Optimal Real-Time Scheduling Algorithm for Fixed-Priority Energy-Harvesting Systems

Younès Chandarli, Yasmina Abdeddaïm, Damien Masson

Optimal Real-Time Scheduling Algorithm for Fixed-Priority Energy-Harvesting Systems

Younès Chandarli, Yasmina Abdeddaïm and Damien Masson

October 20, 2014

1 Introduction

In [4] we saw that finding efficient scheduling algorithms for fixed-priority energy-harvesting systems is one of the challenges of this research area. In [2], we presented PFP_{ASAP} which is an optimal scheduling algorithm. Moreover, the optimality of this algorithm relies on two main assumptions: the considered task sets are energy-non-concrete, and all the tasks consume more energy than it is replenished. Unfortunately, removing one of these two assumptions leads PFP_{ASAP} to lose its optimality. This is due to the fact that without these assumptions, the worst-case scenario of PFP_{ASAP} is no longer the synchronous activation with the minimum battery capacity. Moreover, without these assumptions, the worst-case scenario is unknown up to now. There exist some counter examples that prove the non-optimality of PFP_{ASAP} (see Figures 4 and 1).

The challenge now is to understand why does PFP_{ASAP} lose its optimality and we try to study deeply the fixed-priority scheduling for energy-harvesting systems by trying to build an optimal algorithm or otherwise to prove the nonexistence of such an algorithm.

In this work, we explore different intuitive ideas of scheduling algorithms and we explain why they are not optimal through counter examples. Then, we show the difficulty of finding an optimal algorithm or proving the nonexistence of such an algorithm with a reasonable complexity.

The remainder of this report is organized as follows. In section 3 we define and prove some properties of fixed-priority scheduling for energy-harvesting systems. After that, we explore in Section 4 different ideas of scheduling algorithms and we discuss the existence of an optimal algorithm. Finally, we conclude the report with Section 5.

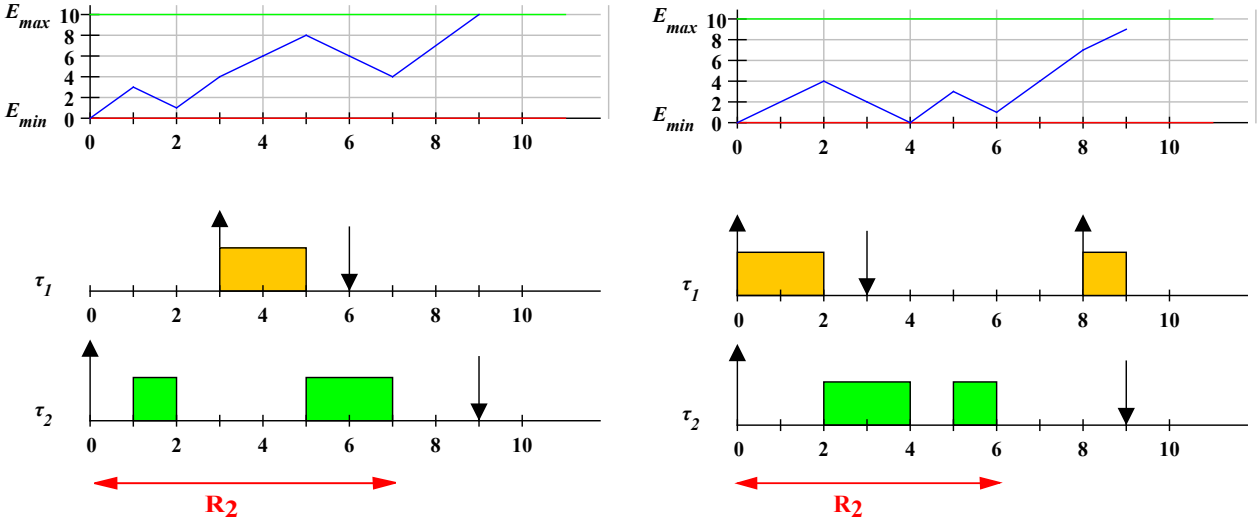
1.1 Task Model

The task model considered here is an extension of Liu and Layland's model that considers task energy consumption in addition to the classical parameters. Then, we consider a real-time task set in a renewable energy environment defined by a set of n periodic/sporadic 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 relative deadline D_i and its *worst-case energy consumption* E_i.

Therefore, a task τ_i releases an infinite number of jobs separated by at least T_i time units and each job is executed during C_i time units and consumes E_i energy units and must finish before D_i time units after being requested. Moreover, the deadlines are constrained

Tasks	C_i	E_i	T_i	D_i
τ_1	2	2	8	3
τ_2	3	15	10	9

(a) Task set



(b) Asynchronous requests

(c) Synchronous requests

Figure 1: Worst-case scenario counter example

or implicit and the task set is priority-ordered such that task τ_n is the task with the lowest priority.

2 General Model

An energy-harvesting system being composed of a real-time computational part should be first described as a classical real-time system with a set of recurrent tasks and second as a more complex system by including the energy source and the energy storage constraints. In this section we specify the general formal model considered by this dissertation.

2.1 Energy Model

We consider an embedded system connected to an energy harvesting device. An energy-harvesting device is a system that collects the ambient energy from the environment using an energy-harvesting technique. The collected energy is then stored in an energy storage unit with fixed capacity (e.g. rechargeable battery or supercapacitor).

Replenishment: We suppose that the amount of energy that arrives into the storage unit is a function of time which is either known or bounded. Recall that the profile of energy arriving from the harvester depends on the energy source and the harvesting technique. Most of energy sources are unpredictable or predictable with difficulty, for example, solar energy harvesting depends on brightness intensity which cannot be predicted accurately. For this reason, since the scope of this dissertation is hard real-time

scheduling for energy-harvesting systems, we consider only energy sources and harvesting techniques that can provide a predictable profile of energy or that can be lower bounded. Hopefully, such energy profiles exist especially with vibration energy source and piezo-electric harvesting technique which seem to be suitable for small embedded systems. It provides a nearly stable output of energy even with a deviation of vibration frequency up to 40% form the optimal one as explained in [1]. Therefore, as a first step, we can consider a uniform or a bounded replenishment function which means that the storage unit receives the a constant amount of energy every time unit. We denote $P_r(t)$ 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 Equation 1.

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

As mentioned above, we assume that that $P_r(t)$ is 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 Equation 2.

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

In the remainder of this dissertation, we use P_r instead of $P_r(t)$ to denote the replenishment function and we suppose that it is lesser than or equal to the battery capacity.

Storage: The replenishment of the storage unit is performed continuously even during jobs execution and the level of the stored energy 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. This capacity is denoted $\mathcal{C}$. 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 some tasks cannot be executed. We suppose also that the storage device is carefully selected to ensure regular behavior, i.e. regular replenishment and regular discharge in order to avoid charge/discharge speed variations and capacity losses due to numerous charge/discharge cycles. Recall that supercapacitors can offer these requirements.

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 $E(t)$. Below, we use the word "battery" to refer to the energy storage unit in order to simplify the language.

Consumption: Tasks energy cost should actually include not only dynamic and static processor energy consumption but also the consumption of other devices that a task can use, e.g. sensors and data transmission devices. Moreover, even if we consider only processor consumption, the global consumption depends much more on the kind of circuitry used by the code than on the execution duration [6]. For this reason we consider that 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 that do not use the same devices, then, we can have $C_i < C_j$ and $E_i > E_j$. Furthermore, we consider that energy consumption is function of time that is in reality not necessarily uniform. Actually, since tasks can use different devices, it is difficult to predict the accurate energy profile of tasks. Moreover, the worst energy consumption profile is not known up to now. This is a serious issue for real-time predictability, however, including this constraint to scheduling decisions makes

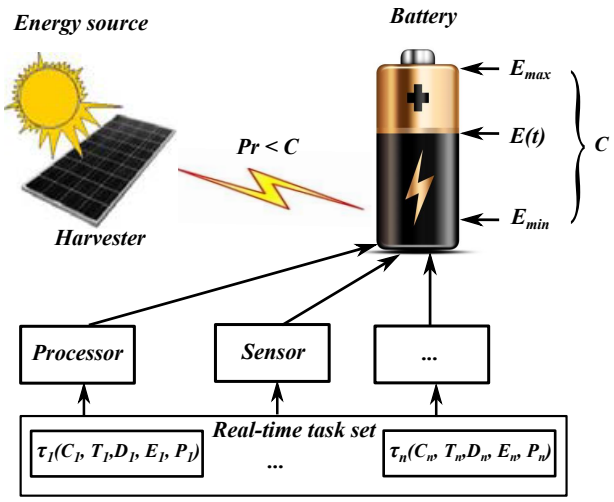


Figure 2: Energy-harvesting system model

it a hard problem with many parameters to consider. As a first step and for the sake of simplicity, we consider for the scope of this dissertation that the energy consumption function is different from task to task but linear, which means that each task has its own constant instantaneous consumption E_i/C_i.
Figure 2 recapitulates these descriptions.

3 Definitions and Notations

In order to facilitate the understanding of the next sections, we first redefine and prove some properties.

3.1 Model and Notations

We consider the following notations.

- $s_{i,j}$: the starting time of job $J_{i,j}$,
- $a_{i,j}$: the next activation time of task τ_i after time t,
- $c_i(t)$: the remaining execution time of the current job of task τ_i at time t. It is equal to 0 if the job is already finished,
- $e_i(t)$: the remaining energy cost of the current job of task τ_i at time t. It is equal to 0 if the job has finished its execution,
- $d_i(t)$: the absolute deadline of the current job of task τ_i at time t, it does not exist if the job is not yet activated,
- $s_i(t)$: the execution starting time of the current job of task τ_i at time t, it is undefined if the job is not yet activated.

3.2 Definitions

Definition 1 (Energy Demand)**.** *The energy demand of priority level-i of time interval* $[t_1, t_2[$ *denoted* $We_i(t_1, t_2)$ *is the amount of energy to be consumed by the execution of the jobs of priority levels* $1, \ldots, i-1, i$ *that are requested within interval* $[t_1, t_2[$ *or are pending at time* t_1*. It can be obtained by Equation 3.*

$$We_i(t_1, t_2) = \sum_{j \leq i} e_j(t_1) + \left\lceil \frac{t_2 - a_j(t_1)}{T_j} \right\rceil \times E_j \tag{3}$$

The intuition behind Equation 3 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 that are requested during the time interval $[t_1, t_2[$. The energy demand of time interval $[0, t[$ is just noted $We_i(t)$ and can be obtained by Equation 4.

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

Definition 2 (Energy Budget)**.** *The energy budget of the system during time interval* $[t_1, t_2]$ *denoted* $Bu(t1, t2)$ *is the amount of energy available until time* t_2*, i.e. the battery level at time* t_1 *plus the energy replenished during time interval* $[t_1, t_2]$*. It can be computed by Equation 5.*

$$Bu(t_1, t_2) = E(t_1) + \int_{t_1}^{t_2} P_r(t) \ dt \tag{5}$$

Definition 3 (Energy Balance)**.** *The energy balance of a job* $J_{i,j}$ *at time t denoted* $Ba_i(t)$ *is the difference between the energy budget between time t and the deadline of* $J_{i,j}$*, and the energy demand of the same priority level and the same time interval. It can be obtained by Equation 6.*

$$Ba_i(t) = Bu(t, d_{i,j}) - We_i(t, d_{i,j}) \tag{6}$$

We notice that if the time t does not coincide with a request time of task τ_i and the previous job has already finished its execution, we must include the execution of the lower priority jobs between time t and the next request time of priority level-i, i.e. $a_i(t)$, because they consume energy before time $a_i(t)$ and can change the energy balance at time $d_{i,j}$ as illustrated in Figure 3. However, these lower priority executions units depend on the used scheduling algorithm, i.e. energy-work-conserving or not. This limitation is discussed in Section 4.

3.3 Energy-Work-Conserving

Without taking into account energy constraints, the work-conserving property in real-time scheduling means that the scheduling algorithm does not add idle times when there is at least one job ready to execute. However, when we consider energy-harvesting constraints, scheduling algorithms may add necessary idle periods to replenish energy. Then, this notion is extended to energy-work-conserving to include replenishment time.

Furthermore, in the energy-harvesting context, a scheduling algorithm is considered as *energy-work-conserving* if it schedules jobs as soon as they are ready to execute and the energy is sufficient to execute at least one time unit.

This definition means that scheduling algorithms do not replenish energy more than needed, otherwise, the algorithm is considered as non-energy-work-conserving.

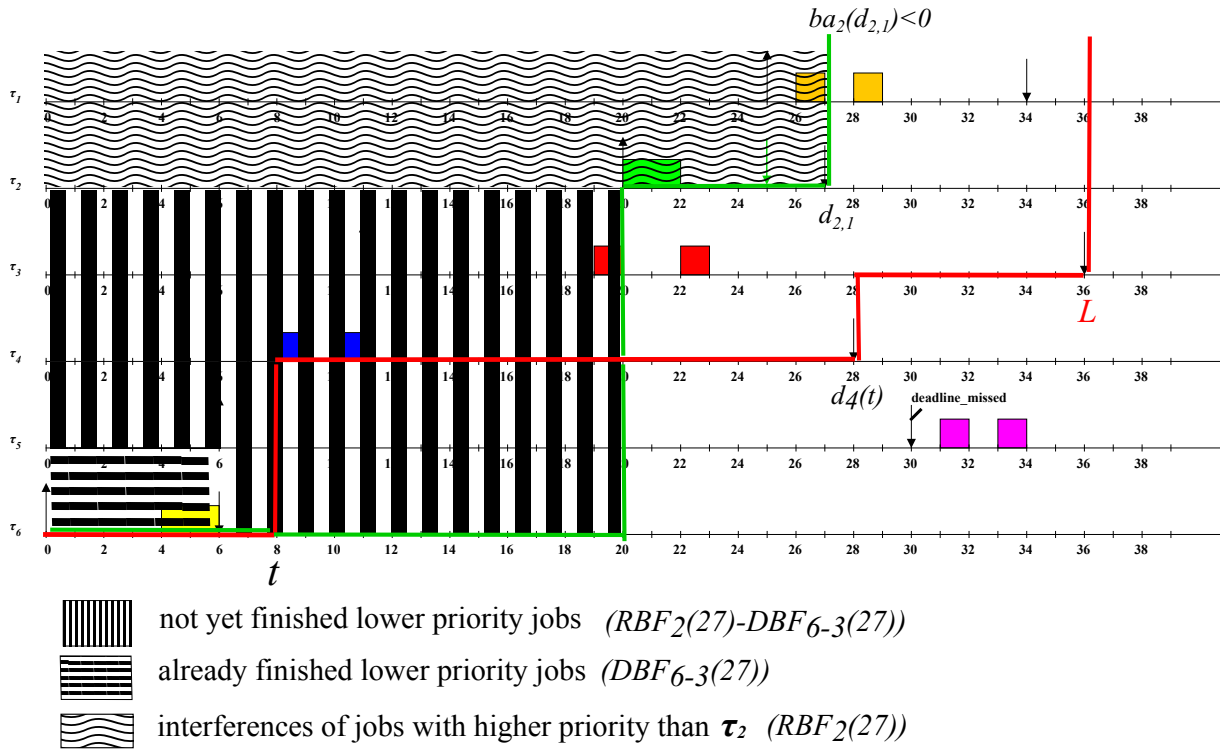
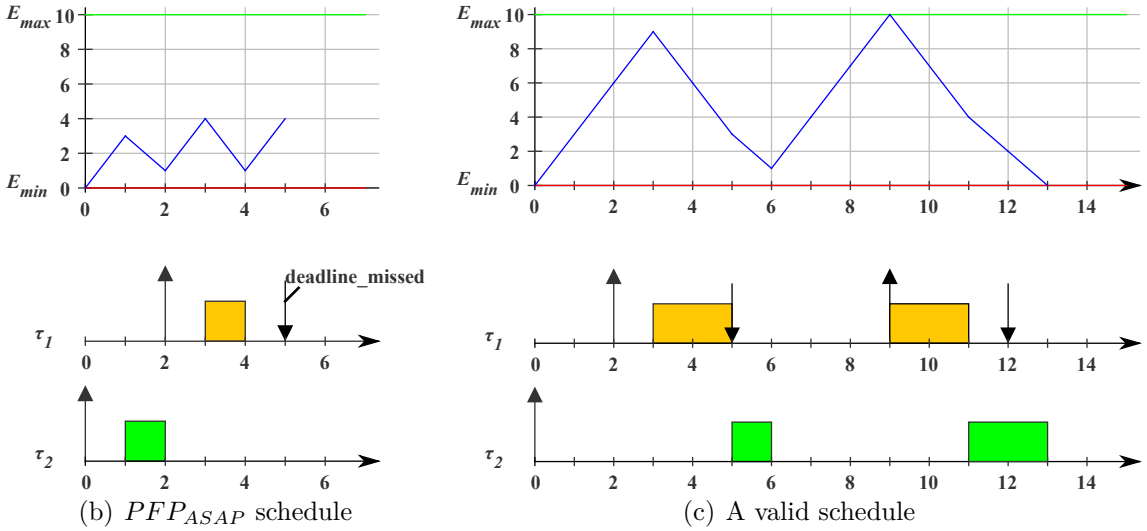


Figure 3: Energy balance of $J_{2,1}$ at time t=8

-	O_i	C_i	E_i	T_i	D_i	P_i
τ_1	2	2	12	10	3	1
τ_2	0	3	15	15	15	2

(a) $E_{max} = 10$, $E_{min} = 0$ and $P_r = 3$



(b) PFP_{ASAP} schedule (c) A valid schedule

Figure 4: PFP_{ASAP} is not optimal

Unfortunately, PFP_{ASAP}, which is an energy-work-conserving algorithm, loses its optimality when we consider tasks with offsets or an initial battery level higher than E_{min}. Figure 4 illustrates a counter example. We can see in Figure 4(b) that the lower priority

task τ_2 is executed before τ_1 following the energy-work-conserving principle, i.e. as soon as the energy is sufficient to execute, it consumes the energy needed for the higher priority task τ_1 which needs more time than its deadline to replenish the required energy. We can see that in such a situation, executing as early as possible can lead to a deadline miss while delaying the execution of lower priority tasks can avoid missing deadlines as shown in Figure 4(c). Following this intuition, we propose Lemma 1.

Lemma 1. *The energy-work-conserving scheduling is not optimal for the scheduling problem of fixed-priority energy-harvesting systems.*

Proof. To prove this property we just have to find an example where a valid energy-work-conserving schedule is not possible while a valid schedule exists.

Let us consider a task set denoted Γ composed of two tasks τ_1 and τ_2 with the following configuration:

$$\begin{aligned}
&D_2 = 2 \times C_2 + C_1 \\
&D_1 = C_1 \\
&O_1 > O_2 \\
&C_1 + C_2 = O_1 + D_1 \\
&E_2 = \int_0^{O_1} P_r(t).dt \\
&E_1 + E_2 > E(0) + \int_0^{O_1+D_1} P_r(t).dt \\
&E_1 + E_2 \leq E(0) + \int_0^{D_2} P_r(t).dt \\
&T_1 = T_2
\end{aligned} \tag{7}$$

We can see that it is possible to finish executing τ_1 before O_2 according to an energy-work-conserving scheduling. However, it is not possible to schedule both τ_1 and τ_2 inside time interval $[0, O_1 + D_1]$ because the available energy is not sufficient (see Condition 7). Thus, more delay is needed to harvest more energy. Then, executing τ_1 before τ_2 and delaying τ_2 leads to miss the deadline of τ_2. Moreover, delaying τ_1 leads to add idle times while there is an active job and enough energy to execute immediately which violates the property of energy-work-conserving scheduling. In this case, it is impossible to produce a valid schedule with an energy-work-conserving scheduling. Then, we prove that energy-work-conserving fixed-priority scheduling cannot be optimal. □

3.4 Energy-Lookahead Scheduling

In the classical real-time scheduling theory, a lookahead algorithm is an algorithm that is able to predict future job requests, for example in sporadic or periodic task models. However, in energy-harvesting model there is a new parameter subject to fluctuations: the incoming energy or the replenishment function. Therefore, we need to redefine the term of lookahead.

Definition 4 (Energy-Lookahead)**.** *A energy-lookahead scheduling algorithm attempts to foresee the effects of a scheduling decision to evaluate the schedulability of future jobs. The aim of lookahead is to chose the best scheduling decision that does not lead to avoidable deadline misses. The clairvoyance includes the battery replenishment function as well as tasks inter arrival times. In the opposite, if the algorithm does not consider the future state of the system, then, it is said non-energy-lookahead.*

-	O_i	C_i	E_i	T_i	D_i	P_i
τ_1	28	2	38	80	40	1
τ_2	7	2	32	16	8	2
τ_3	3	2	14	80	70	3
τ_3	0	1	12	68	44	4

Table 1: Task set Γ with $P_r = 3$, $E_{max} = 100$, $E_{min} = 0$ and $E_0 = 6$

This means that the algorithm has knowledge a priori of the future state of the system, namely jobs activation times and energy replenishment function. The lookahead or clairvoyance consists of computations or scheduling simulation performed over a future interval of time that we call the *lookahead window*.

Definition 5 (Lookahead Window)**.** *The lookahead or clairvoyance window for a priority level-i at time t is the shortest time interval* $[t, t + L[$ *such that the scheduling decision of priority level-i at time t does not impact the scheduling decisions of time interval* $[t+L, \infty)$*.*

Using this definition, we propose Conjecture 1.

Conjecture 1. *No non-energy-lookahead scheduling algorithm is optimal for fixed-priority energy-harvesting systems.*

Insight. From Lemma 1 we know that non-work-conserving scheduling is needed for an optimal algorithm. This means that additional delays are needed to ensure meeting deadlines and energy requirements. It is obvious that the optimal length of these delays depends on the available energy and the potential interferences, which means that the replenishment function and the request times of higher priority jobs are needed to compute the right delay for each job. Therefore, delaying execution without any knowledge about the future incoming energy and the future activation times of higher priority jobs may lead to too short or too long idle periods which can compromise the respect of deadlines.

4 Algorithms

From Lemma 1 and Conjecture 1, we can consider that an optimal algorithm for fixed-priority energy-harvesting systems must be non-work-conserving and energy-lookahead. In this section we explore some scheduling algorithms and heuristics that attempt to be optimal. We start by showing a counter example that is feasible with fixed-priority scheduling but is not schedulable with all the fixed-priority algorithms presented until now in this dissertation. Then, we discuss the possibility of finding or building an optimal algorithm.

The task set described in Table 1 shows many situations that make the fixed-priority scheduling for energy-harvesting systems difficult. In the following we explain why each scheduling algorithm fails to schedule this task set in this configuration while a feasible schedule exists: Figure 5 shows the beginning of such a schedule, and we know that is feasible because there is no deadline miss within twice the hyper-period.

PFP_{ASAP}**:** the first intuitive scheduling algorithm for fixed-priority energy-harvesting systems is to use the classical FTP algorithm and add replenishment periods when E_{min}

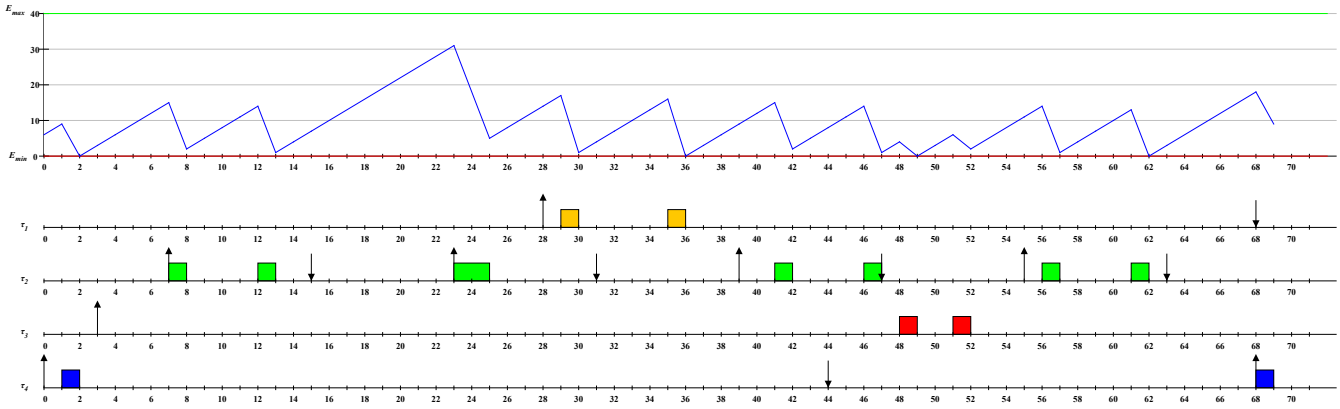


Figure 5: A feasible schedule of task set Γ

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
```

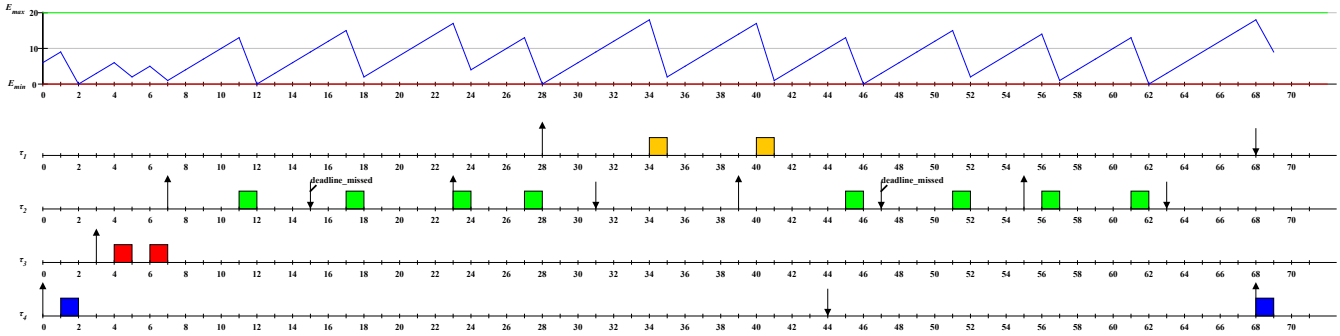


Figure 6: PFP_{ASAP} counter example

is reached. The PFP_{ASAP} policy behaves so. As described in [2], it schedules tasks as soon as possible when the energy is sufficient and replenishes otherwise. The replenishment periods are as long as needed to execute one time unit of the higher priority job ready to execute (see Algorithm 1).

Again, PFP_{ASAP} is not optimal because executing as soon as possible maximizes the energy demand within a short interval of time which can lead to a lack of energy and then a deadline miss. As we can see in Figure 6, jobs of lower priority than $J_{2,1}$, namely $J_{3,1}$ and $J_{4,1}$, are executed at early as possible and consume the energy needed for a higher priority job that is requested few instant later, i.e. $J_{2,1}$. Job $J_{2,1}$ misses its deadline while

Algorithm 2 PFP_{ALAP} Algorithm

```
1: t ← 0
2: loop
3:    if ST(t) ≥ 0 then
4:       τ_k ← the highest priority active task at time t
5:       execute τ_k
6:    else
7:       idle the system to replenish energy
8:    end if
9:    t ← t + 1
10: end loop
```

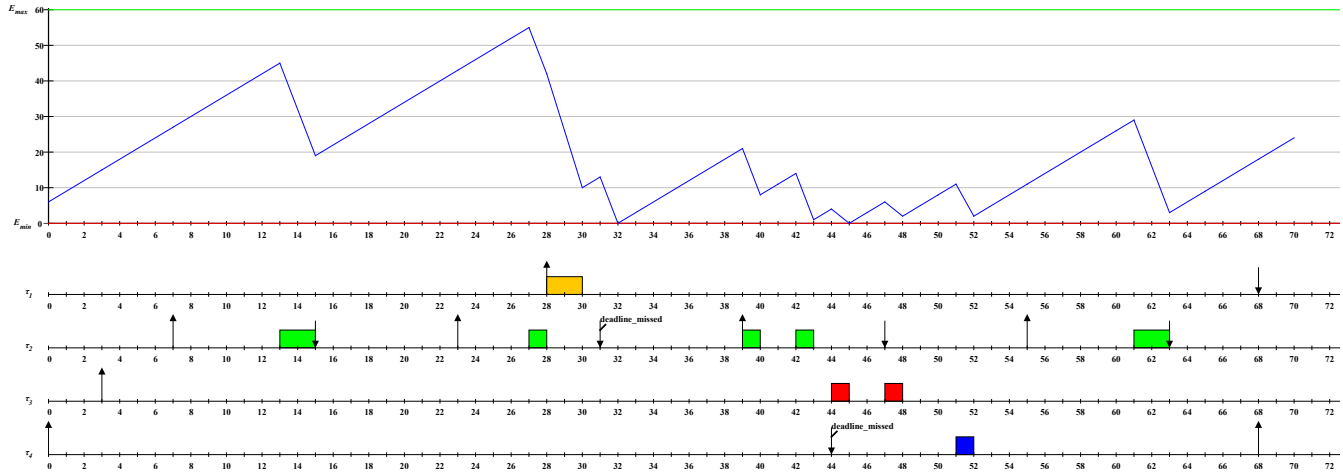


Figure 7: PFP_{ALAP} counter example

the lower priority jobs, namely $J_{3,1}$ and $J_{4,1}$, can be delayed to avoid this situation.

PFP_{ALAP}**:** the second intuitive idea is to schedule jobs prior to their deadlines in order to permit a maximum replenishment of energy before executing. The PFP_{ALAP} algorithm was proposed based on this intuition. PFP_{ALAP} postpones jobs executions as long as possible all the time. Whenever there is available slack-time, executions are delayed (see Algorithm 2).

Unfortunately, this is one of the counter intuitive ideas of fixed-priority scheduling for energy-harvesting systems. In fact, the PFP_{ALAP} algorithm is not optimal also even with an unlimited battery capacity because the computation of slack-time does not consider energy constraints. As we can see in Figure 7, a deadline miss can occur while a feasible schedule exists (Figure 5). A deadline miss occurs at time 31 when the energy balance of time interval $[23, 31]$ is negative even though there is available slack-time. The energy available until the deadline of job $J_{2,2}$ is lesser than the energy demand of the same time interval. This negative energy balance is due to the fact that delaying job too much $J_{2,2}$ leads the system to anticipate the execution of the higher priority job $J_{1,1}$ which increases the energy demand of time interval $[23, 31]$ and leads to an insufficient energy to finish executing $J_{2,2}$ before its deadline. This phenomena is due to the fact that the slack-time computation used for this algorithm does not consider the energy requirements of the

Algorithm 3 PFP_{ST} Algorithm

```
1: while true do
2:   while there is ready jobs do
3:     while E(t) > E_min do
4:       execute jobs according to fixed-priority rules
5:     end while
6:     while E(t) < E_max and ST(t) > 0 do
7:       idle the system to replenish energy
8:     end while
9:   end while
10:  while there is no active jobs do
11:    idle the system
12:  end while
13: end while
```

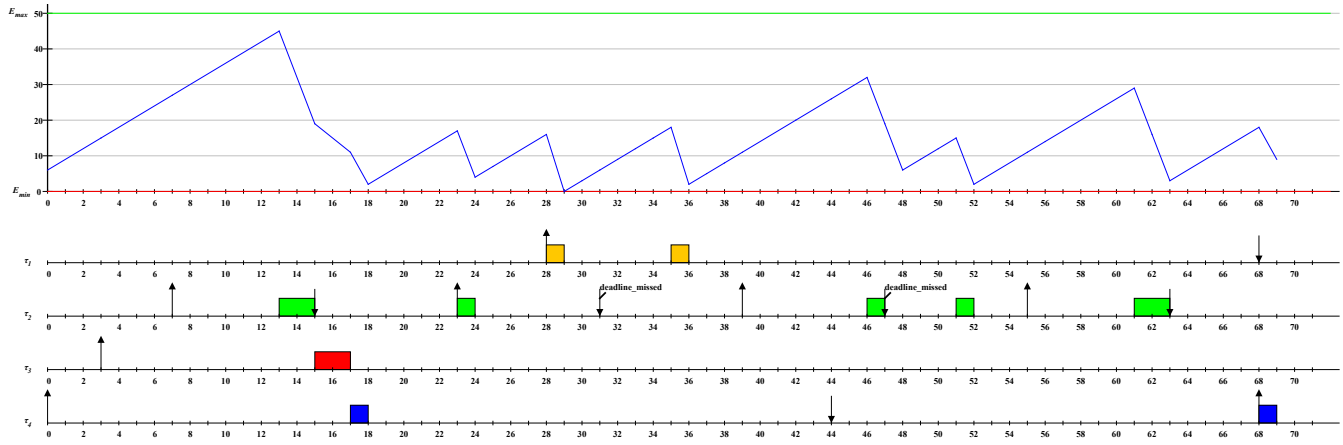


Figure 8: PFP_{ST} counter example

system.

PFP_{ST}**:** after PFP_{ASAP} and PFP_{ALAP} one can propose a hybrid algorithm that can behave sometimes as PFP_{ASAP} and sometimes as PFP_{ALAP}. The PFP_{ST} algorithm was built following this intuition. It executes jobs as soon as possible whenever the energy is sufficient to execute and replenishes otherwise. The replenishment periods are as long as the available slack-time (see Algorithm 3).

Even though this algorithm improves the schedulability rate comparing to PFP_{ALAP}, it is still not optimal because of the same reasons than PFP_{ALAP} and PFP_{ASAP}. Figure 8 shows a counter example.

FPC_{ASAP}**:** the computation of slack-time in PFP_{ALAP} and PFP_{ST} algorithms is considered as time clairvoyance or time lookahead because it uses the arrival times of future jobs. However, the energy constraints were not considered, this is why the precedent algorithms fail to schedule some feasible task sets.

Thus, one can add energy clairvoyance to compute the right retardations that lead to a valid schedule. Following this idea, the As Soon As Possible Clairvoyant Fixed-Priority Algorithm (FPC_{ASAP}) was proposed in [3]. Before authorizing a job to execute,

Algorithm 4 FPC_{ASAP} Algorithm

```
1: t ← 0
2: loop
3:    A ← set of active jobs at time t
4:    if A ≠ ∅ then
5:       J_{i,j} ← the highest priority job of A
6:       d_i(t) ← the next absolute deadline of J_{i,j}
7:       if ResponseTime_{PFP_ASAP}(t + 1, J_{i,j}, E(t + 1)) > d_i(t) then
8:          execute J_{i,j} for one time unit at time t
9:       else
10:         if Clairvoyance_{PFP_ASAP}(t, J_{i,j}, d_i(t), E(t)) then
11:            execute J_{i,j} for one time unit at time t
12:         else
13:            suspend the system for one time unit
14:         end if
15:      end if
16:   end if
17:   t ← t + 1
18: end loop
```

it simulates the PFP_{ASAP} schedule of the current and the future jobs in a clairvoyance window or a looakahead window.

The FPC_{ASAP} algorithm inherits the behavior of PFP_{ASAP} and adds clairvoyance capabilities. It schedules jobs as soon as possible whenever the two following conditions are met:

- there is enough energy available in the storage unit to execute at least one time unit,
- the execution of the current job does not lead to a deadline miss of jobs of higher priority which are requested during the clairvoyance window.

If these conditions are not satisfied, then, the algorithm suspends all executions for one time unit and then it tries again.

Algorithm 4 shows how FPC_{ASAP} takes scheduling decisions at time t when a job of priority level-i is ready to be executed. The FPC_{ASAP} algorithm checks first if the execution of jobs of priority level higher or equal than i meet theirs deadlines. Then, it checks if it is possible to delay the current job by comparing its response time at time $t+1$ with its deadline (line 7). After that, it repeats the process for higher priority jobs. This prevent delaying the current job uselessly because in the case where it is impossible to delay, if a deadline miss occurs in a higher priority level in the clairvoyance window, the deadline miss cannot be avoided and the system is not schedulable with FPC_{ASAP}. The length of the clairvoyance window for a job is not proved but can be intuitively defined as the interval of time starting from time t to the absolute deadline $d_i(t)$ because the current job of level-i cannot be delayed more than its deadline.

Unfortunately, FPC_{ASAP} is not optimal because when a future deadline is detected with the clairvoyance algorithm, all the jobs are delayed until the deadline miss disappears and it is too much in certain cases. This delay is from the left to the right following the time increasing axis. When the energy balance is negative at the end of the clairvoyance

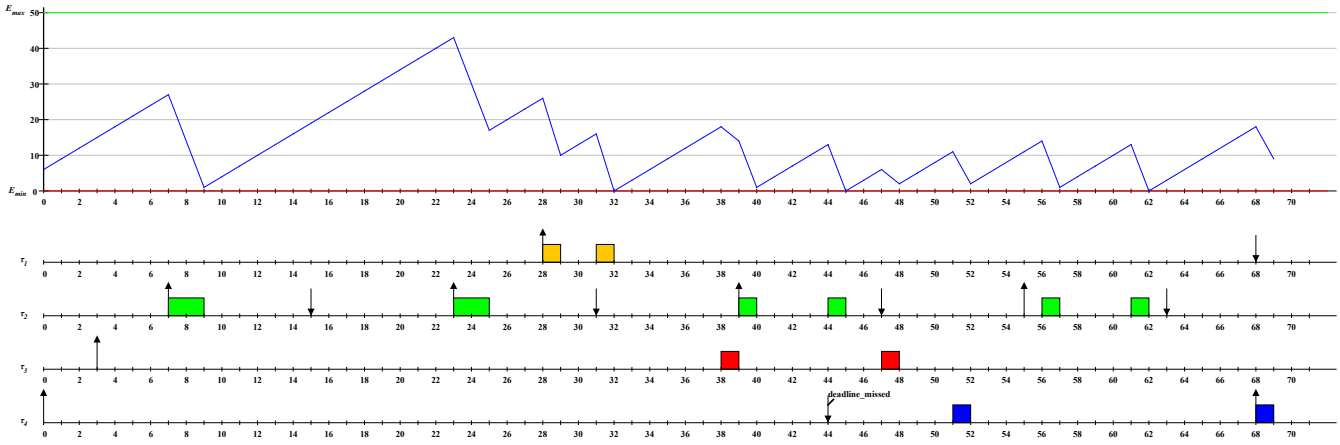


Figure 9: FPC_{ASAP} counter example

window, delaying a lower priority job for an unbounded period can lead to a deadline miss when a higher priority job is also delayed to a later time than the deadline of the lower priority job. Computing the response time at time $t+1$ of the ready job according to PFP_{ASAP} algorithm is not sufficient because it does not reflect the real response time of the job. Figure 9 illustrates a counter example where such a situation occurs. We can see that at time 0 when job $J_{4,1}$ is ready to execute, the PFP_{ASAP} lookahead schedule detects a deadline miss in a higher priority level in the looakahead window, i.e. the deadline of job $J_{2,1}$ (see Figure 6), then, it postpones execution for one time unit and then checks again at time 1, 2 and 3 and the same decision is taken. However, from time 3 to time 51, job $J_{4,1}$ cannot be executed because of the higher priority interferences and the required replenishments. This example teaches us that when the energy balance is negative, in this case in time interval $[0, 44]$, it is impossible to schedule all the jobs that are requested inside this interval. Thus, the only way is to delay some jobs out of this interval. Moreover, we can see that all the jobs cannot be pushed out of this interval, only the ones that are requested inside the interval and have deadlines outside the interval. We can see that FPC_{ASAP} takes the wrong decision by delaying the lower priority jobs, in this case $J_{4,1}$, instead of higher priority jobs.

FPLSA: one of the possible ways to find optimal fixed-priority algorithms is to try to adapt the behavior of some optimal algorithms for *EDF* scheduling. One can use the concept of the *LSA* presented in [7]. It consists of computing the latest time from which jobs can be executed continuously. This algorithm was proved to be optimal for task sets that consume energy with the same rate. To adapt *LSA* algorithm to fixed-priority scheduling, we assume that all tasks consume energy with the same rate, i.e. $\forall\ \tau_i,\ E_i/C_i = r$. Furthermore, we keep the same scheduling schemes and we use fixed-priority scheduling instead of *EDF* ones. Therefore, the algorithm becomes as described in Algorithm 5 and we call it *FPLSA*.

The latest job starting time denoted $s_{i,j}$ is computed by Equation 8.

$$\begin{cases} s_{i,j} = \max(s'_{i,j},\ s^*_{i,j}) \\ s^*_{i,j} = d_{i,j} - \dfrac{E(a_{i,j}) + g(a_{i,j},\ d_{i,j})}{r} \\ g(a_{i,j},\ s'_{i,j}) - \mathcal{C} = g(a_{i,j}, d_{i,j}) + (s'_{i,j} - d_{i,j}) \times r \end{cases} \tag{8}$$

Algorithm 5 $FPLSA$ Algorithm

$t \leftarrow 0$
while true **do**
  $J_{i,j}$ the ready job with the higher priority at time t
  calculate $s_{i,j}$
  if $t \geq s_{i,j}$ or $E(t) + P_r > E_{max}$ **then**
    execute job $J_{i,j}$
  else
    idle the system
  end if
  $t \leftarrow t + 1$
end while

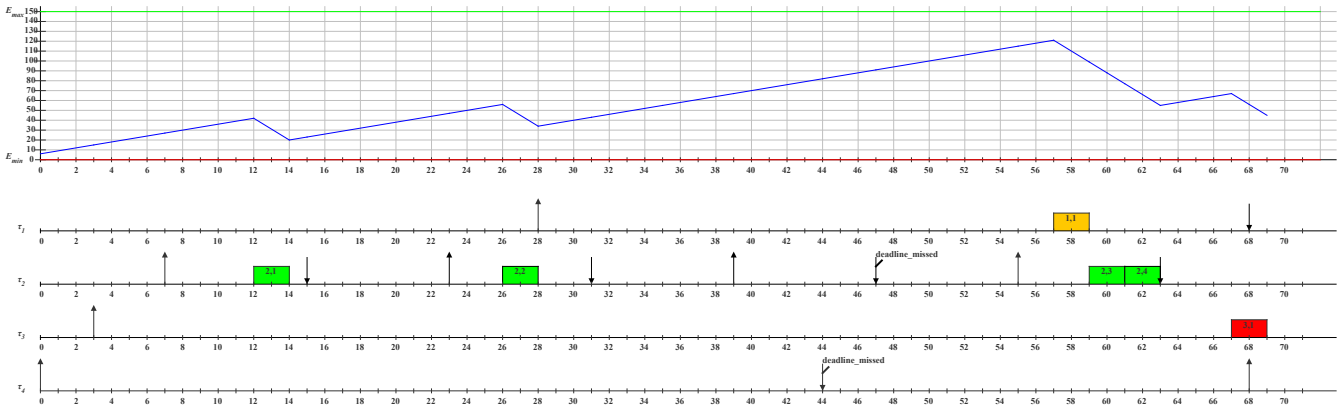


Figure 10: $FPLSA$ counter example

Unfortunately, this algorithm is not optimal even if we consider the same consumption rate for all tasks. We can see in Figure 10 that the computation of the starting time $s_{i,j}$ is not adapted to fixed-priority scheduling. In fact, $s_{i,j}$ is the latest time from which we can execute continuously until the deadline $d_{i,j}$. This works for EDF scheduling because in EDF all higher priority jobs have their request times and deadlines inside interval $[t, d_{i,j}]$. However, this is not the case in fixed-priority scheduling which can lead to too long delays. In Figure 10, we can see that the delay computed at time 3 for job $J_{3,1}$ is much longer than the delay computed for job $J_{4,4}$ at time 0 which led this later to miss its deadline.

FPLeg: the counter examples of the precedent algorithms show that even with energy and time clairvoyance the above algorithms are still not optimal. In fact, the priorities of tasks complicates the computation of the clairvoyance. The precedent examples show that in the case when the energy balance is negative in a certain interval of time, it is necessary to reduce the energy demand by pushing some jobs out of the interval. Then, the difficulty now is to find the subset of jobs to delay and calculate the lengths of their delays. We showed that delaying all the jobs at the maximum without considering energy like PFP_{ALAP}, delaying all the lower priority jobs when a future deadline miss is detected as FPC_{ASAP} and delaying jobs to satisfy only energy constraints like $FPLSA$ are not the right decisions to reduce safely the energy demand within a time interval. In fact, the potential jobs to be pushed out of the interval are the ones that are requested inside

Algorithm 6 $FPLeg$ 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) ≥ E_k/C_k and Slack.Time(t) ≤ 0 then
7:          execute τ_k for one time unit
8:       else
9:          replenish until time t + max(1, Slack.Time.with.virtual.deadlines(t))
10:      end if
11:   end if
12:   t ← t + 1
13: end loop
```

the interval and have their deadlines outside. These kind of jobs can be delayed by anticipating the execution of lower priority jobs. By doing so, the higher priority jobs to be pushed out need more replenishment time since the energy balance is negative. This leads to push them out of the interval and permits lower priority jobs to have more energy to execute which can help them to meet their deadlines.

The anticipation of lower priority jobs can be done by introducing a kind of virtual deadlines that coincide with request time of the jobs to be pushed out of the considered time interval. Then, once the virtual deadlines are set, we delay all the jobs at the maximum using the new virtual deadlines in the slack-time computation algorithm. Following this intuition we propose the $FPLeg$ algorithm for Fixed-Priority as Late as possible with energy guaranty which is inspired from the $EDeg$ algorithm presented in [5].

The idea behind this algorithm is to use the same scheduling schemes as PFP_{ALAP} but by using virtual deadlines that consider energy constraints.

Definition 6 (Virtual deadline)**.** *The virtual deadline of a job $J_{i,j}$ denoted vd_{ij} is the earliest time that makes its energy balance positive. This time can be the effective deadline of the job or the request time of one of the higher priority jobs described above. It must satisfy the following conditions:*

$$\begin{cases} vd_{i,j} \leq d_{i,j} \\ Bu(t, vd_{i,j}) \geq 0 \end{cases}$$

By analyzing the counter example of PFP_{ALAP} shown by Figure 7, we can see that if the energy balance was positive at time 31, the schedule would be valid and the slack-time time would be correctly calculated. Therefore, using virtual deadlines that makes the energy balance positive may be an interesting idea to build an optimal algorithm. Then, $FPLeg$ behaves as described in Algorithm 6.

Figure 11 illustrates the scheduling schemes of $FPLeg$ algorithm. We can see that the virtual deadline of job $J_{4,1}$ was shifted from time 44 to time 28 where the energy balance is positive, i.e. the battery level at time 28 is 0, and then, the as late as possible schedule produced in time interval $[0, 28]$ is valid. However, we notice that this does not solve completely the problem of late scheduling. In fact, using virtual deadlines improves schedulability but there exist some cases where it anticipates the execution of some lower priority jobs that consume the energy needed for other higher priority jobs. This is

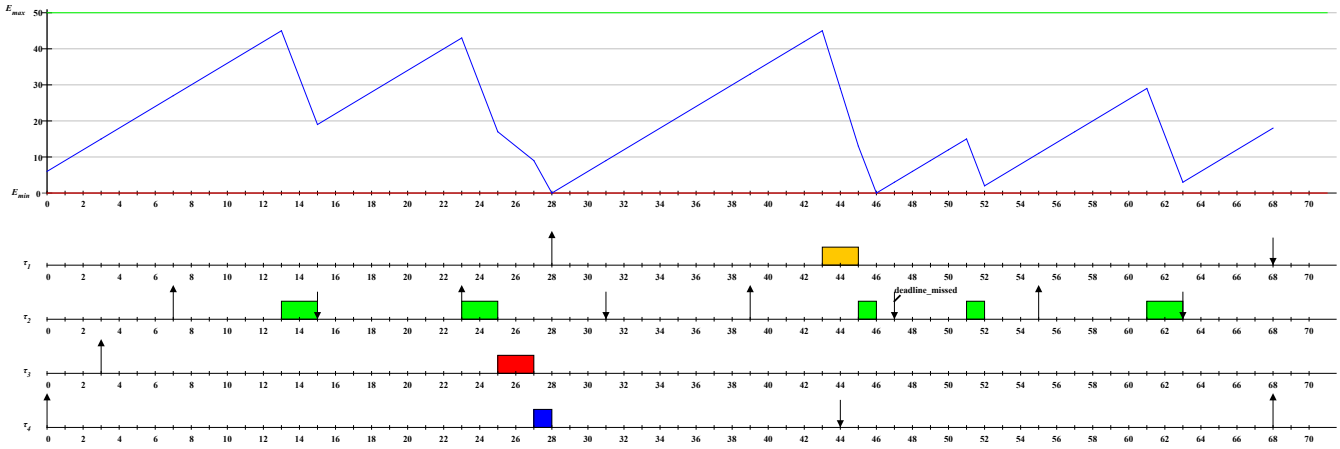


Figure 11: $FPLeg$ counter example

illustrated by Figure 11 at time 46. Even though the energy balance was positive at time 28, the missed deadline (time 47) is due to the anticipated execution of job $J_{3,1}$ which can be delayed further. Unfortunately, this proves that $FPLeg$ is not optimal and that the fixed-priority scheduling for energy-harvesting systems is subject to many counter intuitive ideas.

FP_{lh}: The use of virtual deadlines in $FPLeg$ was a good idea because it helps reducing the energy demand when the energy balance is negative. However, the above counter example shows that having a positive energy balance is not sufficient to check whether the current virtual deadline is the right one or not. Combining the idea of lookahead of FPC_{ASAP} with the idea of virtual deadlines can be an interesting intuition to achieve an optimal algorithm.

In the following we propose a new algorithm called FP_{lh} for *Fixed-Priority lookahead* that combines the ideas of virtual deadlines and lookahead. The virtual deadlines make the energy balances positive and the lookahead computation checks if all future deadlines are met during a bounded window. Therefore, the definition of the virtual deadline should take into account the future higher priority jobs. Moreover, the looakahead computation consists not only of calculating the energy balance of the current job $J_{i,j}$ but also the one of higher priority jobs that are included in the lookahead window. For this reason, we need to generalize the energy balance formula to compute the energy balance of any job at any moment.

Definition 7 (Energy Balance)**.** *The energy balance of priority level-i at time interval* $[t_1, t_2[$ *denoted* $Ba_i(t_1, t_2)$ *is the difference between the energy budget and the energy demand during* $[t_1, t_2[$*. It can be computed with Equation 9.*

$$\begin{cases} A & = & \sum_{j>i} (DBF_j(0, a_i(t_1)) \times E_j \\ B & = & (RBF_j(0, a_i(t_1)) - DBF_j(0, -a_i(t_1))) \times (E_j - e_j(a_i(t_1))) \\ Ba_i(t_1, t_2) & = & Bu(t_1, t_2) - We_i(t_1, t_2) \\ & = & Bu(0, t_2) - We_i(0, t_2) - A + B \end{cases} \quad (9)$$

where :

- *A: is the energy demand of lower priority jobs within time interval* $[0, a_i(t_1)[$*, i.e. finished jobs (see Figure 3),*
- *B: is the energy demand of lower priority jobs that are probably not yet finished at time* $a_i(t_1)$*. To compute this value we need to simulate the execution with* PFP_{ALAP} *algorithm.*

Now, we redefine the virtual deadline to include the lookahead computation.

Definition 8 (Virtual Deadline)**.** *The virtual deadline* $vd_{i,j}$ *of a job* $J_{i,j}$ *is an early deadline that makes the energy balance* $Ba(t, vd_{i,j})$ *positive and does not cause negative energy balances for jobs of higher priority tasks withing the lookahead window that ends at time* L*. It must respect the following conditions:*

- $vd_{i,j} \leq d_{i,j}$
- $Ba_i(t, vd_{i,j}) \geq 0$
- $\forall\ J_{k,l} \in C,\ Ba_k(a_{k,l}, d_{k,l}) \geq 0$ *where* $C = \{J_{k,l},\ k < i$ *and* $a_{k,l} > a_{i,j}$ *and* $d_{k,l} \leq L\}$

The FP_{lh} algorithm is an extension of $FPLeg$ algorithm described above. It postpones executions whenever there is available slack-time like PFP_{ALAP} but the slack-time computation is done using the virtual deadlines. Note that the virtual deadline $vd_{i,j}$ coincides with a request time of a higher priority job that is activated before the deadline of $J_{i,j}$ and has an absolute deadline later than the one of $J_{i,j}$.

For a job $J_{i,j}$ at time t, we know that the higher priority jobs that have an absolute deadline earlier than $d_{i,j}$ cannot be delayed outside the time interval $[t, d_i(t)]$. Thus, the only jobs that can be pushed totally or partially out of this interval are those that have a deadline later than $d_{i,j}$. The virtual deadline of $J_{i,j}$ can be the request time of one of these jobs. To find the right virtual deadline, we test all of them starting with the earliest one. This test consists of checking two conditions:

- whether the energy balance of the considered job at this time is positive or not,
- whether there is future deadline misses within the lookahead window.

The length of the lookahead window is one of the problems of lookahead scheduling, it is at least bounded but should be specified carefully. For this algorithm, we consider that the lookahead window begins at time t and ends at time L which is the latest deadline of higher priority jobs that are requested before time $d_{i,j}$ and have their deadlines after $d_{i,j}$. We choose this length because the jobs to delay cannot be delayed longer than the length of the lookahead window.

The lookahead function consists of checking the energy balance of all the jobs that are requested within the lookahead window as described in Algorithm 8.

These rules allow us to be sure that the selected virtual deadlines prevent negative energy balances and future deadline misses, and lead to a correct energy-aware slack-time computation that can give the correct retardations. Figure 5 shows the correct schedule of the task set that was not feasible with all the precedent algorithms. We can see that the virtual deadline of job $J_{4,1}$ was shifted to time 3 which makes the energy balance in time interval $[0, 3]$ positive and ensure that all higher priority jobs included in the lookahead window $[0, 70]$ meet their deadlines in contrast to $FPLeg$.

Algorithm 7 FP_{lh} Algorithm

$t \leftarrow 0$
loop
  $A \leftarrow$ set of active tasks at time t
  if $A \neq \emptyset$ **then**
    $\tau_k \leftarrow$ the highest priority task of A
    if $E(t) - E_{min} \geq E_k/C_k$ and $Slack.Time.with.virtual.deadlines(t) \leq 0$ **then**
      execute τ_k for one time unit
      $t \leftarrow t + 1$
    else
      replenish until slack-time time unit
      $t \leftarrow t + \max(1, Slack.Time.with.virtual.deadlines(t))$
    end if
  end if
end loop

Algorithm 8 $Lookahead(Ba_i(t), t, \tau_i, L)$ Algorithm

$C = \{J_{k,l},\ k < i\ and\ a_{k,l} > t\ and\ d_{k,l} \leq L\}$
for $J_{k,l} \in C$ **do**
  if $Ba_k(t, d_{k,l}) < 0$ **then**
    return $False$
  end if
end for
return $True$

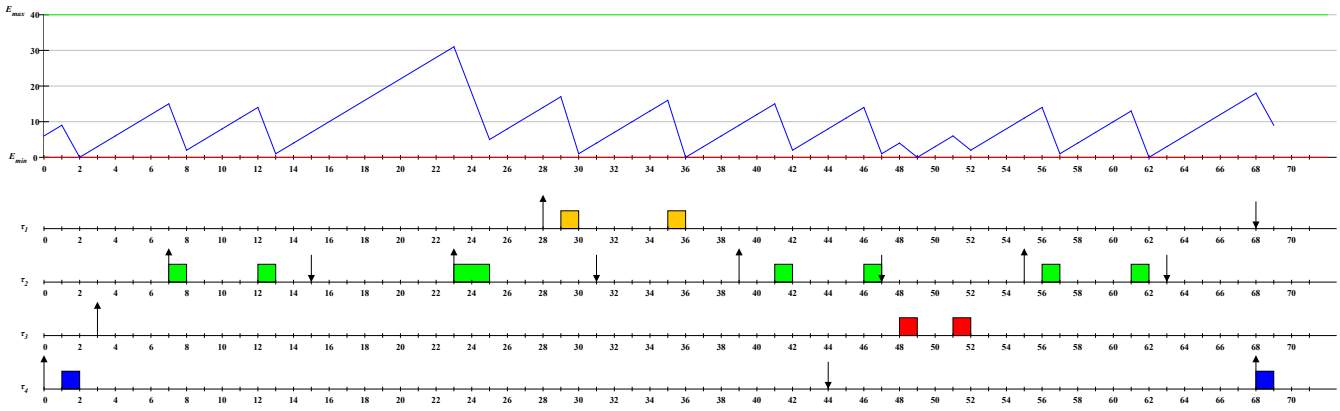


Figure 12: A FP_{lh} schedule

After all the precedent intuitions and counter intuitive ideas, we think that this algorithm is optimal or at least dominates than the other algorithms. However, its main issue is its complexity. In fact, the length of the lookahead window and the relationship between jobs deadlines are the main factors that increase the complexity of FP_{lh}. Therefore, the following questions arise:

1. Does the use of virtual deadlines change the set of jobs to check with lookahead computations ?

   - Using earlier deadlines can decrease the number of jobs to check because the

interval to study is shortened. For example, when a valid virtual deadline is found for the higher priority job with latest deadline, the lookahead window becomes shorter.

- Using real deadlines may lead to a negative energy balance for a higher priority job, as shown in the counter example of PFP_{ALAP}. Furthermore, computing virtual deadlines by only checking the energy balance of the current job can lead to a wrong virtual deadline as shown in the counter example of $FPLeg$.
- Then, the virtual deadline computation for a job of priority level-i needs to compute the energy balance $Ba_i(t, d_i(t))$ and the energy balances of higher priority jobs within the lookahead window as shown in Definition 8. Moreover, since a part of the energy demand of lower priority jobs that have already began their execution before the request time of the job whose virtual deadline is being computed is necessary as shown with vertical lines pattern in Figure 3. Their virtual deadlines are also needed to compute the energy balance of a higher priority job. This means that to compute the virtual deadline of one job, we need both virtual deadlines of higher and lower priority jobs which leads to a kind of cross dependency between the virtual deadlines of different priority levels. Figure 13 illustrates such a situation. We can see that to compute the virtual deadline of job $J_{4,1}$ at time 8, we need the one of $J_{2,1}$ because it is inside the lookahead window. Moreover, to compute the virtual deadline of job $J_{2,1}$ at time 20, we need the real schedule of jobs $J_{4,1}$ and $J_{5,1}$ which depend on their virtual deadlines. Therefore, there is an interdependence between virtual deadlines computation which makes calculating them one by one impossible. A combination of virtual deadlines is the set of virtual deadlines of the jobs that are included in the lookahead window. Since each job may have several potential virtual deadlines, finding the right one means finding the right virtual deadlines of all the other jobs. Then, to find such correct virtual deadlines, we have to pick one combination and test if it works. We do this until we find a correct combination. In the worst-case we have M^N combinations to test which is an exponential complexity, where M is the maximum number of potential virtual deadlines of a job and N is the maximum number of jobs that we can have inside a lookahead window.

2. Are the virtual deadlines necessary to compute the energy balance ?

   - The energy demand of higher priority levels is calculated only with requests times, i.e. with request bound function, thus, virtual deadlines of higher priority levels cannot change the final energy demand.
   - The energy demand of lower priority jobs that have already finished their execution is calculated only with deadlines, i.e. with demand bound function denoted A in Equation 6. Then, the use of virtual deadlines of higher priority levels cannot change the final energy demand because they are earlier than the real deadlines.
   - The energy demand of lower priority jobs that have not yet finished, denoted B in Equation 9 depends on the executions done before time $a_i(t_1)$ with PFP_{ALAP} policy which is based on deadlines. Using virtual deadlines for these jobs can change the final energy demand.

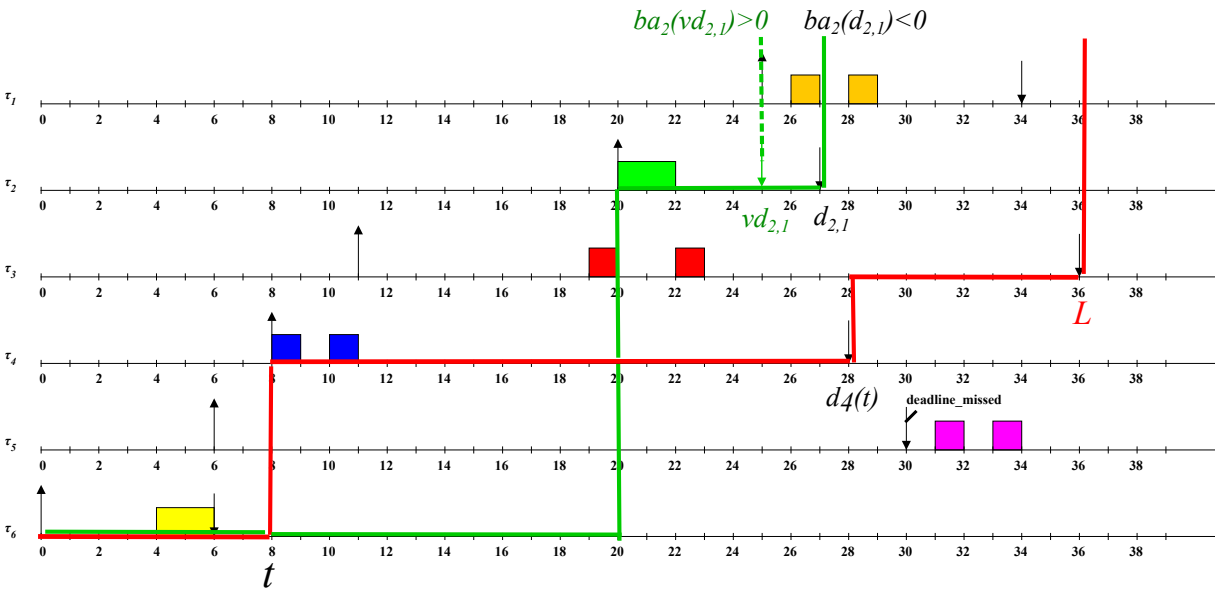


Figure 13: Interdependency between virtual deadlines

Therefore, virtual deadlines of lower priority jobs are needed for the energy balance computation.

3. Assuming that computing virtual deadlines is an NP-hard problem, are they necessary for an optimal algorithm ?

   - If yes, the scheduling problem for fixed-priority energy-harvesting systems is also NP-Hard also,
   - Otherwise, it is an open problem.

These insights confirm that the fixed-priority scheduling for energy-harvesting systems is not easy to handle. Moreover, the only algorithm we have up to now that could be optimal has an exponential complexity. Therefore, finding an optimal algorithm with a reasonable complexity or proving that the fixed-priority problem scheduling for energy-harvesting systems is an NP-Hard problem is still an open problem.

5 Conclusion

In this work we explored the possibility of the existence of an optimal scheduling algorithm for energy-harvesting systems. We started by proving that such an algorithm must be both non-energy-work-conserving and lookahead (or clairvoyant). Then, we listed some ideas of scheduling algorithms that seem intuitively optimal and we showed with counter examples why they are not. We showed that the computation of the needed delays, i.e. replenishment periods, must consider jobs deadlines and energy cost as well as the replenished energy. Moreover, we showed also that deciding which job to delay is the main problem of building an optimal algorithm. We know now that considering a late or a lazy scheduling and having a positive energy balance at the deadline of a job is not sufficient. The maximum delay for a job must ensure not only a positive energy balance but also the respect of all deadlines in a bounded lookahead window. This idea is very interesting,

however, the exact lookahead computation seems to be complicated and have an exponential complexity. This complexity seems to be difficult to reduce because the computation of the maximum delay of one job, i.e. the computation of the virtual deadline, depends on the maximum delay of the other jobs inside the lookahead window. There is a kind of cross dependency between jobs which complicates the computations. The only investigated solution here is to perform a brute force search for the right combination of jobs delays or virtual deadlines which has an exponential complexity. The conclusion of this work is that the correct late scheduling that respects the energy constraints needs earlier deadlines and that the computation of these deadlines has an exponential complexity. Moreover, if an optimal algorithm requires these deadlines, then, the scheduling problem of fixed-priority energy-harvesting systems is a hard problem. Otherwise, we should find an other way to compute the right delays, this is still an open problem.

References

[1] Y. Abdeddaïm, Y. Chandarli, R. Davis, and D. Masson. Schedulability analysis for fixed priority real-time systems with energy-harvesting. In *Proceedings of the International Conference on Real-Time Networks and Systems (RTNS)*, pages 67–76, 2014.

[2] Y. Abdeddaïm, Y. Chandarli, and D. Masson. The Optimality of PFP_{ASAP} Algorithm for Fixed-Priority Energy-Harvesting Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[3] Y. Abdeddaïm, Y. Chandarli, and D. Masson. Toward an optimal fixed-priority algorithm for energy-harvesting real-time systems. In *Proceedings of the Work in progress session of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–48, 2013.

[4] Y. Chandarli, Y. Abdeddaïm, and D. Masson. The Fixed Priority Scheduling Problem for Energy Harvesting Real-Time Systems. In *Proceedings of the work in progress session of the the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[5] M. Chetto. Optimal scheduling for real-time jobs in energy harvesting computing systems. *IEEE Transactions on Emerging Topics in Computing*, 2014.

[6] R. Jayaseelan and T. Mitra. Estimating the Worst-Case Energy Consumption of Embedded Software. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[7] C. Moser, L. Thiele, L. Benini, and D. Brunelli. Real-time scheduling with regenerative energy. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, ECRTS '06, pages 261–270. IEEE Computer Society, 2006.