to consider also *battery (or, energy) failures* ([7], [8]). Specifically, in addition to guaranteeing task completions no later than their respective deadlines, in order to ensure feasibility, the algorithm must also guarantee that the battery level never drops below a certain threshold $E_{low}$: $\forall t, E(t) > E_{low}$. Without loss of generality, we consider the case of $E_{low} = 0$; for higher thresholds, the battery capacity can be downsized accordingly and the problem can be re-stated as an instance with $E_{low} = 0$. Note that, if an energy failure happens, it may not immediately lead to a deadline miss as the required time to charge the battery may not violate real-time constraints. Our adopted definition is stricter than this interpretation: our proactive approach treats any energy underflow as a failure condition, which may indeed introduce unpredictability in real-time embedded system design.

Compared to the existing energy-greedy and computation-greedy energy harvesting algorithms (Section I), $PCS$ is conceptually much simpler and easier to implement with low online complexity. Moreover, thanks to its design principles, it explicitly considers the time/energy overheads involved in the processor state transitions, through the explicit analysis of the break-even times. In contrast, the energy-greedy algorithms (e.g., $PFP_{st}$ [7]) involve online computation of the existing slack to re-charge the battery, which is, in general, of pseudo-polynomial complexity. Similarly, the computation-greedy algorithms (e.g., the theoretically optimal $PFP_{ASAP}$) result in very frequent invocation and processor state transitions with prohibitive costs on real systems that have non-zero transition overheads.

An example of our approach is illustrated in Figure 2, showing how the battery level varies while executing the instances of $\tau_s$ and the workload. For the sake of simplicity, the replenishment function has been assumed constant and all tasks consume the same power. Since $\tau_s$ runs at the highest priority (in order to guarantee a non-preemptive execution), any time an instance of $\tau_s$ is released, the actual running job is preempted, the processor enters into a low-power state, and the battery is replenished. Then, the workload execution is resumed when $\tau_s$ instance completes its execution, running for at most $T_s - C_s$ time units before next instance arrives.

We first present the details of the proposed algorithm in Section III-A. Then, a sufficient condition is provided to test the system feasibility at design time (Section III-B). Finally, for workloads that may include non real-time components, an online enhancement is introduced in Section III-C to improve the responsiveness of such tasks, without affecting the feasibility of the real-time workload.

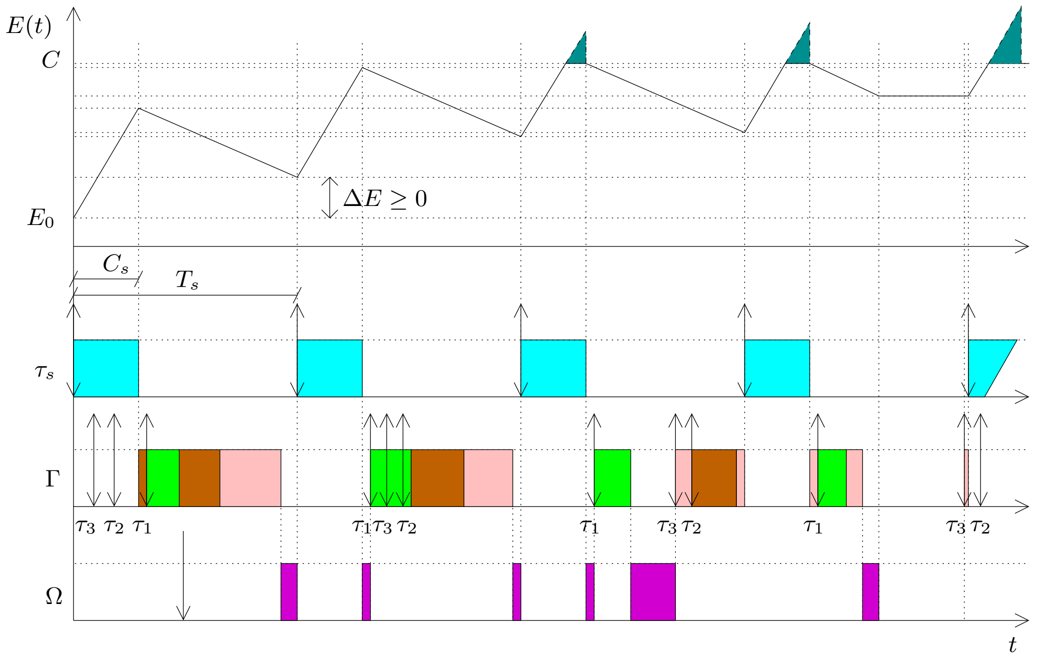


Fig. 2. Example of algorithm execution.

## *A. Algorithm*

This section gives the details of the proposed algorithm: *Periodic Charging Scheme (PCS)*. Specifically, at design time, the algorithm computes the period $T_s$ and charging time $C_s$ which lets $\tau_s$ execute periodically in a non-preemptive fashion. Then, at runtime, the algorithm opportunistically compacts idle intervals and $\tau_s$ execution, to further extend battery phases and exploit deeper low-power states.

First, let us consider the design-time step. The shortest period among the real-time tasks is assigned to $T_s$ in order to let $\tau_s$ have the highest priority and run in a non-preemptive way. The computational time $C_s$ is assigned according to the sensitivity analysis proposed by Bini et al. [12], which computes the highest spare utilization that $\tau_s$ can have, without causing deadline misses among the lower priority tasks. In this case, lower priority tasks correspond to the entire original task set $\Gamma$. Although the sensitivity analysis considers fully-preemptive tasks, the non-preemptive execution of $\tau_s$ is automatically guaranteed by its highest priority, without invalidating the analysis. Moreover, the specific low-power state $\sigma_k$ to which the system switches during the $\tau_s$'s execution is chosen as the deepest sleep state whose break-even time is shorter than or equal to $C_s$. The corresponding pseudocode is presented in Algorithm 1.

**Algorithm 1** $PCS$: Design-Time Algorithm

```
1: function PCS_AT_DESIGN_TIME($\Gamma$)
2:     $T_s = \min_{\tau_i \in \Gamma} T_i$
3:     $C_s = \Delta C_s$ /* From sensitivity analysis [12] */
4:     $\Gamma \leftarrow \Gamma \cup \{\tau_s\}$
5:     $k = \max_{\sigma_i \in \Phi \wedge \xi_{\sigma_i} \leq C_s} i$
6: end function
```

On the other hand, the runtime component of $PCS$ is executed whenever the ready queue becomes empty. Specifically, when the processor is idle, the algorithm first computes earliest possible next arrival time of any periodic task ($next\_arrival$), which can be easily computed given the minimum inter-arrival time information of the tasks. Then, it re-adjusts the next arrival time of $\tau_s$ to coincide with $next\_arrival$. In this way, the system enters an *extended* charging phase from the current time until $next\_arrival + C_s$, potentially enabling the exploitation of even deeper low-power states simultaneously.

To prove that the schedulability is not affected by re-adjusting the next invocation time of $\tau_s$ arrival, let us assume a generic task set whose feasibility is statically guaranteed. Recall that $\tau_s$ has the highest priority in the system. According to the well-known fixed-priority schedulability analysis techniques, the response time of any task is maximized when its job arrives simultaneously with the jobs of higher-priority tasks [12]. Since the task set is deemed feasible at the static phase, the response time of any task does not exceed its deadline even in that *critical instant*, by definition. Hence, by aligning the next invocation time of $\tau_s$ with the $next\_arrival$, other deadlines cannot be compromised. Then, forcing $\tau_s$ to arrive at the same time lets us obtain a configuration equivalent to the critical instant, whose feasibility is already assumed in the static analysis.

The pseudocode in Algorithm 2 gives the details of the runtime component of $PCS$, which computes the actual charge length ($T_{charge}$), adjusts $\tau_s$'s next invocation time $a_{s,j}$ and selects the deepest low-power state to use during that specific charge step.

An example is reported in Figure 3, representing the

**Algorithm 2** $PCS$: Runtime Algorithm

```
1: function PCS_AT_RUNTIME ($t$)    ▷ $t$: CPU becomes idle
2:     $t_1 = next\_arrival$
3:     $T_{charge} = C_s + (t_1 - t)$
4:     $a_{s,j} = t_1$
5:     $k' = \max_{\sigma_i \in \Phi \wedge \xi_{\sigma_i} \le T_{charge}} i$
6: end function
```

schedules without and with the runtime component of $PCS$. Specifically, when the runtime component is enabled, it is invoked at $t$ (when the CPU becomes idle) and, computing the next arrival time in $\Gamma$ as $t_1$, $\tau_s$'s execution and the idle interval are compacted to form a single longer interval (of duration $C_s + (t_1 - t)$). With longer intervals, the algorithm gains the ability to potentially exploit deeper low-power states.

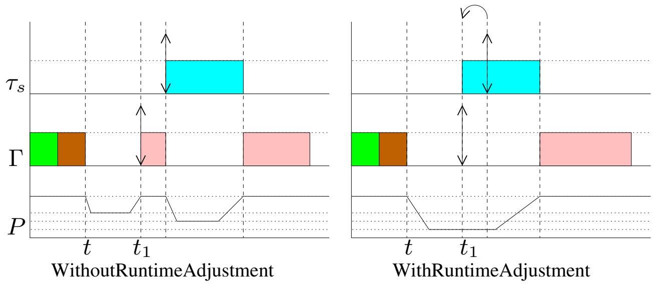


Fig. 3. Schedule examples without and with the $PCS$ runtime component.

At runtime, the introduced complexity for the scheduler is negligible as the algorithm only requires to schedule the additional task $\tau_s$. The complexity of the static (design-time) component is pseudo-polynomial with respect to the number of tasks due to the sensitivity analysis. Finally, the runtime component of $PCS$ has also low complexity: assuming the earliest next arrival time can be evaluated in constant time, then the overall complexity is linear with respect to the number of low-power states, which is $O(m)$.

## B. A Sufficient Condition for Schedulability

In $PCS$, the feasibility in terms of timing constraints is explicitly guaranteed through the sensitivity analysis. On the other hand, providing a *simple* necessary and sufficient condition to check whether a given system configuration, with a certain initial energy budget and harvesting profile is feasible or not, is not trivial.

Nevertheless, a sufficient condition to guarantee execution without energy failures can be derived, for design-time (offline) analysis. The condition is based on guaranteeing that, even in the worst-case scenario, the difference between the harvested energy and the consumed energy during one period $T_s$ of $\tau_s$ is not negative. If this holds, due to the periodic nature of $\tau_s$'s invocations, the energy level of the system will never decrease in the long run, guaranteeing feasibility.

Specifically, the difference in the energy levels at the beginning of two consecutive invocations of $\tau_s$ is given by:

$$\Delta E = (P_r - P_{\sigma_k}) \cdot C_s - (P_{act} - P_r) \cdot (T_s - C_s) \ge 0, \quad (1)$$

where $P_{act}$ is the maximum task power consumption in the active state ($P_{act} = \max_{\tau_i} P_i$) and $\sigma_k$ is the low-power state selected by the offline phase of $PCS$.

An intuitive example is shown in Figure 4, illustrating how the battery level $E$ varies while executing $\tau_s$ and real-time tasks. The first term in Eq. (1), $(P_r - P_{\sigma_k}) \cdot C_s$, gives the net energy gain during the charging phase, while the second term $(P_{act} - P_r) \cdot (T_s - C_s)$ corresponds to the energy loss during the discharging phase.

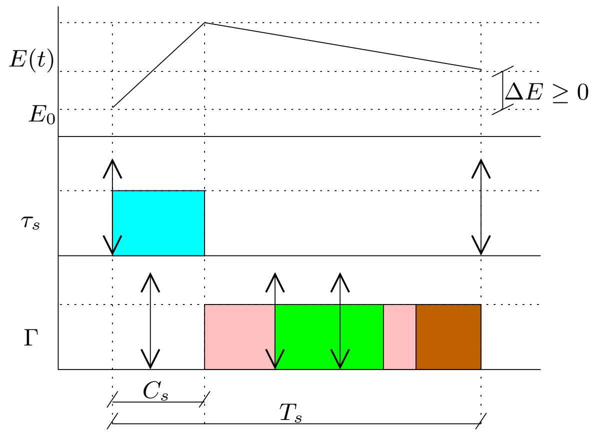


Fig. 4. Energy level changes during one period of $\tau_s$

Eq. (1) can be reformulated with respect to $C_s$, as:

$$C_s \ge \frac{P_{act} - P_r}{P_{act} - P_{\sigma_k}} T_s, \quad (2)$$

Note that this condition is pessimistic, because it is assumed that the system is always in active state ($\sigma_0$), executing the task with the maximum power consumption characteristics, when $\tau_s$ is not running (i.e., during the discharge phase of length $T_s - C_s$). However, it provides a simple formula that can be checked in constant time, regardless of the initial energy level. In the rest of the paper, we refer to this version of $PCS$ which checks feasibility at design time by using Eq. (2), as $PCS^*$.

## C. Enhancing the Algorithm for Mixed Workloads

In some cases, thanks to a favorable scenario, the energy stored in the battery may reach the capacity, leading to a waste of energy. Although battery overflows do not represent a problem for neither real-time nor energy constraints, the scheduler may optimize the use of resources, such as energy and CPU time.

In fact, such an optimization may be quite useful for *mixed workloads* that contain both sporadic real-time and aperiodic non-real-time (NRT) tasks. For mixed workloads, the traditional objective is to meet the hard deadlines of the real-time tasks, while improving the *responsiveness* (i.e., average response time) of NRT tasks [13]. Hence, in our settings, some instances of $\tau_s$ may be skipped, making available its allocated computation time to NRT tasks. Figure 2, at the bottom schedule, illustrates the execution of an NRT task $\Omega$ during the idle intervals of the $PCS$ schedule. If the third job of $\tau_s$ is discarded, the response time of $\Omega$ can be shortened without causing any energy failure.

However, skipping too many instances of $\tau_s$ may hurt feasibility in the long term; in particular, as the harvesting rate $P_r$ changes at the end of each epoch $T_r$, typically of length 15-30 minutes, one should still try to maximize the battery energy level as much as possible by the end of the current epoch. Consequently, our proposed enhancement is based on *skipping* one instance of $\tau_s$ out of $j+1$ consecutive instances ($j \geq 1$), while ensuring maximization of the battery level by the end of epoch. Specifically, by denoting the initial energy level at the beginning of the epoch as $E_0$, the net energy harvested until the end of the current epoch has to be no less than the available capacity ($C - E_0$) in the battery:

$$N \cdot \Delta E^* \geq C - E_0 \geq 0, \tag{3}$$

Above, $N$ is the number of skipped instances during the current epoch $\left(N = \left\lfloor \frac{T_r}{(j+1)\cdot T_s} \right\rfloor\right)$ and $\Delta E^*$ is the difference between harvested and consumed energy in a time interval of length $(j+1)\cdot T_s$:

$$\Delta E^* = j \cdot (P_r - P_{\sigma_k}) \cdot C_s - (P_{act} - P_r) \cdot (j \cdot T_s - j \cdot C_s + T_s) \geq 0. \tag{4}$$

Since $T_r >> T_s$, we can approximate $N$ as $\frac{T_r}{(j+1)\cdot T_s}$. From Eqs (3) and (4), we can derive a lower bound for $j$:

$$j \geq \frac{(P_{act} - P_r) \cdot T_s + \frac{C - E_0}{T_r} T_s}{(P_{act} - P_{\sigma_k}) \cdot C_s - (P_{act} - P_r) \cdot T_s - \frac{C - E_0}{T_r} T_s}, \tag{5}$$

The value of $j$ must be set as the smallest integer that satisfies Eq. (5) – a higher value of $j$ may decrease the responsiveness of the NRT workload, while wasting energy that cannot be stored in the battery. This enhanced version of $PCS$, denoted by $PCS^{NRT}$, is invoked whenever an epoch starts and has constant-time complexity ($O(1)$), as only Eq (5) needs to be solved.

## IV. Experimental results

In this section, we provide the results of simulation experiments that we carried out in order to evaluate the performance of our proposed algorithms under different system parameters.

We considered an embedded system equipped with an NXP LPC1768 [14] processor (ARM Cortex M3 [15]), powered by a battery with capacity $C = 500mAh$ and two solar panels, each providing a maximum of $500mW$. The power consumption of the processor in active state, without considering the peripherals, is $P_{CPU} \approx 690mW$. When all peripherals are activated, the overall power consumption is around $1W$, giving $P_{dev} = 310mW$. Two low-power states are considered: *idle* ($\sigma_1$) and *sleep* ($\sigma_2$). Their power consumption and break-even times are $P_{\sigma_1} = 490mW$, $\xi_{\sigma_1} \approx 0ms$, $P_{\sigma_2} = 290mW$ and $\xi_{\sigma_2} = 15ms$. Observe that, although the *sleep* state consumes least power, its break-even time is not negligible.

The synthetic task sets used in the tests are composed of 10 tasks randomly generated using the UUniFast algorithm [16], where each period $T_i$ is uniformly distributed in the range of $[40, 500]ms$. In our simulations, we generated 4000 task sets (200 for each utilization value under consideration). The power consumption $P_i$ of each job of task $\tau_i$ is computed as:

$$P_i = P_{CPU} + x_i \cdot P_{dev},$$

where $0 < x_i \leq 1.0$ is a real number generated randomly. By choosing different $x_i$ values, we are able to model the case of tasks consuming different amount of power per time unit of execution.

We report the results of our experiments in three parts. The first set evaluates the effectiveness of the proposed algorithms in terms of the ratio of the task sets that are scheduled in feasible manner (called the *feasibility ratio*), with respect to both timing and energy constraints. The second set of experiments assess several online metrics, such as average sleep interval length and preemption count, and the last set analyzes the spare CPU bandwidth that is made available to potential non-real-time tasks.

### A. Feasibility ratio

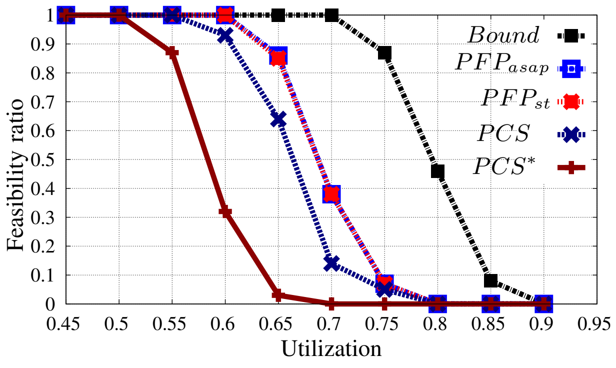
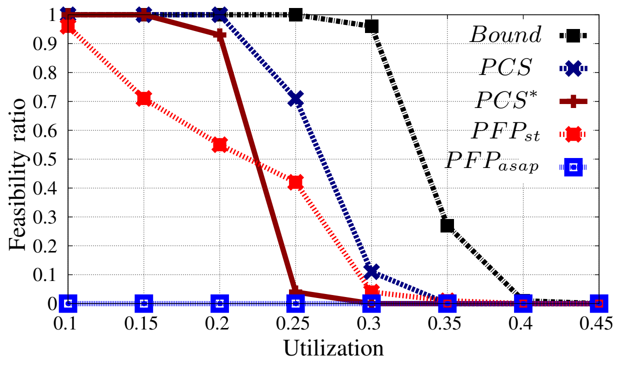


Fig. 5. Feasibility ratio vs. Utilization ($\xi_{\sigma_1} = 0$, $P_{\sigma_1} = 0W$, $P_r = 70\%$).

To evaluate the feasibility ratio under different system configurations, we implemented the following algorithms in our discrete-event simulator:

- $PFP_{asap}$: the computation-greedy algorithm whose optimality for fixed-priority systems with renewable energy, but only under negligible state transition overheads assumption, was formally proven in [8];
- $PFP_{st}$ (also called $EDeg$, from [7]): the energy-greedy algorithm whose feasibility performance was shown to lag slightly behind $PFP_{asap}$ in [8];
- $PCS^*$ that evaluates only the sufficient condition given by Eq. (2) at design time, to guarantee the feasibility;
- $PCS$ – presented in Section III-A;
- *Bound* that represents a theoretical limit on the feasibility performance of any scheduling algorithm.

The executions of $PFP_{asap}$, $PFP_{st}$ and $PCS$ are simulated, assuming an initial energy level of $E_0 = 0$ (the worst-case scenario) and checking for any deadline misses or battery failures during the hyperperiod $H$. To implement *Bound*, we adopted a methodology similar to the one suggested by Pagani


(a) $P_r = 50\%$.

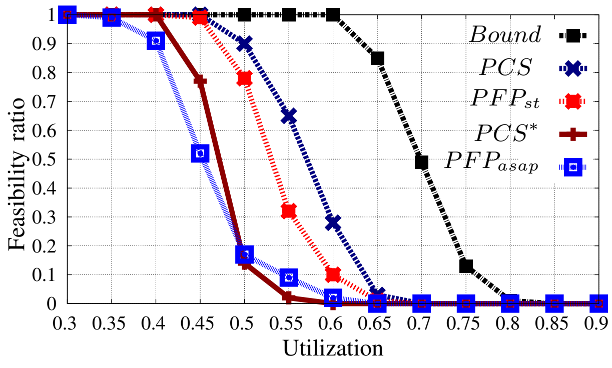

(b) $P_r = 70\%$.

Fig. 6. Feasibility ratio vs. Utilization for non-negligible transition overhead and sleep power.

and Chen [17], by transforming the problem into another one where tasks have identical release times and identical deadlines (equal to the hyperperiod $H$ of the original task set) while keeping task utilizations the same. Specifically, the battery capacity limit is ignored, and the longest possible charging interval within the hyperperiod is considered. In these ideal settings, the entire workload with utilization $U$ can be procrastinated for $(1-U)\cdot H$ time units and the system is feasible if and only if the energy harvested in $(1-U)\cdot H$ is higher than or equal to the energy consumed in $U \cdot H$ time units of execution.

Since $PFP_{asap}$ and $PFP_{st}$ were developed assuming negligible state transition overheads, they have been updated to incorporate a simple mechanism to deal with those overheads at run-time. Specifically, $PFP_{asap}$, which procrastinates only to harvest the energy necessary to execute the next computational unit, chooses the deepest sleep state whose break-even time is not longer than the required time to harvest the missing energy amount. Similarly, $PFP_{st}$, which exploits the whole available slack to charge the battery when it becomes empty, selects the deepest low-power state whose break-even time is shorter than or equal to the target procrastination delay. In addition, both algorithms are enhanced by putting the processor to the deepest low-power state which lets the system be fully operational by the next job arrival, when the CPU is idle.

In Figure 5, we first report the results for a system with "ideal" settings, that is, the one with a negligible power dissipation and zero break-even time associated with the sleep state (i.e., $\xi_{\sigma_2} = 0ms$ and $P_{\sigma_2} = 0W$). The harvesting rate $P_r$ is set to 70% of the maximum system power consumption $(P_{CPU} + P_{dev})$.

As expected, in this scenario with no overheads, $PFP_{asap}$'s optimality is demonstrated: it yields a feasibility ratio higher than $PFP_{st}$ and $PCS$. Also, in accordance with what is experimentally shown in [8], $PFP_{st}$ is a close second – in fact, its performance almost coincides with that of $PFP_{asap}$. $PCS$ comes next, showing that putting periodically the processor in sleep state is not the best approach on systems with zero transition overhead and zero sleep power.

However, when a realistic set of low-power states is considered, the picture changes entirely. The results are reported in Figure 6(a) and Figure 6(b), for $P_r = 50\%$ and 70% of the maximum power consumption $(P_{CPU} + P_{dev})$, respectively.

Our approach outperforms $PFP_{st}$ and $PFP_{asap}$ as it periodically guarantees replenishment phases which last longer than the sleep state's break-even time, overcoming the limitations due to short idle intervals. For instance, when $P_r = 70\%$, $PFP_{asap}$'s performance degrades when $U = 0.35$, while $PCS$ successfully schedules all the task sets up to $U = 0.45$. The performance of $PFP_{asap}$ drops because it is able to exploit only shallow low-power states. The difference between $PCS^*$ and $PCS$ is entirely due to the pessimistic nature of the offline test. Finally, although $PFP_{st}$'s performance is close to $PCS$, its online complexity is pseudo-polynomial whereas our simple algorithm has a linear complexity at runtime.

Figure 7 presents the impact of the relative harvesting rate $(P_r/(P_{CPU} + P_{dev}))$ on the feasibility ratio when $U = 0.4$. In other words, this analysis shows the minimum required harvesting rate that guarantees schedulability. Again, $PCS$ offers the best performance (besides *Bound*, which gives the theoretical limit): it guarantees the feasibility for the lowest harvesting power.

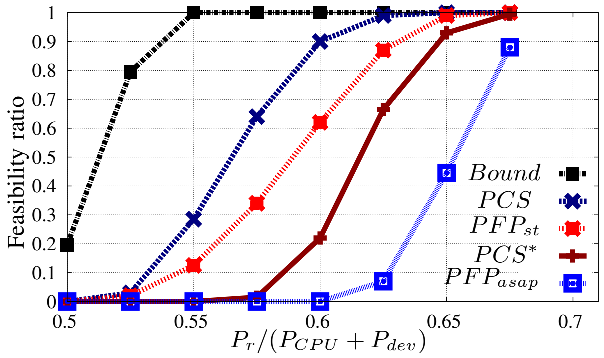
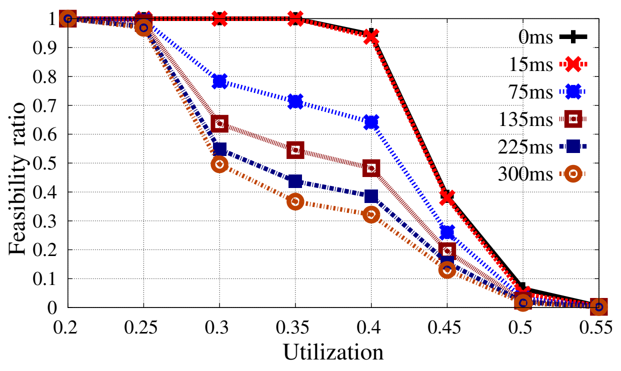
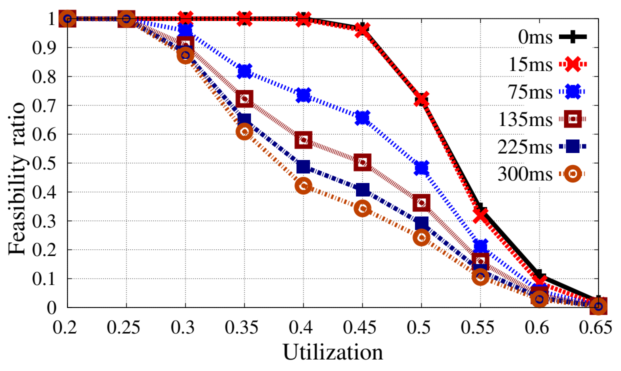


Fig. 7. Feasibility ratio vs. Harvesting rate ($U = 0.4$).

Next, we analyze how the performance changes as a function of the different break-even times associated with the *sleep* state, in Figure 8(a) and Figure 8(b), for $PCS^*$ and $PCS$, respectively. The results are obtained for $P_r = 65\%$, and the minimum period value of $40ms$. As expected, the results show that the longer the break-even time, the lower the



(a) Under $PCS^*$.



(b) Under $PCS$.

Fig. 8. Feasibility ratio vs. Utilization under different break-even times for $P_r = 65\%$.

feasibility ratio. However, $PCS$ is less affected than $PCS^*$ by the changes in the break-even time: this is because, it opportunistically manages to compact inactive intervals at runtime.

## B. Online metrics

This subsection considers some online metrics that are relevant to characterize the algorithms' performance at runtime:

- *average time spent in sleep state:* this metric shows the capability to exploit the deepest low-power state for as long as possible. In generally, the longer the sleep intervals, the higher the harvested energy and the higher the probability to execute the workload without battery failures;
- *number of preemptions*: this metric shows the total number of preemptions experienced by each algorithm. Obviously, an algorithm with prohibitive number of preemptions is not desirable, considering the overhead associated with each preemption;
- *average battery energy level:* this metric highlights the success of the algorithms in effectively harvesting energy, while executing the workload.

The comparisons in this section involve only $PCS$ and $PFP_{st}$, due to the fact that the results in [8], that are in line with ours, clearly show that $PFP_{st}$ outperforms $PFP_{asap}$ by a significant margin when considering these online metrics.

First, we analyze the average length of the sleep intervals for $PFP_{st}$ and $PCS$ in Figure 9. We observe that $PCS$ guarantees longer sleep intervals on the average; however their lengths tend to decrease with increasing utilization (the time assigned to $\tau_s$ becomes shorter). Conversely, the average sleep interval length for $PFP_{st}$ first remains constant with the utilization (due to the feasibility constraint and reaching the battery capacity during recharging), before dropping. It is worth noting that for $PFP_{st}$, the total length of the idle intervals is comparable to that of the sleep intervals. Note that some values are not reported for certain data points whenever the corresponding algorithm does not generate a feasible solution for that utilization value.

When the absolute time spent in low-power states (idle + sleep) is considered instead of the average duration,

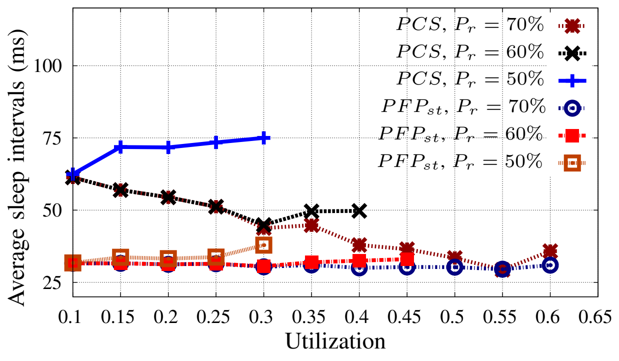


Fig. 9. Average sleep interval vs. Utilization under $PCS$ and $PFP_{st}$.

$PCS$ and $PFP_{st}$ have similar trends. Specifically, the difference is around only 5% of the entire hyperperiod during which $PFP_{st}$ puts the system in the idle state rather than the sleep state, due to the break-even time considerations.

Next, in Figure 10, we show the number of preemptions of $PCS$, normalized with respect to $PFP_{st}$'s. $PCS$ has the side effect of introducing a higher preemption count than $PFP_{st}$ as the system executes the additional charging task $\tau_s$ with the period set to the minimum period value. However, except for very low utilization values, the increase is only around 50% compared to $PFP_{st}$ during the hyperperiod. The preemption ratio is not reported for utilization values where $PFP_{st}$ or $PCS$ does not generate feasible values.

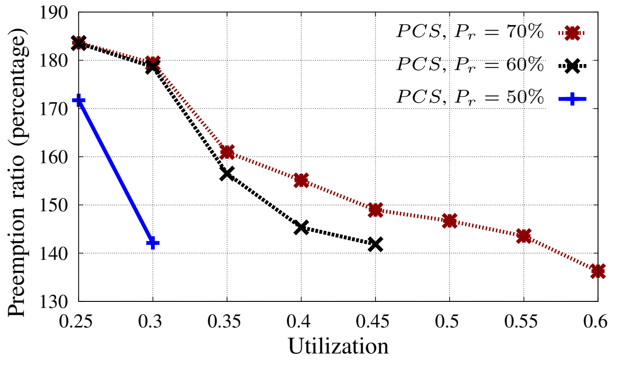


Fig. 10. Ratio of preemption number in $PCS$ to that in $PFP_{st}$ (percentage).

Finally, in Figure 11, we analyze the average battery level, normalized to the battery capacity, within a hyperperiod. We observe that both $PCS$ and $PFP_{st}$ guarantee a similar average battery level, even though, charging principles and timing are different. In general, the higher the utilization, the lower the average battery level as the available time to charge the battery is shorter. In addition, the higher the harvesting rate $P_r$, the higher the average battery level as the charging process is more effective. Despite similar performance trends, we note that the online component of $PFP_{st}$ has pseudo-polynomial time complexity (due to the computation of the maximum slack at runtime), while $PCS$ has linear complexity.

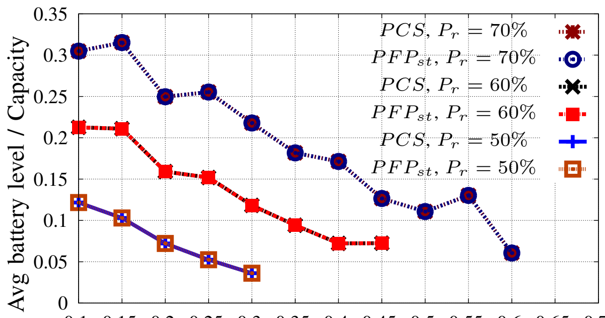


Fig. 11. Average battery level vs. Utilization under $PCS$ and $PFP_{st}$.

### C. *Effective spare CPU bandwidth*

Finally, we undertake an experimental analysis of $PCS^{NRT}$ (Section III-C) which targets allocating as much CPU bandwidth as possible to non-real-time (NRT) tasks in order to improve their responsiveness. Recall that under $PCS^{NRT}$, effectively additional CPU bandwidth is reserved by skipping an instance of $\tau_s$ out of $j+1$ consecutive instances, where $j$ is computed statically at the beginning of each epoch, to preserve feasibility in terms of deadlines and energy constraints.

In these experiments, the utilization of the real-time task set is set to $0.5$, potentially leaving $50\%$ of the CPU time to non-real-time tasks, regardless of $\tau_s$'s configuration. Figure 12 shows the effective spare CPU bandwidth that is left for the NRT workload, by considering also $\tau_s$'s invocations, as a function of the harvesting rates and initial battery level $E_0$. In this first simulation, the battery capacity is $C = 500mAh$ and the average $\tau_s$ utilization is $0.4$. When the online enhancement is not able to skip any $\tau_s$ instance (mainly due to low initial battery level $E_0$, or scarce harvested power $P_r$), the CPU bandwidth allocated to NRT tasks is around $10\%$, which is equivalent to the static slack $(0.5 - 0.4)$. As soon as more favorable execution scenarios occur (either thanks to the higher initial battery level $E_0$ or higher harvesting rate $P_r$), several of $\tau_s$'s jobs can be skipped without hurting the feasibility. Specifically, in the best case, the effective spare CPU bandwidth goes from $10\%$ to $30\%$, meaning that the actual utilization of $\tau_s$ is $0.2$ (an instance is skipped out of 2).

In Figure 13, the same analysis is repeated, this time for a system with lower battery capacity, $C = 250mAh$. Since

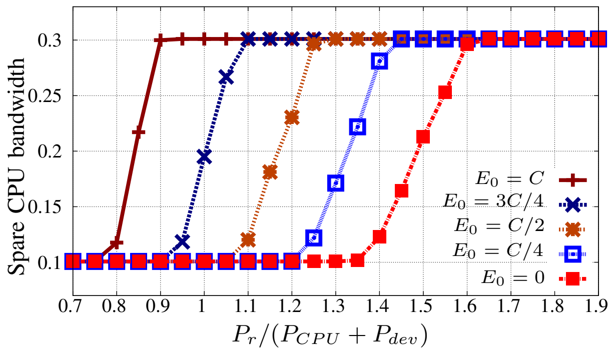


Fig. 12. Effective spare CPU bandwidth vs. $P_r/(P_{CPU} + P_{dev})$ for $C = 500mAh$.

the maximum amount of energy that can be stored is lower, the upper bound on the effective spare CPU bandwidth is now reached sooner.

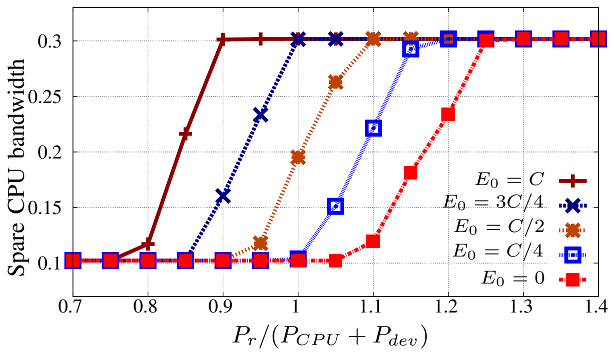


Fig. 13. Effective spare CPU bandwidth vs. $P_r/(P_{CPU} + P_{dev})$ for $C = 250mAh$.

## V. RELATED WORK

Rakhmatov and Vrudhula [18] addressed the problem of minimizing the energy consumption, while guaranteeing a common deadline and avoiding battery failure for real-time tasks. Their first algorithm decides how to schedule the tasks by considering the precedence constraints and battery characteristics. Then, when the system restarts after a battery failure, the algorithm is invoked to exploit idle states or modulate the processing speed.

Chetto et al. [7] considered the real-time scheduling problem under the Rate Monotonic policy for systems with renewable energy. They proposed five reactive heuristics, which are executed whenever the battery becomes empty. The first heuristic keeps the processor in sleep state for a predefined fixed interval $x$, while the second extends the inactive period until the charge level reaches a certain threshold. The third one (called $EDeg$, later labeled as $PFP_{st}$ in [8]) uses the entire available slack time. The fourth heuristic stops charging when the battery is completely charged. Another fifth algorithm is invoked when the battery energy level drops below a predefined lower bound.

El Ghor et al. [19] proposed an algorithm under the EDF policy that applies task procrastination to the energy harvesting problem. Specifically, the pending jobs are executed if there is enough energy to complete them, otherwise the available slack time is exploited to charge the battery.

Abdeddaïm et al. [8] proposed an optimal algorithm, called $PFP_{asap}$. The algorithm puts the processor in sleep state for the shortest time interval that still guarantees to harvest the required energy for executing only the next computational unit of the highest priority task (which is equivalent to the first heuristic in [7] by considering $x = 1$). Despite the algorithm's optimality, the paper shows that $PFP_{asap}$ introduces a significant number of state transitions and preemptions. Moreover, the overhead associated with transitions to/from low-power states is not considered.

Kooti et al. [20] proposed an algorithm which divides the analysis operation interval (one day) into a set of frames (30 minutes each) without any correlation with the task periods. They assume no more than $n$ out of $m$ consecutive jobs of a task miss their deadlines. At design time, an ILP solver computes for each frame the allocated energy and the number of hard real-time and best-effort jobs. Then, within each frame, the online step executes the hard real-time tasks while the best-effort tasks are executed only if there is enough energy.

Moser et al. [21] defined a 3-layered solution for energy harvesting systems: Application Rate Control, Service Level Allocation, and Real-Time Scheduling. For the first two layers, a long-term analysis is conducted to guarantee an average high performance level. For the real-time scheduling phase, an algorithm called *Lazy Scheduling Algorithm (LSA)* is proposed. Specifically, by using the EDF policy, the algorithm sets the start time of a job equal to its deadline minus the total harvested energy divided by the power consumption. In this way, idle intervals that can be used to recharge the battery are introduced in the schedule. However, the transition overheads associated with low-power states are not considered. The idea is further followed in [22], which provides a admission condition that considers the energy demand and supply functions.

## VI. Conclusions

In this paper, we addressed the energy management problem for fixed-priority real-time systems with renewable energy. Our proposed approach is based on planning in advance for periodically alternating charging and task execution phases offering simplicity and predictability. Moreover, our proactive algorithm is designed to take advantage of multiple low-power (sleep) states offered by many current processors.

At design time, the duration and frequency of charging phases are computed, whereas, at runtime, the algorithm extends charging phases opportunistically by taking advantage of the inactive intervals to exploit deeper low-power states. Moreover, an enhancement is provided to improve the responsiveness of non-real-time workloads without affecting the overall feasibility of real-time tasks. Our proposal was experimentally compared with the state-of-the-art algorithms. We showed that our algorithm yields a higher feasibility ratio when a realistic power model with non-zero power state transitions is considered. Moreover, we evaluated several runtime performance metrics, showing the viability of our proposal.

## References

[1] Y.-H. Lee, K. Reddy, and C. Krishna, "Scheduling techniques for reducing leakage power in hard real-time systems," in *Proc. of the 15th IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[2] R. Jejurikar, C. Pereira, and R. K. Gupta, "Leakage aware dynamic voltage scaling for real time embedded systems," in *Proc. of the IEEE Design Automation Conference (DAC)*, 2004.

[3] M. Bambagini, M. Bertogna, M. Marinoni, and G. Buttazzo, "An energy-aware algorithm exploiting limited preemptive scheduling under fixed priorities," in *Proc. of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.

[4] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 584–600, May 2004.

[5] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, pp. 31:1–31:23, Jul. 2009.

[6] J. Hsu, S. Zahedi, A. Kansal, M. Srivastava, and V. Raghunathan, "Adaptive duty cycling for energy harvesting systems," in *Proc. of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 2006.

[7] M. Chetto, D. Masson, and S. Midonnet, "Fixed priority scheduling strategies for ambient energy-harvesting embedded systems," in *Proc. of the IEEE/ACM International Conference on Green Computing and Communications (GreenCom)*, 2011.

[8] Y. Abdeddaim, Y. Chandarli, and D. Masson, "The optimality of $PFP_{asap}$ algorithm for fixed-priority energy-harvesting real-time systems," in *Proc. of the 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[9] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 8, no. 3, pp. 299–316, 2000.

[10] R.-S. Liu, P. Sinha, and C. Koksal, "Joint energy management and resource allocation in rechargeable sensor networks," in *Proc. of the IEEE INFOCOM*, 2010.

[11] B. Zhang, R. Simon, and H. Aydin, "Maximum utility rate allocation for energy harvesting wireless sensor networks," in *Proc. of the 14th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM)*, 2011.

[12] E. Bini, M. Di Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," *Real-Time Syst.*, vol. 39, no. 1-3, pp. 5–30, Aug. 2008.

[13] G. C. Buttazzo, *Hard Real-time Computing Systems*. Santa Clara, CA, USA: Springer-Verlag, 2004.

[14] "Nxp web site," http://www.nxp.com/.

[15] "Arm web site," http://www.arm.com/.

[16] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, May 2005.

[17] S. Pagani and J.-J. Chen, "Energy efficiency analysis for the single frequency approximation (sfa) scheme," in *Proc. of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013.

[18] D. Rakhmatov and S. Vrudhula, "Energy management for battery-powered embedded systems," *ACM Trans. on Embedded Computing Systems*, 2003.

[19] H. El Ghor, M. Chetto, and R. H. Chehade, "A real-time scheduling framework for embedded systems with environmental energy harvesting," *Journal of Computers and Electrical Engineering*, 2011.

[20] H. Kooti, N. Dang, D. Mishra, and E. Bozorgzadeh, "Energy budget management for energy harvesting embedded systems," in *Proc. of the 18th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[21] C. Moser, J.-J. Chen, and L. Thiele, "Dynamic power management in environmentally powered systems," in *Proc. of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[22] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Real-time scheduling with regenerative energy," in *Proc. of the IEEE European Conference Conference on Real-Time Systems (ECRTS)*, 2006.