The Optimality of PFPasap Algorithm for Fixed-Priority Energy-Harvesting Real-Time Systems

Yasmina Abdeddaïm, Younès Chandarli, Damien Masson

The Optimality of PFP_{ASAP} Algorithm for Fixed-Priority Energy-Harvesting Real-Time Systems

Yasmina Abdeddaïm, Younès Chandarli and Damien Masson
Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge UMR 8049,
UPEMLV, ESIEE Paris, ENPC, CNRS, F-93162 Noisy-le-Grand France
y.abdeddaim@esiee.fr / younes.chandarli@univ-paris-est.fr / d.masson@esiee.fr

April 29, 2013

Abstract

The paper addresses the real-time fixed-priority scheduling problem for battery-powered embedded systems whose energy storage unit is replenished by an environmental energy source. In this context, a task may meet its deadline only if its cost of energy can be satisfied early enough. Hence, a scheduling policy for such a system should account for properties of the source of energy, capacity of the energy storage unit and tasks cost of energy. Classical fixed-priority schedulers are no more suitable for this model. Based on these motivations, we propose PFP_{ASAP} an optimal scheduling algorithm that handles both energy and timing constraints. Furthermore, we state the worst case scenario for non concrete tasksets[1] scheduled with this algorithm and build a necessary and sufficient feasibility condition for non concrete tasksets. Moreover, a minimal bound of the storage unit capacity that keeps a taskset schedulable with PFP_{ASAP} is also proposed. Finally, we validate the proposed theory with large scale simulations and compare our algorithm with other existing ones.

1 Introduction

Due to the growing demand for smaller devices with longer battery life, energy management has become one of the major goals in embedded systems research. Indeed, a naive use of the energy available on board can lead to a short runtime for these devices. However, the targeted embedded applications can be required to operate over long periods after they are deployed, for example, in the case of sensor nodes. The extended life of these electronic devices is of particular importance when they have limited accessibility. Thus, collecting energy from the ambient environment can be a very interesting solution, which is known as Energy Harvesting. In this process, energy is drawn from the environment and then converted and stored for use in electronic applications. Compared to classical energy storage devices, the environment proves to be an infinite source of available energy. Furthermore, using this kind of energy eliminates the need to replace batteries periodically that constitute a major part of service and maintenance.

Many environmental sources can be exploited, including thermal, optical, mechanical, fluid, etc. Energy sources must be considered according to the characteristics of the application. Self powered sensors for medical implants and remote condition monitoring embedded sensors in structures such as bridges or buildings are typical examples of targeted applications.

In addition, the applications running on power-limited systems can be subject to timing constraints. Consequently, real-time and energy-aware features are both highly desirable and sometimes crucial for such systems. An energy harvesting system is composed of three parts:

The harvester is the part that converts the energy from ambient surroundings into usable electrical power.

[1] a non concrete taskset is a set of real-time tasks whose offsets are known only at run-time

The storage unit is a device used to store the electrical energy produced by the harvester (e.g. a rechargeable battery or a capacitor).

The computing system is a real-time system that uses the energy stored in the battery to run the softwares.

In this paper we are interested in the problem of real-time scheduling for Energy-Harvesting systems. The challenge is to schedule real-time tasks and to make the best use of the available energy which is highly dependent on the environment. The energy consumption of the system should be adjusted to maximize its performance instead of minimizing its overall energy consumption as in classical battery-powered systems. A new role of the operating system is to properly manage the activity of the processing unit so that, at any time, there is sufficient energy in the storage unit to satisfy all the constraints.

This work focuses on optimal fixed-priority solution of this problem. The remaining part of the paper is organized as follow: first, we present the related work in Section 2. The model is described in Section 3. In Section 4 we introduce PFP_{ASAP}, a fixed-priority scheduling algorithm, and then we study some of its properties, namely the worst case scenario and its optimality for non concrete traffic. A feasibility condition based on PFP_{ASAP} is also proposed. In Section 5, we evaluate the performance of PFP_{ASAP} and we compare it with other algorithms by performing large scale simulations. Finally, we discuss the themes of future work, then, we conclude in Section 6.

2 Related Work

Researchers started to address the issues of power and scheduling only in the past decade with the objective of either minimizing power usage under timing constraints or maximizing the system performance under energy constraints. Nevertheless, most of them have not considered the limited capacity of the battery and the need to manage its continuous replenishment.

Until recently, the most of this research has focused on saving energy using the DVFS technique (Dynamic Voltage and Frequency Scaling) [16, 15]. The idea here is to save energy by slowing down the processor just enough to meet tasks deadlines.

These techniques have limitations in energy harvesting systems because they increase the probability of transient faults [16, 15] and cannot be used alone in case where there is not enough energy to execute. An energy aware scheduling strategy for harvesting systems must dynamically manage tasks according to the profiles of both available energy and the workload of the processor.

The first work which addressed the scheduling problem of energy harvesting systems was presented by Mossé in [2]. The problem was solved under a very restrictive task model: the frame-based model where all the tasks have exactly the same period and the same implicit deadline. Later in [14], Moser et al. proposed an optimal algorithm called LSA (Lazy Scheduling algorithm) for periodic or aperiodic tasks. However, in their hypotheses, the CPU frequency can be changed to adjust the Worst Case Execution Time (WCET) of the tasks depending on their energy consumption. Thus, the results of this work rely on the assumption that task energy consumption is directly linked to their WCET. Recent work shows that this hypothesis is not suitable for embedded systems [11].

Later, a clairvoyant algorithm called $EDeg$ and several heuristics have been proposed in [10, 7]. In this context, an algorithm is said to be clairvoyant if it takes scheduling decisions according to the processor and the energy load, the amount of incoming energy and the energy level in the storage unit. The algorithm $EDeg$ relies on a generalizable meta policy: as long as the system can perform without energy failure, a standard policy such as EDF is used. Then, as soon as *future* energy failure is detected, the system is suspended as long as possible depending on timing constraints or until the energy storage unit is full. To detect such future energy failure, the notion of slack time [12] was extended to the notion of slack energy. This algorithm was evaluated with non clairvoyant heuristics that schedule jobs as soon as possible until energy runs out, then suspend the system a while without looking for the future state of energy. For example they suspend the system for a fixed period of time or consume all available slack-time to replenish the battery.

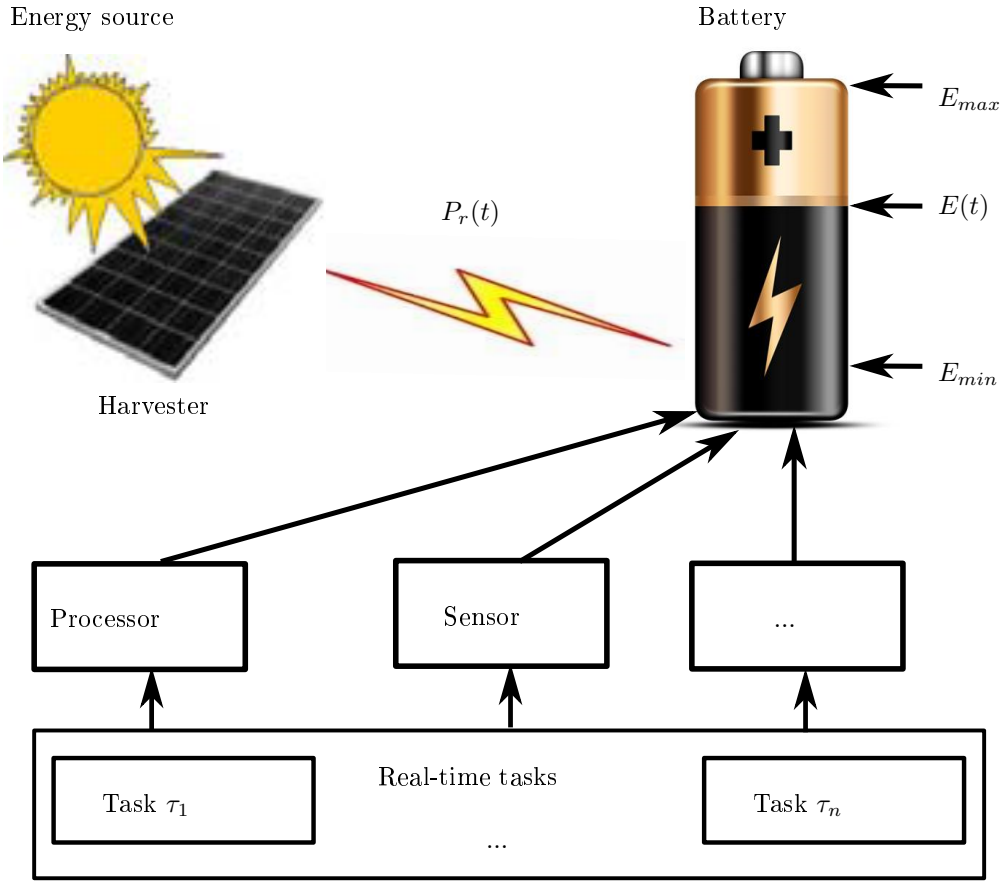


Figure 1: Energy Harvesting Embedded System Model

Most work about scheduling energy-harvesting systems focus on dynamic priority scheduling by proposing algorithms mainly based on EDF because of its optimality for classical scheduling problems. However, most of embedded systems usually operate with fixed-priority scheduling policies because of their simplicity and their low overhead.

3 Problem Statement

3.1 Taskset Model

We consider a non concrete real-time taskset in a renewable energy environment defined by a set of n periodic and independent tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized by its priority P_i, its worst case execution time C_i, its period T_i, its deadline D_i and its worst case energy consumption E_i. The execution time C_i and the energy consumption E_i of a task are fully independent, for example considering two tasks τ_i and τ_j, we can have $C_i < C_j$ and $E_i > E_j$. A task τ_i releases an infinite number of jobs separated by T_i time units and each job must execute during C_i time units and consume E_i energy units. All the tasks consume energy linearly, i.e. a constant amount of energy for each execution time unit. Deadlines are constrained or implicit. The taskset is priority-ordered, task τ_n being the task with the lowest priority. Since the considered taskset is a non concrete one, the offsets denoted as O_i are known only at runtime.

3.2 Target Application Description

We consider an embedded system connected to an energy harvesting device. An energy harvesting device is a system collecting energy from its environment (e.g. with a solar panel). The collected energy is stored in an energy storage unit with fixed capacity (e.g. chemical battery or capacitor). We suppose that the quantity of energy that arrives in the storage unit is a function of time which is either known or bounded. As mentioned in Section 3.1, task energy and processor cost are fully independent. Indeed, in practice, a task can use some devices that are independent from the processor (e.g. sensors, engines). Even if we consider only the processor consumption, the later relies heavily on the kind of circuitry that is used by the code, rather than on the duration of its execution [11].

The replenishment of the storage unit is performed continuously even during the execution of tasks, and the energy level of the battery fluctuates between two thresholds E_{min} and E_{max} where E_{max} is the maximum capacity of the storage unit and E_{min} is the minimum energy level that keeps the system running. The difference between these two thresholds is the part of the battery capacity dedicated to tasks execution, denoted as $\mathcal{C}$. We suppose that $\mathcal{C}$ is sufficient to execute at least one time unit of each task. This means that $\mathcal{C}$ must be greater or equal to the maximum instantaneous consumption, i.e. $\mathcal{C} \geq \max_{\forall i}(E_i/C_i)$, otherwise the taskset cannot be executed. For the sake of clarity, we can consider without loss of generality that $E_{min} = 0$ and that $\mathcal{C} = E_{max}$. The battery level at time t is denoted as $E(t)$. As the tasks offsets, the initial level of the battery $E(0)$ is unknown before runtime. We note $P_r(t)$ the replenishment function of the battery, then, the energy replenished during any time interval $[t_1, t_2]$ denoted as $g(t_1, t_2)$ is given by Formula 1.

$$g(t_1, t_2) = \int_{t_1}^{t_2} P_r(t)dt \tag{1}$$

To simplify the problem, we assume $P_r(t)$ to be a constant function, i.e. $P_r(t) = P_r$. Then, the energy replenished during any time interval $[t_1, t_2]$ is given by Formula 2.

$$g(t_1, t_2) = (t_2 - t_1) \times P_r \tag{2}$$

Below, we use P_r instead of $P_r(t)$ to denote the replenishment function and we suppose that $P_r \leq \mathcal{C}$ to avoid energy loss. The replenishment process in energy harvesting systems is usually slower than the dissipation, for this reason we suppose that tasks consume more energy than the one which is replenished during executions, i.e. $\forall i$, $E_i \geq C_i \times P_r$.

We define the processor utilization of τ_i as $u_i^p = C_i/T_i$ and its energy utilization as $u_i^e = E_i/(T_i \times P_r)$. The total utilization of the system is the sum of all the tasks utilization, i.e. $U^p = \sum_{i=1}^{n} u_i^p$ and $U^e = \sum_{i=1}^{n} u_i^e$.

In the considered model, the system has to respect all deadlines and energy constraints, namely tasks energy cost and battery capacity. The system executes and consumes energy when it is available and only replenishes it when it is not. The battery energy level never exceeds its threshold E_{max} nor fall below E_{min}. Thus, a taskset is feasible if and only if there is a schedule where all the deadlines are met and the battery level never fall below E_{min}.

Figure 1 recapitulates these descriptions.

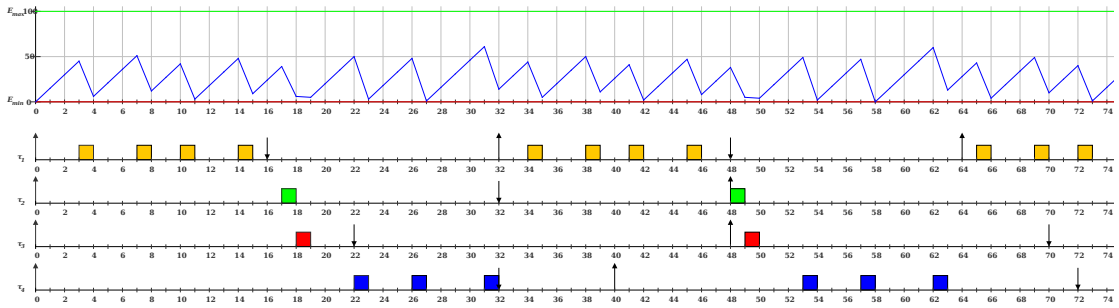
4 Theoretical Study of PFP_{ASAP}

4.1 As Soon As Possible Preemptive Fixed-Priority Algorithm

In [7], a scheduling algorithm for energy harvesting systems was introduced. This algorithm is a fixed-priority one which takes into account the tasks energy cost and the battery capacity during scheduling operations. Tasks are executed according to their priority, furthermore, whenever there is not enough energy to execute, jobs execution can be suspended to replenish energy for a fixed amount of time x. The authors performed tests by varying the x parameter from $x = 4$ to $x = 100$

-	C_i	E_i	T_i	D_i	P_i
τ_1	4	216	32	16	1
τ_2	1	48	48	32	2
τ_3	1	16	48	22	3
τ_4	3	186	40	32	4

(a) Taskset Γ_1



(b) PFP_{ASAP} time chart for taskset Γ_1

Figure 2: A PFP_{ASAP} schedule

Algorithm 1 PFP_{ASAP} Algorithm

```
1: t ← 0
2: loop
3:    A ← set of active tasks at time t
4:    if A ≠ ∅ then
5:       τ_k ← the highest priority task of A
6:       if E(t) + P_r − E_min ≥ E_k/C_k then
7:          execute τ_k for one time unit
8:       end if
9:    end if
10:   t ← t + 1
11: end loop
```

and the best value for their tasksets sample was 6. However, they did not evaluate the algorithm for $x = 1$.

In this section we study a special case of this algorithm, one where $x = 1$ that we call *As Soon As Possible Preemptive Fixed-Priority Algorithm* (PFP_{ASAP}). Algorithm 1 shows how PFP_{ASAP} takes decisions at time t. It schedules jobs as soon as possible when there is enough energy to execute one time unit, otherwise, it suspends tasks executions to replenish the battery. The replenishment periods are as long as needed for the execution of one time unit.

Figure 2(b) illustrates an PFP_{ASAP} schedule of the taskset Γ_1 described in Table 2(a) in the time interval $[0, 72]$. In this example we have $E_{max} = 100$, $E_{min} = 0$ and $P_r = 15$. At time $t = 0$ the battery is empty, therefore, task τ_1 cannot be executed. The battery is replenished until time $t = 3$, i.e. until there is enough energy to execute one time unit of τ_1. Then, the algorithm follows the same scheduling rules for the rest of the schedule.

Below, we will first address the PFP_{ASAP} worst case scenario, then we will discuss its optimality and finally, we will build a necessary and sufficient feasibility condition for the scheduling problem.

4.2 Worst Case Scenario

The aim of this section is to characterize the worst case scenario that a taskset can encounter during its execution. First, let us recall the notion of processor demand, then we will extend it to include task energy consumption.

Definition 1. *The processor demand of the i^{th} priority level at time t denoted as $wp_i(t)$, is the amount of time necessary to execute jobs of priority levels $1, \ldots, i-1, i$ requested in the interval of time $[0, t]$. It can be obtained by formula 3.*

$$wp_i(t) = \sum_{j \leq i} \left\lceil \frac{t - O_j}{T_j} \right\rceil \times C_j \tag{3}$$

Now we introduce the notion of replenishment demand.

Definition 2. *The replenishment demand of the i^{th} priority level at instant t denoted $we_i(t)$, is the amount of energy to be replenished to execute jobs of priority levels $1, \ldots, i-1, i$ requested in the interval of time $[0, t]$. It can be calculated by formula 4.*

$$we_i(t) = \sum_{j \leq i} \left\lceil \frac{t - O_j}{T_j} \right\rceil \times E_j - E(0) \tag{4}$$

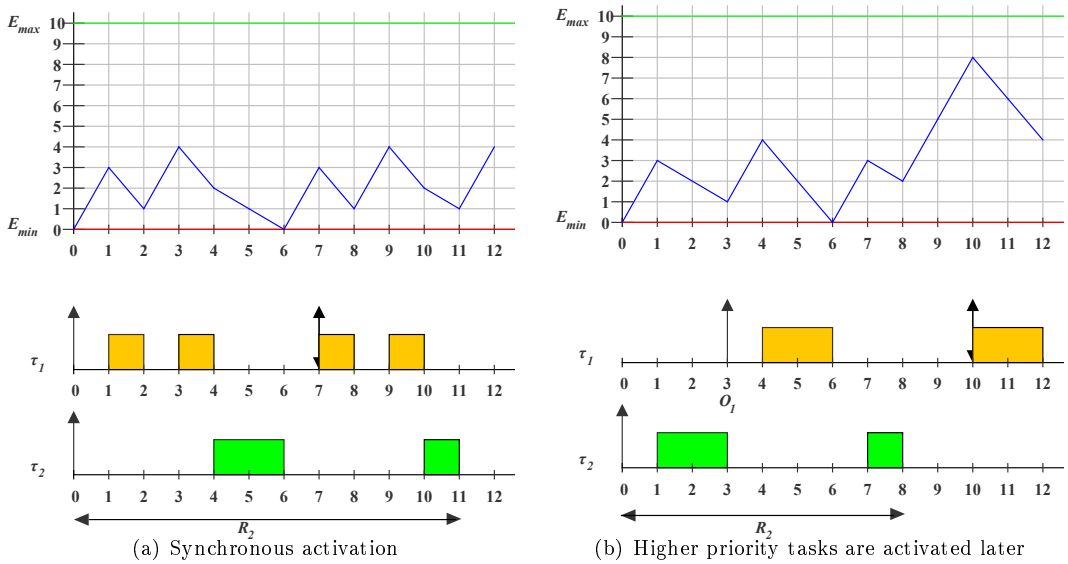
The intuition of formula 4 is derived from the notion of processor demand. It represents the sum of the cost of energy of all the jobs of priority equal or higher than i requested during the time interval $[0, t]$. Then, we remove the initial battery level $E(0)$ to fit the exact amount of energy to be replenished.

Definition 3. *The time demand of the i^{th} priority level at instant t denoted as $w_i(t)$, is the minimum amount of time necessary to satisfy both of the replenishment and processor demand in the time interval $[0, t]$. This can be calculated by formula 5.*

$$w_i(t) = max\left(\left\lceil \frac{we_i(t)}{P_r} \right\rceil, wp_i(t)\right) \tag{5}$$

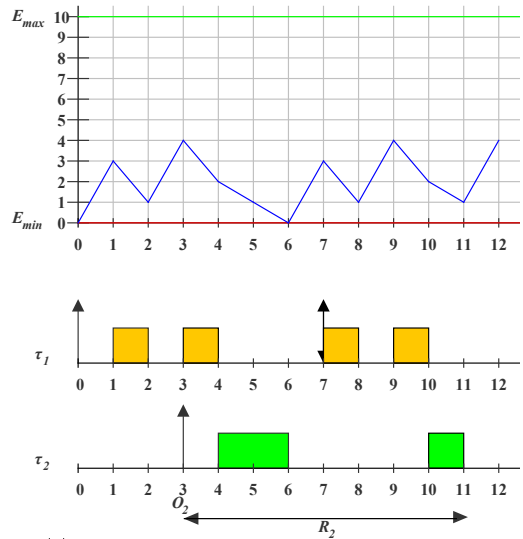
Definition 4. *The response time of the first job of τ_i according to PFP_{ASAP} denoted as R_i is the execution termination date of i^{th} priority level minus O_i. The termination date of the first job of τ_i denoted as t_f is the smallest solution of the system of equations 6.*

$$\begin{cases} w_i(t_f) &= max\left(\left\lceil \frac{we_i(t_f)}{P_r} \right\rceil, wp_i(t_f)\right) \\ w_i(t_f) &= t_f \\ w_i(t_f) &> O_i \end{cases} \tag{6}$$

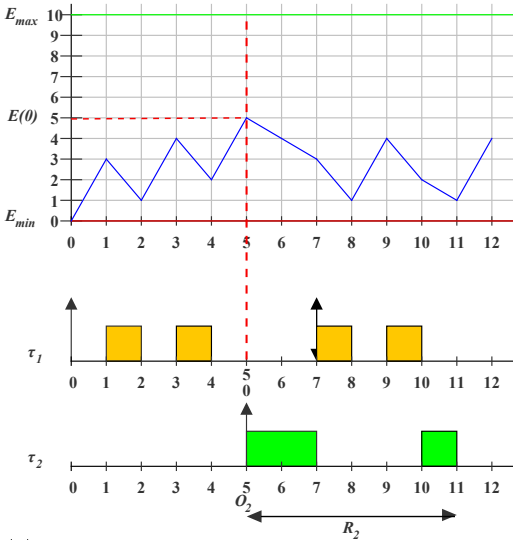


(a) Synchronous activation

(b) Higher priority tasks are activated later



(c) Higher priority tasks are activated earlier



(d) Higher priority tasks are activated earlier with modified offsets

Figure 3: Response time in different activation scenarios

Now, we can use these definitions to characterize the worst case scenario which is expected to be the synchronous activation of all the tasks when the battery is at its minimum level. This intuition is justified by the comparison of all possible activation scenarios as shown in Figure 3.

Figure 3(a) illustrates the case where all the tasks are requested simultaneously. If at least one higher priority task is requested later, the response time of lower priority tasks decreases as illustrated in Figure 3(b). Then, if higher priority tasks are requested earlier, the response time of lower priority tasks cannot be longer than the one in the synchronous scenario as shown in Figures 3(c) and 3(d).

Thus, we propose Theorem 1.

Theorem 1. *Let Γ denote a non concrete taskset composed of n priority-ordered tasks with constraint or implicit deadlines. The PFP_{ASAP} worst case scenario for any task of Γ occurs whenever this task is requested simultaneously with requests of all higher priority tasks and the battery is at the minimum level E_{min}.*

Proof. We will compare the jobs response times in the scenario of the theorem with all other possible ones. As mentioned earlier, the response time of a job is equal to its termination date minus its offset. The main key of the proof is to argue with termination dates and offsets by comparing their possible values in different cases of activation scenario.

Let $\{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of n priority-ordered tasks where τ_n is the task with the lowest priority. Let S_i^s denote the scenario where task τ_i and all higher priority tasks are requested simultaneously at the lower battery level E_{min}. The worst case scenario for a task τ_i is the one which maximizes its response time, i.e. the scenario which maximizes the termination date of the first job of the i^{th} priority level.

If S_i^s is not the worst scenario, there must be an other one leading to a greater response time for the i^{th} priority level.

Firstly, we consider the scenario where $E(0) > E_{min}$. In this case there is some amount of energy available at time $t = 0$. Therefore, the system needs less replenishment demand than the scenario where $E(0) = E_{min}$, and PFP_{ASAP} introduces shorter or equal replenishment periods and leads to shorter response time for all the tasks. This is in contradictory with our hypothesis, thus, such a scenario cannot lead to longer response times.

Secondly, we consider the scenario with different offsets. Let us denote S_i^a as the scenario where $E(0) = E_{min} = 0$ and all tasks have different offsets. Let t_s denote the termination date of the first job of task τ_i in the synchronous scenario S_i^s and let t_a denote the termination date of the same job in the asynchronous scenario S_i^a. Scenario S_i^a is worse than scenario S_i^s implies that $t_a > t_s$.

We know that

$$t_s = w_i^s(t_s) = max\left(\left\lceil \frac{we_i(t_s)}{P_r} \right\rceil, wp_i(t_s)\right) \tag{7}$$

and $\left\lceil \frac{we_i(t_s)}{P_r} \right\rceil \geq wp_i(t_s)$ because in our model $E(0) = 0$ and $\forall i$, $E_i \geq C_i \times P_r$. This reveals the fact that in the considered model, we must have replenishment periods which increase job response time. Then,

$$t_s = w_i^s(t_s) = \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_s}{T_j} \right\rceil \times E_j}{P_r} \right\rceil \tag{8}$$

Similarly,

$$t_a = w_i^a(t_a) = \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_a - O_j}{T_j} \right\rceil \times E_j}{P_r} \right\rceil \tag{9}$$

Knowing that $w_i^a(t)$ is strictly increasing in the interval $[0, t_a]$ and $t_a = w_i^a(t_a)$, we obtain

$$t_s < t_a \Rightarrow t_s < w_i^a(t_s) \tag{10}$$

By replacing t_s with $w_i^s(t_s)$ we obtain

$$\left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_s}{T_j} \right\rceil \times E_j}{P_r} \right\rceil < \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{t_s - O_j}{T_j} \right\rceil \times E_j}{P_r} \right\rceil \tag{11}$$

Finally, we have

$$\sum_{j \leq i} \left\lceil \frac{t_s}{T_j} \right\rceil \times E_j < \sum_{j \leq i} \left\lceil \frac{t_s - O_j}{T_j} \right\rceil \times E_j \tag{12}$$

We know that $t_s \geq t_s - O_j$ because $O_j \geq 0$. Therefore

$$\sum_{j \leq i} \left\lceil \frac{t_s}{P_r} \right\rceil \times E_j \geq \sum_{j \leq i} \left\lceil \frac{t_s - O_j}{P_r} \right\rceil \times E_j \tag{13}$$

Inequality 12 is in contradiction with inequality 13. Thus, we prove that $t_s \geq t_a$.
Knowing that $R_i = t_f - O_i$, we also have $R_i^s \geq R_i^a$ because $t_s - 0 \geq t_a - O_i$.

□

4.3 Optimality

Theorem 2. *PFP_{ASAP} is optimal for the scheduling problem of non concrete tasksets with constrained or implicit deadlines.*

Proof. Let Γ denote a non concrete taskset. We suppose that Γ is feasible using a fixed-priority assignment, but not schedulable with PFP_{ASAP} using the same priority assignment. This means that at least one task denoted as τ_k misses its deadline during the first instance of the worst case scenario (see Theorem 1). Indeed, it is sufficient to consider only the first job because deadlines are constrained or implicit. According to PFP_{ASAP} rules, a deadline miss occurs in the worst case scenario for the k^{th} priority level only if the energy needed to execute priority levels higher or equal to k is greater than the energy that can be replenished from $t = 0$ to the first deadline of τ_k, Inequality 14 summarizes that.

$$D_k \times P_r < \sum_{j \leq k} \left\lceil \frac{D_k}{T_j} \right\rceil \times E_j \tag{14}$$

If PFP_{ASAP} is not optimal, there must be an other fixed-priority schedule for Γ that makes it feasible. Let us suppose that such a schedule exists. This implies that there exists at least one task which is executed even if the energy is not sufficient. This is impossible because the system cannot execute without energy, therefore such a schedule cannot exist. Then we prove that PFP_{ASAP} is optimal for non concrete fixed-priority tasksets with constrained or implicit deadlines.

□

Discussion

The optimality of PFP_{ASAP} relies on the hypothesis fixed in Section 3, mainly the ones about task consumption and replenishment functions. If we relax some of them PFP_{ASAP} may lose its optimality.

Up to now, we have only dealt with linear consumption. If we model consumption as a non linear function, PFP_{ASAP} may not be able to estimate the energy to be replenished to execute exactly one time unit and can lose its optimality.

Tasks consuming less energy than the replenished one are not considered in our model. Including this kind of tasks makes the priority ordering relevant in response time computation and makes the proof we provided insufficient.

In a more realistic model, the replenishment function is not constant. Therefore Equation 14 is no more valid. Thus, we cannot conclude about PFP_{ASAP} optimality. Finally, we have counter examples that prove the non-optimality of PFP_{ASAP} for concrete tasksets.

4.4 Feasibility Condition

A simple way to build a necessary and sufficient feasibility condition for non concrete tasksets is to check if the given taskset is schedulable with PFP_{ASAP} in the worst case scenario, i.e. check if the first job of each task meets its deadline when it is requested simultaneously with the higher priority while the battery is at its minimum level. It consists in computing the worst response time according to PFP_{ASAP} rules for each task and comparing it to its first deadline. Algorithm 2 explains how to do this.

Algorithm 2 Feasibility Test

1: **for** $i = 1 \rightarrow n$ **do**
2: $m \leftarrow 0$
3: $w' \leftarrow \epsilon$
4: **repeat**
5: $m \leftarrow m + 1$
6: $w \leftarrow w'$
7: $w' \leftarrow \left\lceil \frac{\sum_{j \leq i} \left\lceil \frac{w}{T_j} \right\rceil \times E_j}{P_r} \right\rceil$
8: **if** $w' > D_i$ **then**
9: **return** $False$
10: **end if**
11: **until** $w = w'$
12: **end for**
13: **return** $True$

The complexity of Algorithm 2 is $O(m \times n)$ where m is the number of iterations and n is the number of tasks. We note that the number of iterations m depends on the periods and deadlines of the tasks (see line 8) and is bounded by $\max_{\forall i}(D_i)$. Thus, the complexity of Algorithm 2 is pseudo-polynomial. We can reduce this complexity by computing estimations for response times rather than the exact values. However, the feasibility test we propose will not be longer necessary but will remain sufficient.

4.5 Battery Capacity

The design of a system with an arbitrary battery capacity may lead to an overestimated $\mathcal{C}$, which can be very costly (space, weight, money). Finding the lowest battery capacity value is a very important issue.

Given a feasible taskset with $\mathcal{C} = \infty$, the minimum battery capacity issue in harvesting systems is to find the smallest value of $\mathcal{C}$ denoted as $\mathcal{C}^{min}$ that keeps the taskset feasible when we launch the system at the minimum battery level.

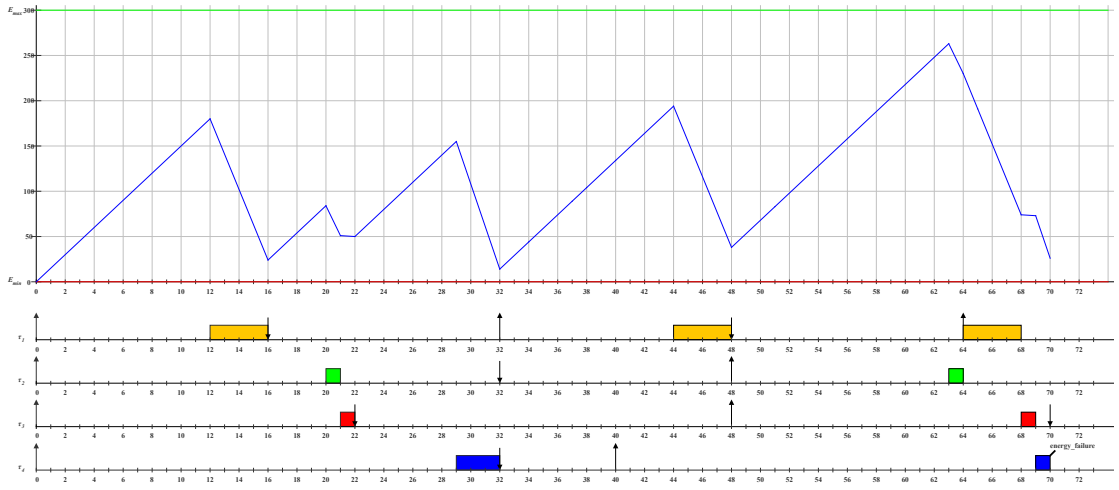


Figure 4: PFP_{ALAP} schedule for taskset Γ_1

The exact value of $\mathcal{C}^{min}$ is difficult to estimate because it depends on the environmental characteristics and the used scheduling algorithm. We can solve the problem by bounding the $\mathcal{C}^{min}$ value, however, in the case of PFP_{ASAP} algorithm, we can compute the exact value.

Algorithm PFP_{ASAP} replenishes the minimum amount of energy needed for only one execution time unit, in this case the minimum battery capacity needed to keep the taskset feasible is the maximum amount of energy that can be consumed during one time unit, i.e. the maximum instantaneous consumption. In our model all the tasks consume energy linearly. Therefore, the minimum battery capacity that keeps the taskset feasible is $\mathcal{C}^{min} = \max_{\forall i}(E_i/C_i)$ in the general case, and $\mathcal{C}^{min} = \max_{\forall i}(E_i/C_i) - P_r$ with the constant replenishment function hypothesis.

If we relax the hypothesis related to the consumption model, a task can at worst consume all its energy cost E_i during the first execution time unit. In this case the minimum battery capacity needed to execute one time unit is the maximum of tasks cost, i.e. $\mathcal{C}^{min} = \max_{\forall i}(E_i)$.

In both cases, the battery capacity cannot be lower because if we deal smaller battery capacity, the system will never be able to execute one time unit and will never have enough energy to execute. Therefore, no other algorithm can run with smaller battery capacity.

5 Performance Evaluation

We proved that PFP_{ASAP} is optimal for non concrete tasksets. In this section we study the behavior of PFP_{ASAP} and we compare it to other algorithms by simulations and analyze its performance.

Some scheduling algorithms and heuristics were proposed in [7, 5]. We selected PFP_{ST} which is not optimal but has the lowest failure rate according to the experiment performed in [7]. We also selected the PFP_{ALAP} algorithm because it can be used to implement a sufficient feasibility condition [5]. In this section, we compare these two algorithms with PFP_{ASAP}, then we analyze their performance.

5.1 Competitors

- PFP_{ALAP} : is a fixed-priority scheduling algorithm that postpones all jobs execution as late as possible, i.e. it introduces idle-periods that consume all available slack-time to charge energy before each job execution. The PFP_{ALAP} algorithm may lose energy if E_{max} is

reached before the end of an idle period. Figure 4 illustrates an PFP_{ALAP} schedule at time interval $[40, 80]$ of the taskset Γ_1 described in Table 2(a),

- PFP_{ST} : is a fixed-priority scheduling heuristic that executes tasks as soon as possible when there is energy available in the battery, and replenishes it when it is not. The replenishment periods are as long as available slack-time. Tasks execution is resumed whenever E_{max} is reached.

5.2 Simulation

In this section, we describe the configuration of the experiments, namely the simulation tool, the input data, the parameters and the set assumptions.

5.2.1 Simulation Tool

To perform such an experiment, we need a simulation tool able to run large scale simulations on various data and algorithms. We used YARTISS, a simulation tool presented in [6]. It provides a simulation framework able to run the simulation of a large set of tasksets on different energy parameters and according to different scheduling algorithms simultaneously. It can also provide statistics about the performed simulations like the failure rate, the preemption rate or the average battery level during the simulations and many other metrics. The tool is available on-line in [1].

5.2.2 Input Data

For these simulations we used an adapted version of the UUniFast-Discard algorithm [4] coupled with a limitation of hyper-period technique [13] to generate tasksets. The generated tasksets respect the following hypotheses:

- all tasksets are time feasible,
- time and energy are discretized, this means that they are integers and all scheduling operations are performed before or after one time unit,
- the charging function P_r is constant, i.e. a constant amount of energy is added to the battery level in every time unit,
- tasks consume energy linearly, i.e. a task consumes E_i/C_i energy units for each execution time unit.

In order to represent most of the possible tasksets, we generate them according to their processor and energy utilizations, i.e. U^p and U^e. We vary U^p and U^e in the interval $[0.2, 1]$ to obtain a couple of (U^p, U^e) for each 0.05 unit of U^p and U^e. Then, we obtain 350 distinct tasksets for each couple (U^p, U^e).

In this paper we restricted the study to non concrete tasksets. Therefore, all tasksets are simulated in the worst case scenario. The data used for our simulations are available online in [1].

5.2.3 Simulation Description

In order to evaluate the behavior of the compared algorithms, we vary some parameters, namely the battery capacity $\mathcal{C}$ and the number of tasks per taskset. Firstly, we vary $\mathcal{C}$ in six energy scenarios to analyze its effect on the failure rate. Secondly, we vary the number of tasks per taskset in several distinct simulations to observe the evolution of the scheduling overhead of each algorithm.

We set the remaining parameters to the same values that we used for tasksets generation in order to fit the considered assumptions. For these experiments, we set these parameters as follows, $P_r = 15$, $E_{min} = 0$. The simulations are executed for 3000 discretized time units. Furthermore,

tasksets are run for more time than one hyper-period that is bounded by 2500 time units. Thus, if a taskset does not miss any deadline during the simulation time, the taskset is said to be feasible. We use Deadline Monotonic policy (DM) to assign priorities because of its optimality for the classical scheduling problem. However, DM policy loses its optimality when we integrate energy constraints (tasks consumption, battery capacity) but according to preliminary experiments, it is still dominating Rate Monotonic.

Several statistical metrics are computed during simulations. These metrics give information about algorithms behavior. For our experiments we selected the following metrics: *failure rate*, *Preemption count*, *Average overhead*, *Average idle-period*, *Average busy-period* and *Average energy level*.

5.2.4 Metrics Definition

Failure Rate is the percentage of non feasible taksets among all the tested ones.

Preemption Count for one simulation, it represents the number of preemption events. A preemption event occurs when a job is stopped while it is still not finished. All of the events that occur at the same instant are considered once. Therefore, the number of possible events is bounded by the number of time units composing the simulation, i.e. the simulation duration. For several simulations, this metric is computed only for feasible tasksets and represents the ratio of the average number of preemptions relative to all possible events.

Average Overhead it is the amount of time spent while handling a scheduling event. For one simulation, this metric represents the average overhead of all of the scheduling events. Its exact value is difficult to compute. We simply calculate an estimate by distributing the real simulation time (in milliseconds) upon the number of scheduling events. The simulation tool that we use is event-based. Therefore, only the events handling consume processor time. Thus, it gives us an acceptable estimation of the average overhead.

Average Idle-Period represents the average duration of periods when the processor is idle. It includes battery replenishment and slack-time periods. For several simulations, we compute the ratio of the average idle-period duration relative to the simulation duration.

Average Busy-Period is the average duration of continuous processor activity. For several simulations, we compute the ratio of the average busy-period duration relative to the simulation duration.

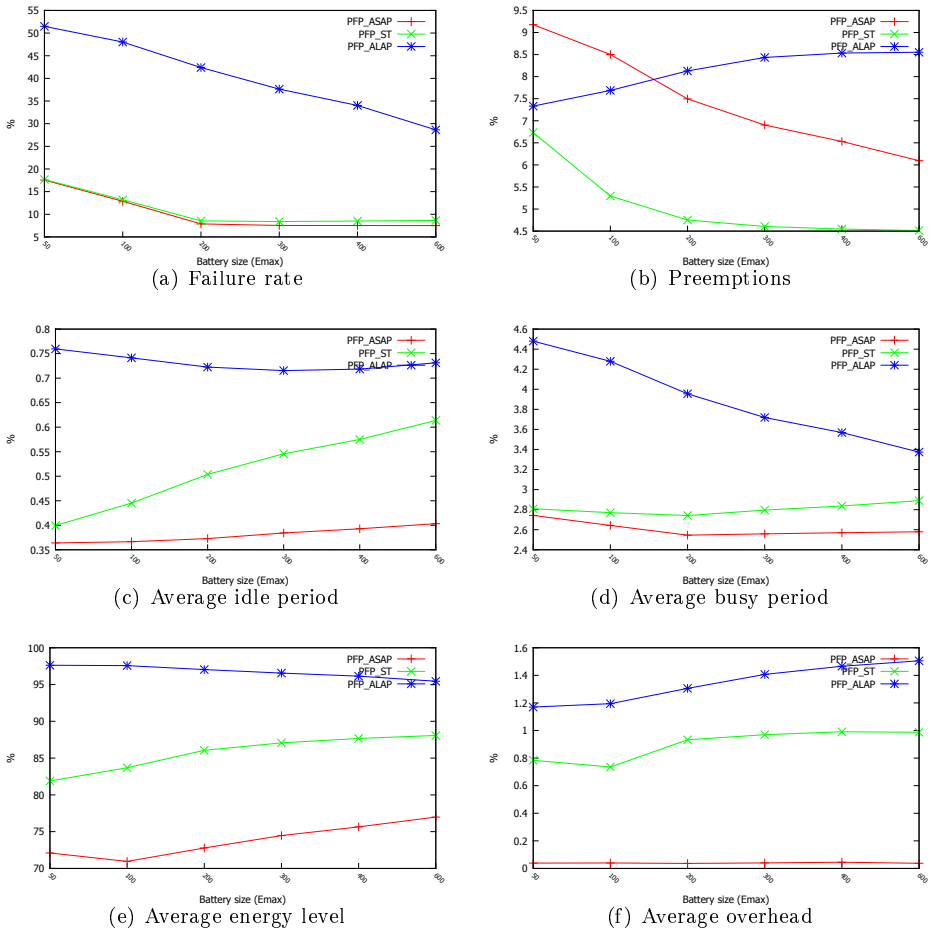
Average Energy Level represents the average energy level of the battery or capacitor at any instant during the simulation. It is the average of the energy level of all scheduling events.

5.2.5 Metrics Relevance

Failure Rate the greater the failure rate, the lower the algorithm performance.

Preemption Count the greater the number of preemptions, the greater the context switch. This increases the overhead cost and decreases performance, which makes the algorithm unusable in practice.

Average Overhead the greater the average overhead, the greater the timing constraints violation risk. This makes the algorithms unusable in practice.



(a) Failure rate (b) Preemptions

(c) Average idle period (d) Average busy period

(e) Average energy level (f) Average overhead

Figure 5: Comparison between PFP_{ALAP}, PFP_{ST} and PFP_{ASAP}

Average Idle-period and Average Busy-Period the relevance of these two metrics is closely linked to the number of preemptions. The longer the idle/busy periods, the lesser the number of preemptions and the higher the algorithm performance. Therefore, the longer the idle-periods, the higher the average energy level and the lesser energy-constrained the system.

Average Energy Level the best algorithm relative to this criterion is one which maximizes the average energy level. This means that the algorithm makes the system less energy constrained.

5.3 Results Analysis

5.3.1 The Variation of E_{max}

Figure 5 presents the results of comparing the algorithms. In the following part, we analyze the effect of E_{max} varying on the performance of each algorithm for each metric:

Failure Rate The increase of E_{max} reduces the failure rate of all the evaluated algorithms. This result was expected because the more E_{max} is increased the less the system is energy-constrained. We also observe that PFP_{ALAP} has the highest failure rate for all values of E_{max}. Both of PFP_{ASAP} and PFP_{ST} have a lower failure while PFP_{ASAP} demonstrates the lowest one. To explain why PFP_{ST} fails to schedule some tasksets which are schedulable with PFP_{ASAP} we

have to examine its behavior closely. When the battery is down, PFP_{ST} suspends the system as long as possible before the next execution while PFP_{ASAP} suspends the system for only one time unit. In this case PFP_{ST} may uselessly postpone executions and may accumulate an unbearable energy load for a future time. This can lead the system to replenish more time than the slack-time available and may lead to missing deadlines. PFP_{ALAP} suffers from the same problem as PFP_{ST} because it postpones execution as long as possible regardless of the battery level. When all jobs are postponed to a maximum and the system incurs a long execution period, it is impossible to introduce more replenishment periods. Therefore, deadline misses may occur. Figure 4 illustrates a deadline miss caused by delay similar to the one explained for PFP_{ST}. At time 63 a long busy period begins and the system has already consumed all the slack-time. The energy replenished during the former idle periods is not sufficient and the system runs out of energy. The PFP_{ASAP} algorithm allows to avoid this situation by starting jobs immediately.

As shown in Section 4, PFP_{ASAP} is optimal for non concrete tasksets, the simulations confirm that. All of the tasksets that are feasible with PFP_{ALAP} and PFP_{ST} are still feasible with PFP_{ASAP} but not the reverse. However, the difference in the failure rate between PFP_{ST} and PFP_{ASAP} is still negligible, the study of the rest of metrics may be crucial.

Preemption Rate The simulations show that increasing E_{max} helps to stabilize the number of preemptions. However PFP_{ASAP} demonstrates a very high number of preemptions regardless E_{max} values. By construction, PFP_{ASAP} executes for one time unit then preempts tasks to check again if there is enough energy, while PFP_{ST} consumes all the slack-time available to replenish energy and avoid preemptions due to a lack of energy. PFP_{ALAP} does the same for each job activation.

Average Overhead We observe that for every value of E_{max}, PFP_{ALAP} and PFP_{ST} have much higher average overhead than PFP_{ASAP}. This is due to the pseudo-polynomial complexity of the slack-time algorithm [9]. PFP_{ALAP} computes slack-time whenever a job is requested and PFP_{ST} does the same only if there is not enough energy while PFP_{ASAP} only needs to order the activated jobs.

Average Idle-Period and Busy-Period These two metrics are closely linked to the number of preemptions, the longer the idle or busy periods, the lower the number of preemptions. PFP_{ASAP} maximizes the number of preemptions, therefore, it has shorter idle and busy periods. PFP_{ST} consumes all available slack-time to replenish energy, then, executes tasks while the battery level is sufficient. This maximizes the duration of both of the idle and the busy periods.

Average Energy Level regardless E_{max} values, PFP_{ALAP} has the highest average energy level and PFP_{ASAP} has the lowest one. We expected this result because the PFP_{ST} replenishes energy during long periods (idle periods) which increase the average battery level.

5.3.2 Varying Taskset Cardinal

The aim of this experiment is to study the effect of the taskset cardinal on the average overhead. We performed the simulations and the results confirmed our previous observations. Both of PFP_{ALAP} and PFP_{ST} have a very large overhead relative to PFP_{ASAP}. This is explained by the complexity of the slack-time computation algorithm.

Table 1 summarizes the performance of the evaluated algorithms.

PFP_{ASAP} is optimal for non concrete tasksets and has the lowest failure rate compared to the other algorithms and need less battery capacity to operate. However, it increases the number of preemptions and context switches which can be considered to be a great disadvantage. PFP_{ST} is not optimal but simulations show that its failure rate is very close to that of PFP_{ASAP}. Furthermore, it maximizes the average energy level and reduces preemptions. The pseudo-polynomial

-	PFP_{ALAP}	PFP_{ST}	PFP_{ASAP}
Worst case scenario	-	-	synchronous activations
Optimality	bad	bad	good
Failure rate	bad	good	good
Average overhead	bad	bad	good
preemptions	neutral	good	bad
Average idle-period	neutral	good	bad
Average busy-period	neutral	good	bad
Average energy level	neutral	good	bad

Table 1: PFP_{ALAP} vs PFP_{ST} vs PFP_{ASAP}

complexity of stack-time calculation is the main drawback of PFP_{ST}, then, it cannot be used for systems with a large number of tasks. Concerning PFP_{ALAP}, in addition to its non-optimality, simulations demonstrate very bad performance for all metrics.

Finally, we can conclude that PFP_{ASAP} is optimal but not applicable in practice because of its preemption rate. However, despite the great complexity of PFP_{ST} and its non optimality, it is still the only algorithm that shows many advantages. It may be very interesting to study the possibility of significantly reducing the complexity of the slack-time computation by dealing with approximated values.

6 Conclusion and Future Work

This paper addresses the problem of fixed-priority scheduling for energy harvesting real-time systems. We proposed PFP_{ASAP}, an optimal algorithm for non concrete tasksets, then we built a necessary and sufficient feasibility condition based on PFP_{ASAP} algorithm. We also proved that the worst case scenario for PFP_{ASAP} occurs whenever all tasks are requested simultaneously while the battery is at its minimum level E_{min}. We performed large scale simulations to evaluate PFP_{ASAP} performance and compared it to other algorithms. The experiment showed that the main drawback of PFP_{ASAP} is its very large number of preemptions.

Moreover, the PFP_{ASAP} algorithm is only optimal for non concrete tasksets. As future work, we will explore other scheduling policies and look for more general feasibility condition. Firstly, we plan to study the possibility of finding a clairvoyant algorithm based on PFP_{ASAP} that can be optimal for both concrete and non concrete tasksets. Then, we will try to optimize the number of preemptions to make it usable in practice. Secondly, we will study the possibility of using our feasibility test to build an optimal priority assignment (OPA) based on Audsley's algorithm [3] and Davis' criteria [8]. Finally, we will be interested in measuring the effect of the assumptions we set on both replenishment and task consumption functions, indeed, we will try to find the worst case of both consumption and replenishment models.

References

[1] D. Zhu, X. Qi, and H. Aydin, "Priority-Monotonic Energy Management for Real-Time Systems with Reliability Requirements," in *Proceedings of the 25th International Conference on Computer Design*, 2007.

[2] D. Zhu and H. Aydin, "Energy management for real-time embedded systems with reliability requirements," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006.

[3] A. Allavena and D. Mossé, "Scheduling of frame-based embedded systems with rechargeable batteries," in *Workshop on Power Management for Real-Time and Embedded Systems (in conjunction with RTAS)*, 2001.

[4] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Real-time scheduling with regenerative energy," in *Proceedings of 18th Euromicro Conference on Real-Time Systems*, 2006.

[5] R. Jayaseelan and T. Mitra, "Estimating the Worst-Case Energy Consumption of Embedded Software," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[6] H. EL Ghor, M. Chetto, and R. H. Chehade, "A real-time scheduling framework for embedded systems with environmental energy harvesting," *Computers and Electrical Engineering*, vol. 37, pp. 498–510, July 2011. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2011.05.003

[7] M. Chetto, D. Masson, and S. Midonnet, "Fixed Priority Scheduling Strategies for Ambient Energy-Harvesting Embedded systems," in *Proceedings of IEEE/ACM International Conference on Green Computing and Communications*, 2011.

[8] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks fixed priority preemptive systems," in *Proceedings of the 13th IEEE Real-Time Systems Symposium*, 1992.

[9] Y. Chandarli, Y. Abdeddaïm, and D. Masson, "The Fixed Priority Scheduling Problem for Energy Harvesting Real-Time Systems," in *Proceedings of WIP RTCSA*, 2012.

[10] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, and M. Qamhieh, "YARTISS : A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms," in *Proceedings of 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2012.

[11] "YARTISS," 2012. [Online]. Available: http://yartiss.univ-mlv.fr/

[12] 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.

[13] C. Macq and J. Goossens, "Limitation of the hyper-period in real-time periodic task set generation," in *Proceedings of the 9th international conference on real-time systems*, 2001.

[14] R. I. Davis, K. Tindell, and A. Burns, "Scheduling slack time in fixed priority pre-emptive systems," in *Proceedings of the 14th IEEE Real-Time Systems Symposium*, 1993.

[15] N. Audsley, *Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times*, ser. Technical report. University of York, Department of Computer Science, 1991. [Online]. Available: http://books.google.fr/books?id=ooZkGwAACAAJ

[16] R. I. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, 2009.