Energy Management for IoT

Lab 1: Dynamic Power Management

Alfredo Paolino: s295152@studenti.polito.it
Davide Fogliato: s303497@studenti.polito.it

December 4, 2022

Contents

1 Introduction

In this technical report, lab 1 of the course "Energy Management for IoT" will be described.
The goal of this first lab is to understand the basics of Dynamic Power Management (DPM) using a simple Power State Machine (PSM) simulator in C. More specifically, we will evaluate in a case study how energy saving changes as a function of:

- The applied DPM policies (highlighting the pros and cons of each of them)
- The distribution of inactivity times
- The PSM parameters

The report is divided into 3 sections, each of them addressing problems of increasing complexity. Each section is in turn composed of a first part where we briefly present the approach chosen to face the problem and fulfill the requirements, and a second part where we present the obtained results by means of tables and graphs.
A brief theoretical introduction has been inserted prior to the core of the report, in order to have a common groundwork for all three sections, thus avoiding spreading information here and there between the *Workflow* and the *Results* parts.

2 Background

The general idea of DPM is to reduce power consumption by turning resources into a low-power state when under-utilized; in practice, a state in which power is lower than in normal operating conditions. To achieve this improvement, different possible techniques can be used, such as:

- Clock gating
- Power/ground gating
- Supply voltage scaling
- Threshold voltage scaling

A resource that has more than one single internal state is called a power manageable component. The several internal states of a resource are summarized in the PSM, an abstracted model composed of many nodes as many power states, with edges that represent the transitions between them.
The PSM example in figure 1 shows how each power state is characterized by a certain amount of power consumption, while the time taken and the power consumed for the transitions are annotated on the edges.
Given a PSM and a workload, the goal is to determine the optimal allocation of power states over time that minimizes power under performance constraints.

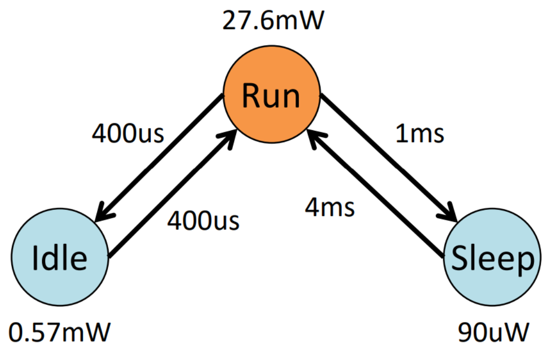


Figure 1: PSM example

DPM lends itself to a clean conceptual systemic view and abstract model, called **pm!** (**pm!**). A model of the Power Manager is represented in figure 2; the scheme shows how the goal of the PM is to monitor the service requestor's activity and set the state of the provider, the resource for which the power consumption should be reduced, according to a specific policy.

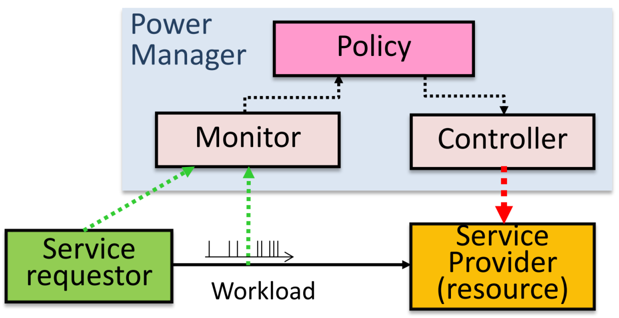


Figure 2: PM model

The main function of a power management policy is to decide when to perform component state transitions and which transition should be performed. This type of decision has to be taken based on a certain policy, that decides if an inactive interval is long enough to make the transition convenient.
Therefore, it is important to define the threshold used as a reference point to enter the low-power state: the break-even time T_{BE}.
DPM policies are divided into two categories:

- Static
  - Based on time-out
  - Predictive
- Dynamic
  - Adaptive
  - Stochastic

Speaking about Timeout-policy, its implementation is based on observing the initial part of the inactive time (T_{TO}) to predict the length of the remaining part.

$$Prob(T_{inactive} > T_{TO} + T_{BE} | T_{inactive} > T_{TO}) \approx 1$$

This solution is easy to implement but the basic assumption could not be always true and it always pays a penalty for the T_{TO} waiting and for the wakeup as shown in figure 3.

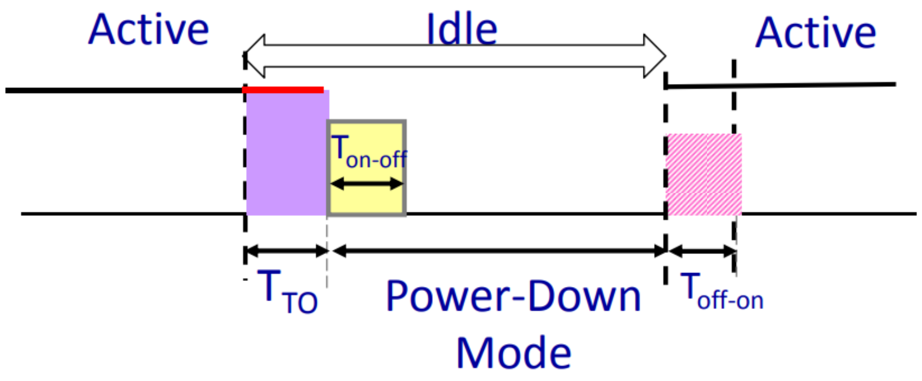


Figure 3: Timeout policy

On the other hand, predictive policies address the issue of the power wasted while waiting for the timeout to expire, using different prediction approaches to see whether $T_{inactive}$ is long enough to justify shutdown.

3 Part 1 : Default Timeout Policy

In this first section, we have to compile the DPM simulator and run it with two different workloads, using the default timeout policy. The aim is to understand how the DPM adapts to different workloads. On top of that, we will knob the timeout value for both workloads to see how the DPM reacts to it.
The workloads we are going to use are typical IoT ones, divided in three main phases (as shown in figure 4):

- Sensing
- Computation
- Waiting

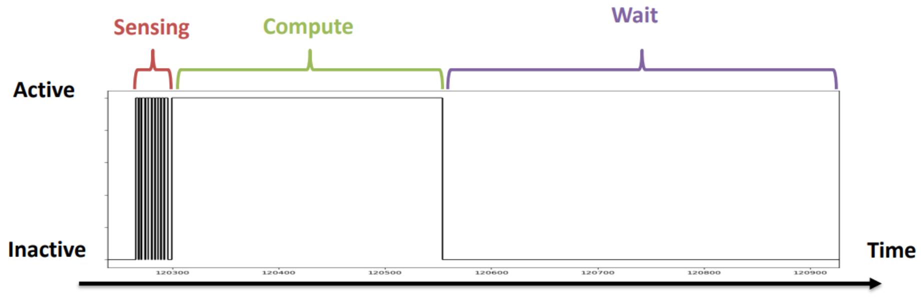


Figure 4: Typical IoT workload

The first workload uses fast sensors, which alternate very fast between active and inactive time (no more than 4 ms) during the sensing phase, while the second one uses slow sensors, which have longer inactive times between a sensing and the next one (up to 100 ms). We will analyze how this difference impacts the DPM in the *Results* section.

3.1 Workflow

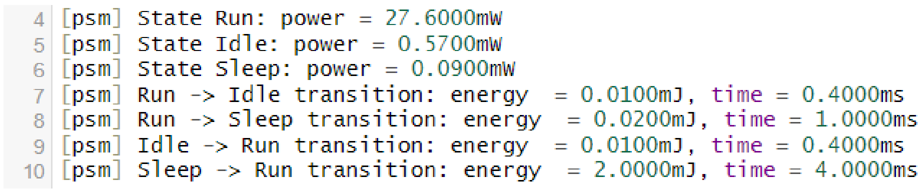
Before starting with this first part, some preliminary work has to be done. We are working on VSCode, so the first step is to download and install MinGW in order to have access to the *GCC compiler* and *Make*, required to automate the mundane aspects of building executable from source code.
With this done we can compile the file *dpm_simulator.c* and launch the executable generated with *Make* through the command:

```
$ ./dpm_simulator -t 20 -psm example/psm.txt -wl example/wl.txt
```

By changing the value of the -t parameter we can set a new value for the timeout, thus analyzing the DPM behavior in different situations.

3.2 Results

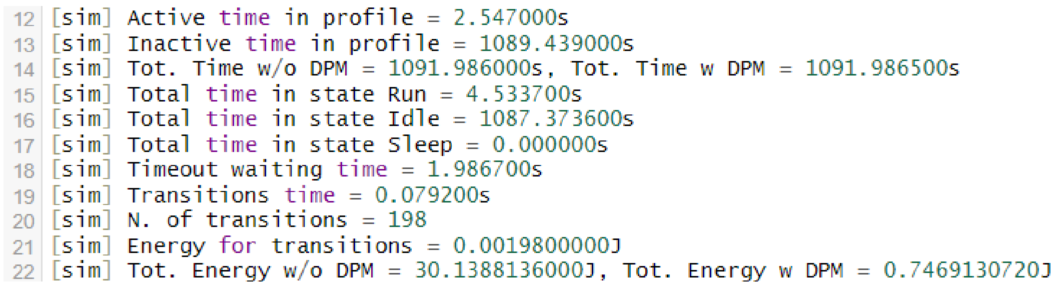
We report here the results obtained from different simulations with different timeout values for both workloads, and then we will comment the results.
Before starting the simulation, the program shows some information about the PSM, reported in figure 5.



```
[psm] State Run: power = 27.6000mW
[psm] State Idle: power = 0.5700mW
[psm] State Sleep: power = 0.0900mW
[psm] Run -> Idle transition: energy  = 0.0100mJ, time = 0.4000ms
[psm] Run -> Sleep transition: energy  = 0.0200mJ, time = 1.0000ms
[psm] Idle -> Run transition: energy  = 0.0100mJ, time = 0.4000ms
[psm] Sleep -> Run transition: energy  = 2.0000mJ, time = 4.0000ms
```

Figure 5: PSM state power and transition energy/time information

Right after the PSM information section, the program shows the simulation results, as shown in figure 6.



```
[sim] Active time in profile = 2.547000s
[sim] Inactive time in profile = 1089.439000s
[sim] Tot. Time w/o DPM = 1091.986000s, Tot. Time w DPM = 1091.986500s
[sim] Total time in state Run = 4.533700s
[sim] Total time in state Idle = 1087.373600s
[sim] Total time in state Sleep = 0.000000s
[sim] Timeout waiting time = 1.986700s
[sim] Transitions time = 0.079200s
[sim] N. of transitions = 198
[sim] Energy for transitions = 0.0019800000J
[sim] Tot. Energy w/o DPM = 30.1388136000J, Tot. Energy w DPM = 0.7469130720J
```

Figure 6: Simulation results for $T_{TO} = 20\text{ms}$

Every line of this section is fundamental to understanding why DPM is a game changer in terms of power savings:

- The difference between the *Total time in state Run* and the *Active time in profile* makes us understand how far we are from an ideal case of a DPM implemented with an oracle.
  More specifically, we can observe that:

  $$Total\ time\ in\ state\ Run \ = \ Active\ time\ in\ profile \ + \ Timeout\ waiting\ time$$

  This means that the *Timeout waiting time* is the non-ideality introduced by the timeout policy.

- A similar argument can be done for *Total time in state Idle*, *Inactive time in profile* and *Timeout waiting time*.
  In fact, similarly to the previous case:

  $$Total\ time\ in\ state\ Idle \ = \ Inactive\ time\ in\ profile \ - \ Timeout\ waiting\ time$$

- For what concerns the total *Timeout waiting time*, it can be computed as:

  $$Timeout\ waiting\ time \ = \ Timeout\ parameter \ * \ N.\ of\ transitions\ /\ 2$$

  In the case of figure 6 simulation, a timeout parameter of 20ms was set.

- The last row is the most important. From it, we can understand the real power of a DPM approach. In fact, we can compute the total energy saved as:

  $$Energy\ saved \ = \ Tot.\ Energy\ w/o\ DPM \ - \ Tot.\ Energy\ w\ DPM$$

Let's have a look now at the results obtained for the two available workloads:

Workload 1								
T_{TO} [ms]	0	1	2	3	4	100	10000	150000
Total Energy [J]	0.6977	0.6997	0.7015	0.7026	0.7032	0.7265	3.1349	29.879
$\sum T_{TO,i}$ [ms]	0.0000	0.0864	0.1545	0.0328	0.2561	1.1201	90.220	1079.6
N. of transitions	170	138	128	82	18	18	18	0

Workload 2								
T_{TO} [ms]	0	1	10	90	100	110	1000	120100
Total Energy [J]	0.6932	0.6961	0.7202	0.9313	0.9465	0.9522	1.1929	30.139
$\sum T_{TO,i}$ [ms]	0.0000	0.1058	0.9968	8.8222	9.4217	9.6428	18.553	1089.4
N. of transitions	198	198	198	160	56	28	18	0

By looking at the tables we can observe many important aspects of a DPM approach.
First of all, it is possible to notice the trend: the more we decrease the timeout value, the more power consumption decreases. This is not always true, but it depends on the power consumed during a transition (which could be even higher than the one consumed by simply remaining Run state) and on the time it takes for the transition to be completed. In this case, both these values are very small, so many short transitions would not introduce any overhead.
Let's put some numbers down to understand why, and have a benchmark for the next parts:

$$State\ Run:\ power\ =\ 27.6000mW$$

$$Run\ \rightarrow\ Idle\ transition:\ energy\ =\ 0.0100mJ,\ time\ =\ 0.4000ms$$

$$Idle\ \rightarrow\ Run\ transition:\ energy\ =\ 0.0100mJ,\ time\ =\ 0.4000ms$$

$$\Rightarrow Run\ \rightarrow\ Idle\ transition:\ power\ =\ \frac{energy}{time} = 25.0000mW\ <\ 27.6000mW\ =\ Run\ power$$

$$\Rightarrow Idle\ \rightarrow\ Run\ transition:\ power\ =\ \frac{energy}{time} = 25.0000mW\ <\ 27.6000mW\ =\ Run\ power$$

This relationship denotes that in terms of power consumption it is always beneficial to perform a Run → Idle → transition, even if the inactive window is small even null in the worst case).
Please note that we are not taking into account the possibility of pre-wakeup. This means that Idle → Run transitions occur while we should be already in Run state. This would for sure end up in a performance loss and possible delays in a real scenario, but here we just want to focus on power consumption, so let's ignore this problem for the moment.
Of course, with a low T_{TO}, the number of transitions will be higher because every time the simulator detects a long enough inactive time in the profile it will perform a transition to the Idle state. We reach the peek with $T_{TO} = 0$ indeed. In our case this setup is beneficial, because it emulates the behavior of an oracle, consuming the least possible amount of power, but in many other cases, it could be detrimental. As an example, if the PSM parameters provide for high transition energy or long transition time, we could end up wasting power instead of saving it when short transitions account for most of the total (as we will see later).
The trend between the two workloads is very similar, but the intrinsic difference between them causes a shift of the T_{TO} threshold for which the number of transitions suddenly drops:

- For workload 1 the threshold is around $T_{TO} \sim 4ms$.
- For workload 2 the threshold is around $T_{TO} \sim 100ms$.

These results are in line with the sensors simulated by workloads 1 and 2 (fast sensing for workload 1 and slow sensing for workload 2).
A high value of T_{TO}, higher than the longest inactive time, completely nullifies every benefit brought by a timeout policy, because no transitions to the Idle state are performed. This happens at $T_{TO} \sim 125000ms$ for workload 1 and $T_{TO} \sim 120100ms$ for workload 2. If this is the case, $E_{DPM} = E_{noDPM}$.

4 Part 2: Extension of the timeout policy

The goal of this second section is to modify the basic timeout-based policy used for Part 1 by enabling the possibility to move from Run state to Sleep state.
This new scenario is different from the previous one for two main reasons:

- The transition Run → Sleep → Run lasts much longer and consumes a high amount of power
- The Sleep state consumes a very low amount of power

For these two reasons we can assume that the Run → Sleep transition will be convenient only for those inactive times long enough to compensate for the power loss caused by the transition.

4.1 Workflow

The first thing to do is to modify the policy by simply changing the value of *next_state* variable in the file *dpm_policies.c* from *PSM_STATE_IDLE* to *PSM_STATE_SLEEP* in case of an inactive window:

```
case DPM_TIMEOUT:
    /* Day 2: EDIT */
    if(t_curr >= t_inactive_start + tparams.timeout) {
        //*next_state = PSM_STATE_IDLE;
        *next_state = PSM_STATE_SLEEP;
    } else {
        *next_state = PSM_STATE_RUN;
    }
    break;
```

With this done, we can run again the simulation and compare the results with the previous case.

4.2 Results

First of all, we report some results in tables similar to those used before to highlight some differences:

Workload 1								
T_{TO} [ms]	0	1	2	3	4	100	10000	150000
Total Energy [J]	0.2606	0.2559	0.2489	0.2304	0.2022	0.2259	2.6771	29.879
$\sum T_{TO,i}$ [ms]	0.0000	0.0473	0.0889	0.1480	0.2237	1.0877	90.1877	1079.6
N. of transitions	82	76	68	48	18	18	18	0

Workload 2								
T_{TO} [ms]	0	1	10	90	100	110	1000	120100
Total Energy [J]	0.3682	0.3711	0.3957	0.5546	0.4822	0.4588	0.6960	30.139
$\sum T_{TO,i}$ [ms]	0.0000	0.1058	0.9968	8.7557	9.3532	9.6405	18.520	1089.4
N. of transitions	198	198	198	144	56	26	18	0

Using these random values for T_{TO} we noticed that the trend was no more uniform. It is in fact easy to notice that for workload 2 we have two minimums this time; one global at $T_{TO} = 0ms$ and one local at around $T_{TO} = 110ms$. Thus, we conducted a more systematic analysis campaign through a couple of bash/Matlab scripts to find an explanation for this behavior.

Bash script

```bash
#!/Bin/bash
true > "myfile.txt"
while getopts s:e:f: flag
do
    case "${flag}" in
        f) timevar=${OPTARG};; #first timeout
        s) timeout_step=${OPTARG};;
        e) timeout_end=${OPTARG};;
    esac
done
while [ $timevar -le $timeout_end ]
do
    energy=$(./dpm_simulator -t $timevar -psm example/psm.txt -wl example/workload_2.txt
    ↪  | awk '/Energy[[:space:]]w[[:space:]]DPM/ {print $13}')
    echo "${energy:0:11} $timevar" >> "myfile.txt"
    timevar=$((timevar+timeout_step))
done
```

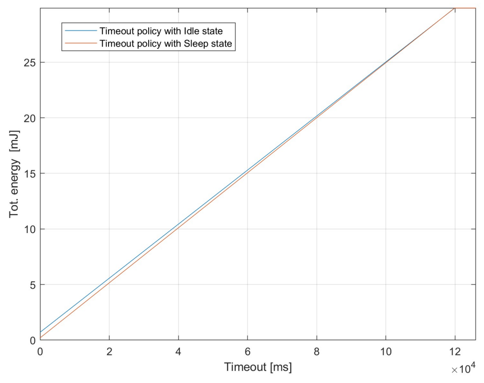
The purpose of this script is to generate an output file with couples indicating the energy consumed using a certain T_{TO} value. The extraction of this information is performed by an *awk* command on the results obtained by launching *dpm_simulator* each time with a different T_{TO}. Results are then written into a file to process data with Matlab.

Matlab script

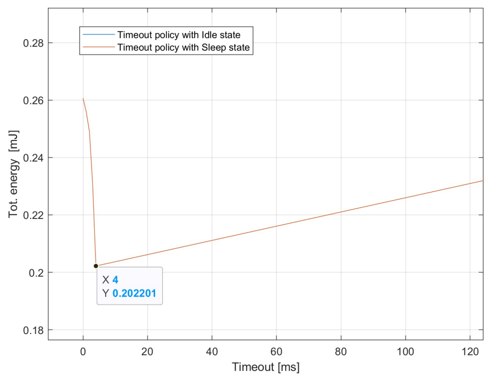
```
A=load('idle_file2.txt');
B=load('sleep_file2.txt');
plot(A(:,2),A(:,1),B(:,2),B(:,1))
legend('Timeout policy with Idle state','Timeout policy with Sleep state')
xlabel('Timeout [ms]')
ylabel('Tot. energy  [mJ]')
%axis([0 127000 0 31])  %entire graph
axis([0 130 0 1])  %to find the minimum
grid on

min_loc_idle=zeros(length(A(:,1)),2);
min_loc_sleep=zeros(length(B(:,1)),2);
m=1;
for n = 1:length(A(:,2))
   if(n==1)
       min_abs_idle=A(n,:);
   else
       if(min_abs_idle(1,1)>A(n,1))
           min_abs_idle=A(n,:);
       end
       if(n>2) && (n<length(A(:,2)))
           if(A(n-1,1)<A(n-2,1)) && (A(n,1)>A(n-1,1))
            min_loc_idle(m,:)=A(n-1,:);
            m=m+1;
           end
       end
   end
end
k=1;
for n = 1:length(B(:,2))
   if(n==1)
       min_abs_sleep=B(n,:);
   else
       if(min_abs_sleep(1,1)>B(n,1))
           min_abs_sleep=B(n,:);
       end
       if(n>2) && (n<length(B(:,2)))
           if(B(n-1,1)<B(n-2,1)) && (B(n,1)>B(n-1,1))
            min_loc_sleep(k,:)=B(n-1,:);
            k=k+1;
           end
       end
   end
end
min_abs_idle
for n = 1:m-1
    if(min_loc_idle(n,1)>0)
        min_loc_idle(n,:)
    end
end
min_abs_sleep
for n = 1:k-1
    if(min_loc_sleep(n,1)>0)
        min_loc_sleep(n,:)
    end
end
```

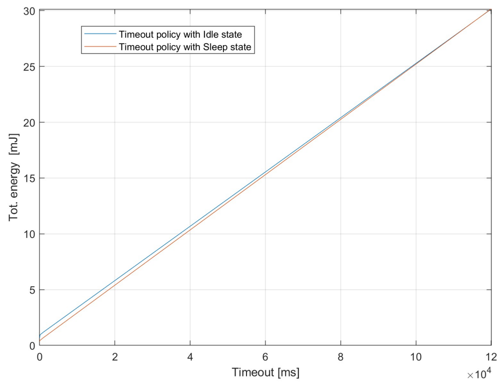
The purpose of this script is to plot the results obtained from the bash script and to look for the possible minimums of the function.
To have a complete view of the Sleep state benefits we plotted also the corresponding results using only Run → Idle transitions. It is possible to notice how the Sleep state curve is always below the Idle state curve.
We decided to write the minimums research part of the script in a more generic way, which takes into account multiple local minimums, in order to have a reusable script. This procedure shows the presence of multiple local minimums for workload 2 in the range ≈ (100ms - 120ms)



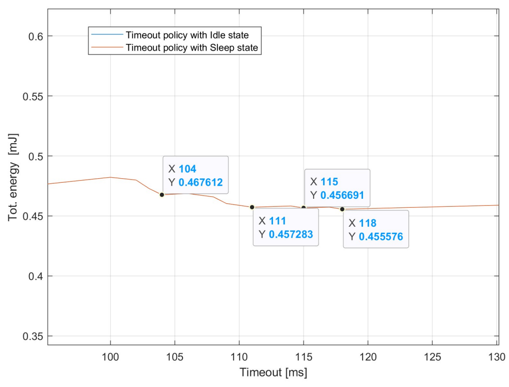
(a) Matlab plot for workload 1



(b) Detail of Matlab plot for workload 1



(c) Matlab plot for workload 2



(d) Detail of Matlab plot for workload 2

In face of the evidence, we tried to give an explanation of the difference between workloads 1 and 2 and between using the Idle state and Sleep state.
Let's reason again in terms of numbers:

$$State\ Run:\ power\ =\ 27.6000mW,\ State\ Sleep:\ power\ =\ 0.0900mW$$

$$Run\ \rightarrow\ Sleep\ transition:\ energy\ =\ 0.0200mJ,\ time\ =\ 1.0000ms$$

$$Sleep\ \rightarrow\ Run\ transition:\ energy\ =\ 2.0000mJ,\ time\ =\ 4.0000ms$$

$$\Rightarrow Run\ \rightarrow\ Sleep\ transition:\ power\ =\ \frac{energy}{time} = 20.0000mW\ <\ 27.6000mW\ =\ Run\ power$$

$$\Rightarrow Sleep\ \rightarrow\ Run\ transition:\ power\ =\ \frac{energy}{time} = 500.0000mW\ >>\ 27.6000mW\ =\ Run\ power$$

As you can see the Sleep → Run transition consumes a huge amount of power compared with simply staying in Run state (≈ $20x$ more). This means that we need to stay in the Sleep state for a quite long amount of time to compensate for the amount of energy lost during the Sleep → Run transition. But how much exactly?

$$\begin{cases} E_{Run\to Sleep} + E_{Sleep\to Run} + P_{Sleep}t_{Sleep} \leq P_{Run}t_{Run} \\ t_{Sleep} = t_{Run} - (t_{Run\to Sleep} + t_{Sleep\to Run}) \end{cases}$$

$$\Rightarrow \begin{cases} 2.02mJ + 0.09mWt_{Sleep} \leq 27.6mWt_{Run} \\ t_{Sleep} = t_{Run} - 5ms \end{cases}$$

$$\Rightarrow 2.02mJ + 0.09mW(t - 5ms) \leq 27.6mWt$$

$$\Rightarrow 2.02mJ - 0.00045mJ \leq 27.6mWt - 0.09mWt$$

$$\Rightarrow 2.01955mJ \leq 27.51mWt$$

$$\Rightarrow t \geq \frac{2.01955mJ}{27.51mW} = 73.4ms$$

This means that moving to the Sleep state is convenient only if the inactive window is longer than 73.4ms.
Now we can understand what is the reason behind the obtained results:

- For workload 1, if we set a $T_{TO} = 4ms$ we remain in Run state in all the inactive windows of the sensing phase, thus avoiding wasting power for transitions of short inactive windows. The only inactive windows where the transition happens are the waiting phases, leading to a massive reduction in power consumption.
- For workload 2, if we set a $T_{TO} = 0ms$ a transition to Sleep state is triggered at each and every inactive window. This is beneficial because all of them last more than 73.4ms, thus leading to the lowest possible power consumption.
  Increasing T_{TO} does not bring any advantage, and if brought above $\approx$ 30ms we shrink the duration of the sensing phase's inactive windows below the 73.4ms threshold. In this case, it is even better to set a higher T_{TO}, for example, $T_{TO} \approx 120ms$ to completely avoid the transition in the sensing phase's inactive windows and keep only the one belonging to the waiting phase.

5 Part 3: Predictive policy

The goal of this last section is to change again the DPM policy and use a predictive (history-based) policy instead. This new scenario opens new possible optimizations.
In fact, we have seen so far that with a timeout-based policy it is only possible to switch from Run to Idle or from Run to Sleep.
This is not the best setup for two reasons:

- In case of only Idle state allowed, if a long inactive window comes in we will consume more than we would consume in the Sleep state
- In case of only Sleep state allowed, if a short inactive window comes in we will consume more because we remained in Run state instead of possibly moving to Idle state

History-based policy solves this problem, allowing both Run → Idle and Run → Sleep transitions. The choice is based on the expected duration of the next inactive window; if the inactive window is predicted to be short we move to the Idle state, otherwise, if it is predicted to be long enough to compensate the Run → Sleep transition we move to Sleep.
But before starting with the explanation of policies we are still missing one point; we calculated when it is worth moving from Run to Idle and from Run to Sleep, but in this case, we can also choose which state is better between Idle and Sleep. Thus we need to calculate also when it is worth choosing Sleep over Idle state.
Using again a system of equations we find the threshold.

$$\begin{cases} E_{Run\to Sleep} + E_{Sleep\to Run} + P_{Sleep}t_{Sleep} \leq E_{Run\to Idle} + E_{Idle\to Run} + P_{Idle}t_{Idle} \\ t_{Sleep} = t_{Idle} + (t_{Run\to Idle} + t_{Idle\to Run}) - (t_{Run\to Sleep} + t_{Sleep\to Run}) \end{cases}$$

$$\Rightarrow \begin{cases} 2.02mJ + 0.09mWt_{Sleep} \leq 0.02mJ + 0.57mWt_{Idle} \\ t_{Sleep} = t_{Idle} - 4.2ms \end{cases}$$

$$\Rightarrow 2.00mJ + 0.09mW(t - 4.2ms) \leq 0.57mWt$$

$$\Rightarrow 2.00mJ - 0.00038mJ \leq 0.57mWt - 0.09mWt$$

$$\Rightarrow 1.99962mJ \leq 0.48mWt$$

$$\Rightarrow t \geq \frac{1.99962mJ}{0.48mW} = 4.16s = 4160ms$$

5.1 Workflow

5-windows based policy

First and foremost we tried the history-based policy already implemented in the DPM simulator, but the obtained results are not even worthy of mention. The problem with this policy is that it predicts the duration of the next inactive window based on the duration of the latest five inactive windows.
Knowing the workload we can firmly state that this policy is not only wrong but also detrimental in terms of power consumption. In fact, the long inactive window of the waiting phase will always be predicted as short, while the first four inactive windows of the sensing phase will probably be predicted as long (depending on the coefficients inserted by the user).
Nevertheless, it is a great starting point to create our own policy.

11-windows-based policy

Starting from the previous policy we extended the *history_inactive* vector from 5 to 11 elements. We know in fact that our workload is very regular in terms of inactive windows: ten short (even 0 length) windows in the sensing phase followed by a single long window in the waiting phase. This means that if the last element (the oldest) of our 11-entries vector is a high number we are again approaching the waiting phase hence the inactive window will be long.
With this said, we can get rid of the input coefficients and simply predict the current inactive window as long as the 11-windows-old one.
To do this we modified the function *dpm_decide_state* in a very simple manner:

```
case DPM_HISTORY:
    /* Day 3: EDIT */
    t_pred = history_inactive[0];
    if (t_pred >= 0) { //always convenient
            *next_state = PSM_STATE_IDLE;
        if (t_pred > 4160) //threshold calculated before
            *next_state = PSM_STATE_SLEEP;
    }
    else {
        *next_state = PSM_STATE_RUN;
    }
break;
```

5.1.1 Active window-based policy

At a certain point, we also noticed that the duration of the next inactive window was not only related to the 11-windows-old one, but also to the current window. We decided then to implement another policy that predicts the next inactive window duration proportional to the currently active window.
To implement this policy we again modified the function dpm_decide_state in a very simple manner:

```
case DPM_HISTORY:
    /* Day 3: EDIT */
    t_pred = 100*history_active;
    if (t_pred >= 0) { //always convenient
            *next_state = PSM_STATE_IDLE;
        if (t_pred > 4160) //threshold calculated before
            *next_state = PSM_STATE_SLEEP;
    }
    else {
        *next_state = PSM_STATE_RUN;
    }
break;
```

5.2 Results

11-windows-based policy

Workload 1:

```
[sim] Active time in profile = 2.924000s
[sim] Inactive time in profile = 1079.652000s
[sim] Tot. Time w/o DPM = 1082.576000s, Tot. Time w DPM = 1082.576500s
[sim] Total time in state Run = 2.924000s
[sim] Total time in state Idle = 120.113400s
[sim] Total time in state Sleep = 959.447900s
[sim] Timeout waiting time = 0.000000s
[sim] Transitions time = 0.091200s
[sim] N. of transitions = 144
[sim] Energy for transitions = 0.0174400000J
[sim] Tot. Energy w/o DPM = 29.8790976000J, Tot. Energy w DPM = 0.2529573491J
```

Workload 2:

```
[sim] Active time in profile = 2.547000s
[sim] Inactive time in profile = 1089.439000s
[sim] Tot. Time w/o DPM = 1091.986000s, Tot. Time w DPM = 1091.986500s
[sim] Total time in state Run = 2.547000s
[sim] Total time in state Idle = 129.456800s
[sim] Total time in state Sleep = 959.869900s
[sim] Timeout waiting time = 0.000000s
[sim] Transitions time = 0.112800s
[sim] N. of transitions = 198
[sim] Energy for transitions = 0.0179800000J
[sim] Tot. Energy w/o DPM = 30.1388136000J, Tot. Energy w DPM = 0.2484558671J
```

Active window based policy

Workload 1:

```
[sim] Active time in profile = 2.924000s
[sim] Inactive time in profile = 1079.652000s
[sim] Tot. Time w/o DPM = 1082.576000s, Tot. Time w DPM = 1082.576500s
[sim] Total time in state Run = 2.924000s
[sim] Total time in state Idle = 0.143900s
[sim] Total time in state Sleep = 1079.416400s
[sim] Timeout waiting time = 0.000000s
[sim] Transitions time = 0.092200s
[sim] N. of transitions = 136
[sim] Energy for transitions = 0.0193600000J
[sim] Tot. Energy w/o DPM = 29.8790976000J, Tot. Energy w DPM = 0.1972918990J
```

Workload 2:

```
[sim] Active time in profile = 2.547000s
[sim] Inactive time in profile = 1089.439000s
[sim] Tot. Time w/o DPM = 1091.986000s, Tot. Time w DPM = 1091.986500s
[sim] Total time in state Run = 2.547000s
[sim] Total time in state Idle = 9.452000s
[sim] Total time in state Sleep = 1079.870500s
[sim] Timeout waiting time = 0.000000s
[sim] Transitions time = 0.117000s
[sim] N. of transitions = 198
[sim] Energy for transitions = 0.0199800000J
[sim] Tot. Energy w/o DPM = 30.1388136000J, Tot. Energy w DPM = 0.1928531850J
```

Analyzing the results of the 11-windows-based policy on the first workload, it is possible to observe a high amount of time spent in the Idle state compared to the expected one. This happens because the first long inactive window duration is always underpredicted due to the lack of an initial history (vector initialized with 0's).
We managed to solve this issue by implementing the active-based policy; by doing so, not only we solved the problem of the first long inactive window, but also reduced the amount of memory needed by the simulator (crucial in certain embedded applications).
Indeed, the results of the new simulations show a lower amount of time in the Idle state (needed only for the small inactive windows) and a lower value of energy consumed as a result.

6 Conclusion

To sum up, in this laboratory we learned how powerful DPM is. We tried both timeout and predictive policies, analyzing the pros and cons of each of them on two different workloads.
For what concerns timeout policies, the focus was on finding the suitable timeout threshold able to reduce power consumption to a minimum; we tested this policy using both Idle and Sleep states to see the impact of a high cost transition.
Regarding predictive policies instead, we tried different methods to perform the prediction and some of them have proven to be more efficient than others.
We are satisfied with the obtained results, especially the last implemented predictive policy showed a result very close to the ideal one.