University of Pisa
Faculty of Engineering

Master Degree in
Computer Engineering

Power Management in Real-Time Embedded Systems

Relators
Prof. Cosimo Antonio Prete
Prof. Pierfrancesco Foglia
Prof. Giorgio Buttazzo

Candidate
Mario Bambagini

Academic year 2009/2010

Abstract

Nowadays, thank to the improvement of computer science and electronic technologies, it is possible to develop very small devices with extraordinary computation capabilities and low cost for unit.

Hi-tech products are everywhere around us and, in next years, almost everything will embed electronic components.

In this scenario, the energy saving issue has become dominant, especially in real-time systems, with a high level of integration with the monitored environment and limited power suppliers. These requirements impose to reduce the energy wasting by organizing as smart as possible job execution and resource use, guaranteeing the satisfaction of constraints (such as real-time, energy and shared resources).

In this thesis, a software, placed between the operating system and user application level, is proposed to meet those targets in hard real-time systems, taking into account servomechanisms, radio modules and CPUs. Servo motors are the muscles of many systems and their analysis leads to the best updating period which reduces the energy according to the applied torque. Concerning radio modules, the proposed module puts them into a low power state when not used, guaranteeing to switch them in fully-operative mode in time by next time they will be used. The CPU manager is the core of the analysis and implements three *inter-task reclaiming* algorithms.

The report is divided in five chapters. Chapter 1 introduces the energy saving problem, part of the state of art and the exploited hardware and software tools. Chapter 2 highlights the module structure, analyzing all the components and how to use them. Chapter 3 describes the algorithms in charge of minimizing the energy consumption of servo motors and radio modules. For each device, a brief introduction, the power model, the implemented policy and relative results have been explained. In Chapter 4, after a brief introduction on algorithm taxonomy, the analysis of several of the most important algorithms in literature has been reported. Then, the implemented algorithms and their results have been considered. Chapter 5 concludes this thesis with the final remarks, pointing out possible future improvements.

Contents

Chapter 1

Introduction

This thesis has been developed as thesis during my *Master Degree* in *Computer Engineering* at the *University of Pisa*. The thesis was carried on at the *Real-Time System lab.* (ReTiS) of the *Scuola Superiore Sant'Anna*, under the supervision of Prof. Giorgio Buttazzo.

The main target consists of implementing a kernel module for improving the energy saving in hard real-time embedded systems with reduced resources. More precisely, the module must minimize the energy consumption (or maximize the energy saving) while guaranteeing the real-time constraints. The application tasks are characterized by the following features:

- hard real-time;
- periodic/aperiodic/sporadic;
- preemptive;
- independent.

Figure 1.1: ReTiS' logo

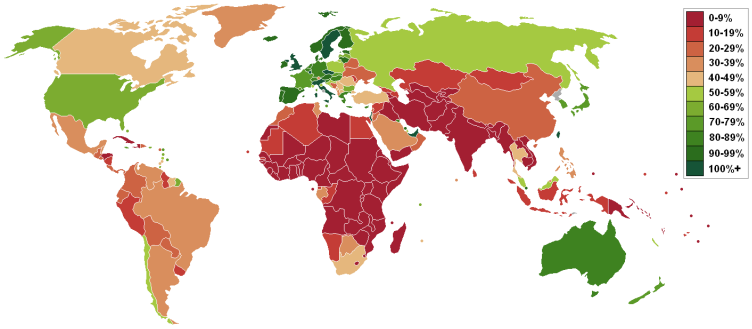

Figure 1.2: Mobile phones per capita

1.1 Power management

Since the the second world war end, the *Information Technology* (/IT) has been having an exponential growth, changing significantly our lives and many concepts, such as distance, information and job are. As reported in Figure 1.2 and Figure 1.3, the number of electronic devices (such as mobile phones and computers) is very high, especially in developed countries.

As introduced by Bill Gates [1], the future trend of computer science and electronic technologies will focus on robotics. More precisely, Bill Gates devised a world where all objects around us are smart and designed to specific tasks, systems rather than the general-purpose Asimov's cyborgs. Such machines, as depicted in Figure 1.4, would communicate together and execute a small set of focused actions, covering many jobs that would not require the human effort any more. Dealing with such a wide group of electronic products, the energy saving problem will be as important as assuring functionality.

The energy saving problem is due to the different growth of the computation capability (and the relative power consumption) and the battery capacity. The first one is led by the Moore's law (Figure 1.5) which states that *the number of transistors in a chip doubles every eighteen months*. On the other hands, the energy density and the capacity of the batteries doubles only every ten years (Figure 1.6).

The power consumption is not the only source of problem, indeed, also the power density is very important (as show in Figure 1.7), as high temperatures lead to lower system performance. The higher the power density, the higher the temperature of the area. This is one of the reasons that pushed the introduction of multicore systems (for instance, consider the Intel's Tejas).

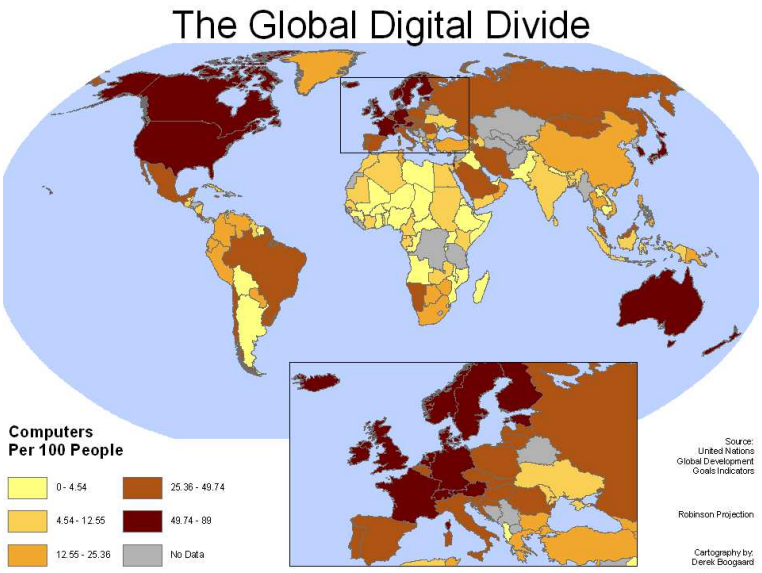

Figure 1.3: Computers per capita

1.2 State of art

There are many techniques for reducing energy consumption in electronic circuits, for example designers may let users change the speed and the voltage of the system or they may divide the circuit in areas with different characteristics (e.g.: geometric, electric). A description concerning those techniques is reported in Chapter 4.

Nowadays, the most CPU producers implement hardware facilities for reducing energy consumption, as the followings:

- *SpeedStep* is a trademark for a series of dynamic frequency scaling technologies (including *SpeedStep*, *SpeedStep II*, and *SpeedStep III*) built into some *Intel* microprocessors that allow the clock speed of the processor to be dynamically changed by software;

- *Intel VRT technology* split the chip into a 3.3V I/O section and a 2.9V core section. The lower core voltage reduces power consumption;

- *Cool'n'Quiet* is a CPU speed throttling and power saving technology introduced by *AMD* with their *Athlon 64* processor line. It works by reducing the processor's clock rate and voltage when the processor is idle. The aim of this technology is to reduce overall power consumption and lower heat generation. The technology is similar to *Intel*'s *Speed-*

Figure 1.4: Example of futuristic house

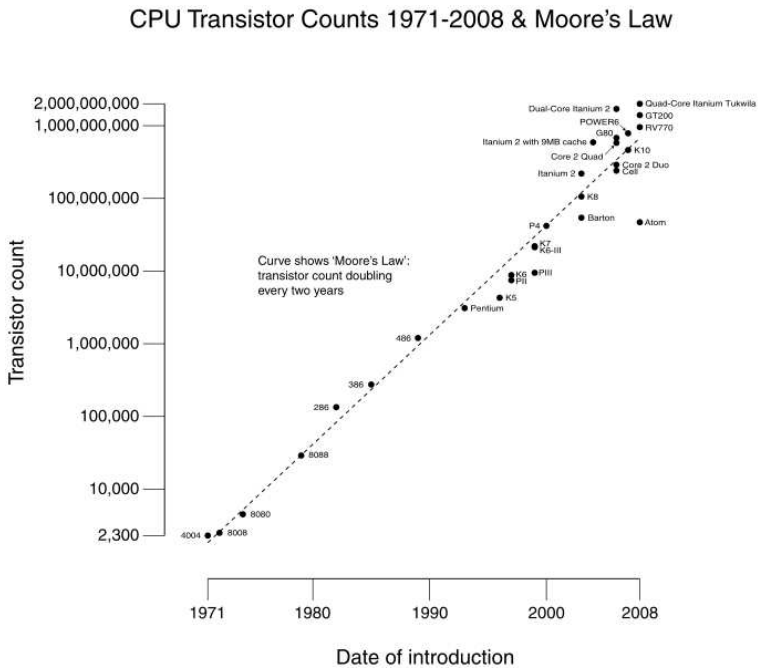

Figure 1.5: Moore's law

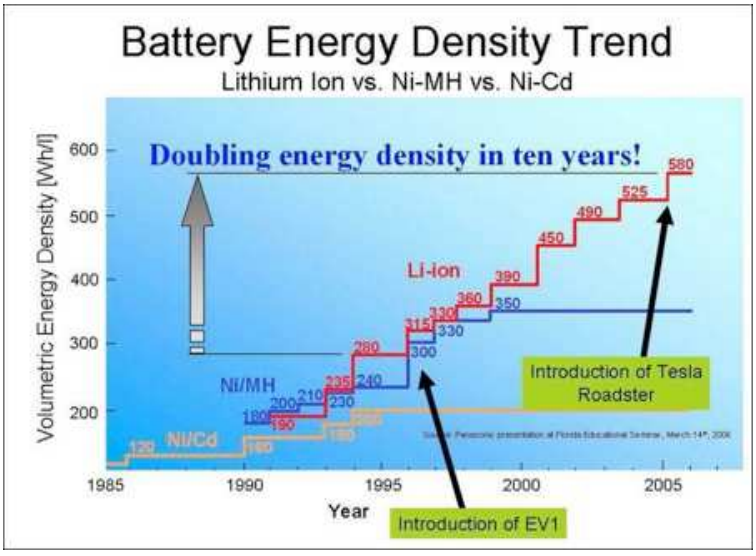

Figure 1.6: Energy density growth

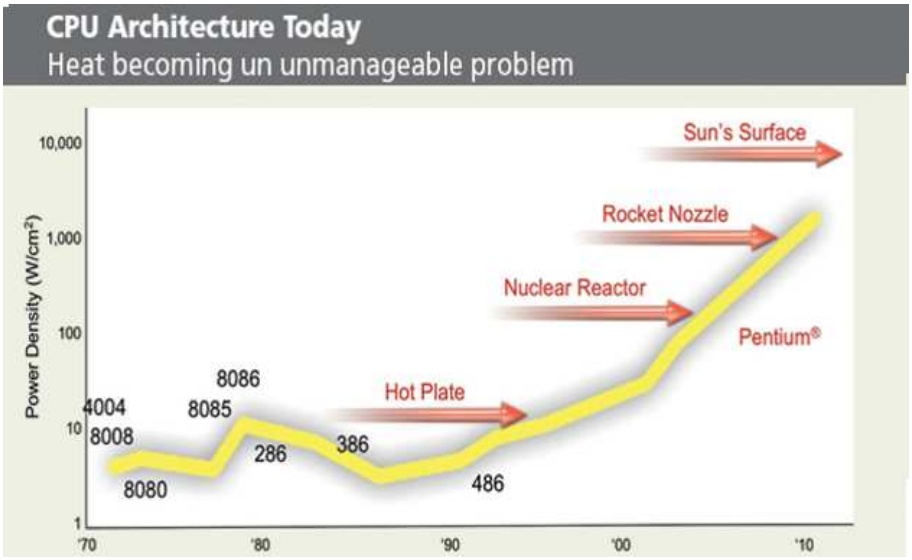

Figure 1.7: Power density in *Intel*'s CPUs

Step and *AMD*'s own *PowerNow!*, which were developed with the aim of increasing laptop battery life by reducing power consumption;

- *AMD CoolCore* technology get more efficient performance by dynamically activating or turning off parts of the processor;
- *PowerNow!* is speed throttling and power saving technology of *AMD*'s processors used in laptops. The CPU's clock speed and VCore are automatically decreased when the computer is under low load or idle, to save battery power, reduce heat and noise. The lifetime of the CPU is also extended because of reduced electro migration, which varies exponentially with temperature;
- *VIA LongHaul* is a CPU speed throttling and power saving technology developed by *VIA Technologies*. By executing specialized instructions, software can exercise fine control on the bus-to-core frequency ratio and CPU core voltage. While the operating system runs, a CPU driver controls the throttling according to how much load is put on the CPU;
- *LongRun* and *LongRun2* are power management technologies introduced by *Transmeta*. *LongRun* was based primarily on aggressively reducing the clock frequency and voltage supplied to the processor, in order to reduce active power consumption. *LongRun2* built further on

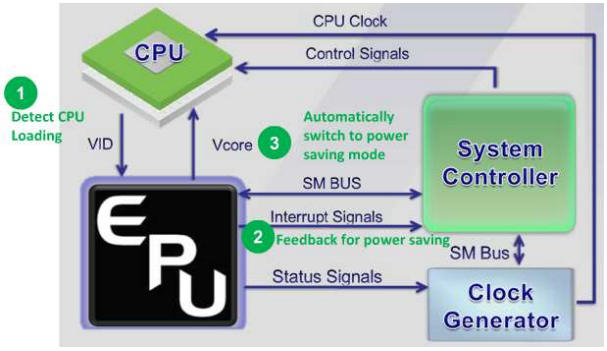

Figure 1.8: *EPU* technology

this by incorporating process technology meant to reduce variations in the manufacturing process and thereby improve yields;

- *IBM EnergyScale* is the techology for the *IBM* platforms;
- *ASUS* has introduced the *EPU* technology (*Energy Processing Unit*, Figure 1.8) on its mother boards. This solution uses a chip to scale the CPU clock according to the CPU load.

The first high-level solution has been APM (Advanced Power Management), which was introduced in the 1992 with the main target of increasing notebook operative time (in those years, their sales rapidly increased). APM was developed by *Intel* and *Microsoft*, whose last version, 1.2, goes back to 1996. *Microsoft* has stopped including APM support since *Windows Vista*. It is based on a layered approach, depicted in Figure 1.9, where the *APM-aware applications* communicates with the *APM driver* (within the operating system) to read or write information related to the power states. The *APM driver* sends information and requests to the *APM BIOS*, generating events. Generated events may regard the power supply states (e.g.: changing in the battery level) and request notifications (from hardware and user). The available functions, for communicating with *APM driver* and *APM BIOS*, are used to manage the APM module, set/get a power state and intercept events.

The following five operative states are supported for the entire computer:

- Full On: the computer is powered on, and no device is in power saving mode;
- APM Enabled: the computer is powered on, and APM is controlling device power management as needed;

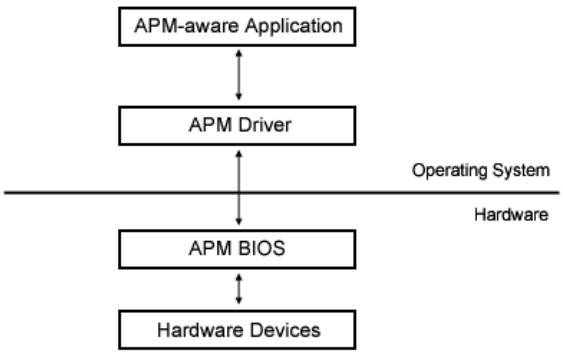

Figure 1.9: APM's layers

- APM Standby: most devices are in their low-power state, the CPU is slowed or stopped, and the system state is saved. The computer can be returned to its former state quickly (in response to activity such as the user keyboard pressing);
- APM Suspend: most devices are powered off, but the system state is saved. The computer can be returned to its former state, but takes a relatively long time. (Hibernation is a special form of APM Suspend state);
- Off: The computer is turned off.

The following four states feature devices power management:

- Device On: the device is in full power mode;
- Device Power Managed: the device is still powered on, but some functions may not be available, or may have reduced performance;
- Device Low Power: the device is not working. Power is maintained so that the device may be woken up;
- Device Off: the device is powered off.

The solution that is actually implemented in almost all the computer is ACPI (Advanced Configuration and Power Interface), developed by *Hewlett-Packard*, *Intel*, *Microsoft*, *Phoenix Technologies* and *Toshiba* in 1996. In order to be compliant, devices must offer four possible states (Figure 1.10) with different power consumptions (only the CPU could have more states).

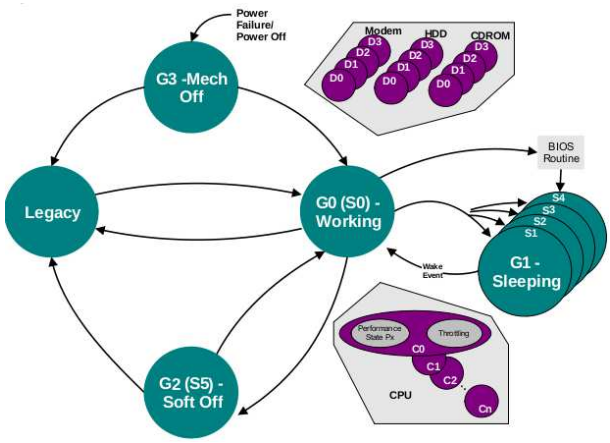

Figure 1.10: ACPI device states

As reported in Figure 1.11, the ACPI architecture is wider and more articulated than *APM*'s. Hardware must be ACPI compliant, meaning that peripherals (even the CPU and chipset) must implement the FFH interface (Function Fixed Hardware), a set of registers usable from the upper levels. The OS-independent level is specified in the *ACPI* standard and contains registers, functions and tables, to uniform the power management of the devices. It introduces AML (ACPI Machine Language) for abstracting the operating system from the machine language of the processor. The real novelty of *ACPI* is OSPM (Operating System Power Management) which is in charge of implementing the real energy saving policies by defining the state of each supported device according to the actual situation (such as workload and battery level). OSPM delegates the power management to the operating system unlike APM which implements such policies in either the BIOS or application level. ACPICA (ACPI Component Architecture) in Figure 1.12 provides an open-source OS-independent implementation of the ACPI specifications. A typical ACPI activity, after unlinking the power supply connector of a notebook, consists of reducing the screen brightness.

ACPI is a good solution for computers but not applicable for embedded systems due to its complexity. In the 2004, the IBM research lab in Austin and *MontaVista* proposed a software module called DPM (Dynamic Power Management) which aims at reducing energy consumption for embedded systems. The architecture, show in Figure 1.13, is composed of several policies stored within the operating system that specify the device states to use (i.e.: rows in the table) according to the system usage. Let us consider the example in Table 1.1. Tasks are divided in three groups (e.g.: TASK+, TASK

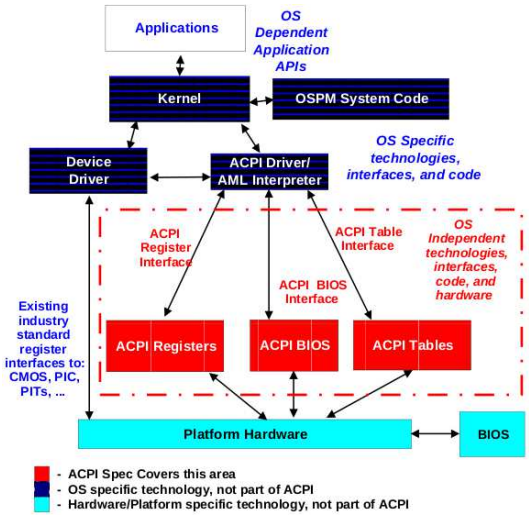

Figure 1.11: ACPI architecture

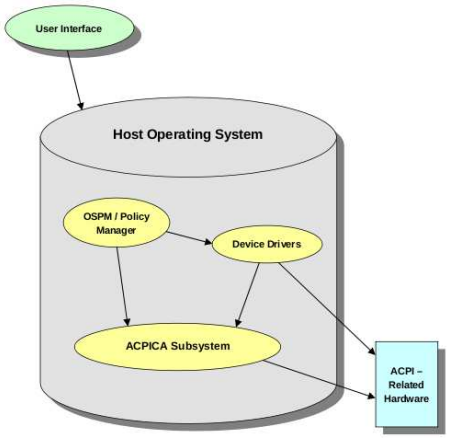

Figure 1.12: ACPI Component Architecture

POLICY	TASK+	TASK	TASK-	IDLE
Battery Critical	100@1.0, TX off	50@1.0, TX off	50@1.0, TX off	33@1.0, TX off
Battery Low	133@1.8, TX on	100@1.0, TX off	50@1.0, TX off	33@1.0, TX off
Battery Good	266@1.8, TX on	100@1.5, TX on	100@1.0, TX on	33@1.0, TX off

Table 1.1: DPM example

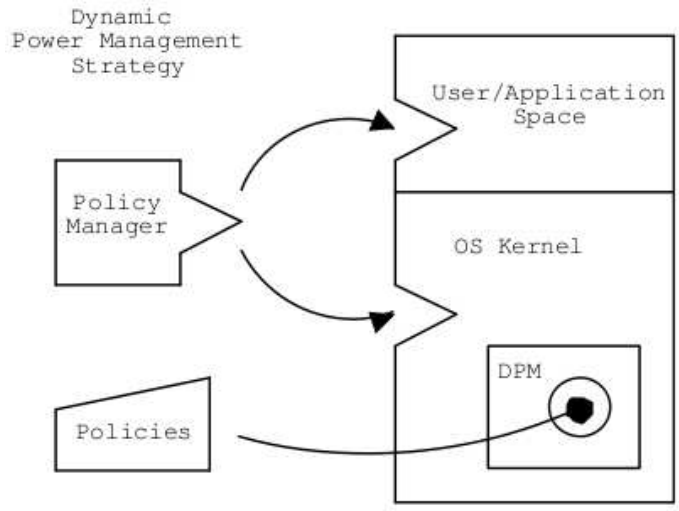

Figure 1.13: DPM architecture

and TASK-) (i.e.: columns in the table) according to the computation requirements. The IDLE group is used when there are not tasks in execution. According to the task in execution (chosen by the scheduler) and the actual policy (chosen by the application level or the Policy Manager), the module uses the configuration in the resulting cell.

Many widespread real-time operating systems (RTOS), under GPL license, don't implement power-aware policies, except *Contiki* and *Nano-RK*. Contiki contains advanced power-aware policies for the wireless communication protocols and lets users throttle the system speed according to the actual workload. More precisely, energy saving related to communications are embedded within the protocol, delegating other policies to the application level. Nano-RK exploits resource reservation techniques to manage the CPU and devices.

1.3 Used Resources

To develop and test the implemented module, a wide set of tools has been used, introduced in this section.

1.3.1 Hardware

The Flex boards ([2], [3]) are the chosen hardware for running the algorithms. The board is equipped with a dsPic33FJ256MC710 ([4]), a 16 bits micro

Figure 1.14: cc2420 transceiver

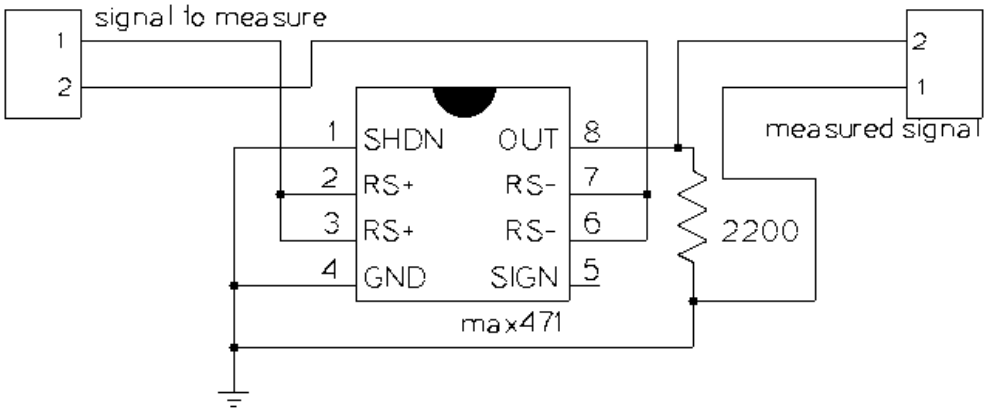

Figure 1.15: Max471 circuit

controller of the dsPic®DSC family produced by Microchip®. The CPU has a 256KB and 30KB as code and data memory, respectively. Several supported peripherals are ADC, timers, output compare, I/O pins, UART and SPI interfaces. The onboard oscillator allows up to 40 MIPS and the PLL is able to generate a wide range of frequencies (between 2 and 40 MIPS). The cc2420 transceiver (Figure 3.12) has been used as wireless interface. This transceiver is plugged in a specific board slot, implementing the physical communication layer. Such module is *IEEE 802.15.4* compliant and offers a maximum bit rate of 250Kbps.

The board has been modified by unsoldering the voltage regulator in order to monitor the absorbed current by the IC Max471 (Figure 1.15) (current-sense amplifier produced by Maxim). Such device is characterized by the following parameters:

- Precision internal sense resistor of 35mΩ;

- 2% accuracy over temperature;
- 3A sense capability;
- 100μA max supply current;
- 3V to 36V supply voltage;
- 0 °C to 70 °C temperature range;
- 8-Pin DIP/SO package.

1.3.2 Software

Erika Enterprise is the real-time kernel which has been used for this thesis. Such RTOS implements both Fixed Priority and Earliest Deadline First, while supporting multitasking, preemptability, shared or private stack, shared resources, alarms (for periodic activations), error handling and scheduling hooks. Erika Enterprise's API is compliant with the OSEK/VDX standard and OSEK Implementation Language (OIL). Erika Enterprise is natively supported and optimized by RT-Druid. RT-Druid is an open and extensible environment, based on XML and integrated in Eclipse, allowing the generation of portable OSEK C code from OIL definitions to create applications that run in real-time.

Chapter 2

Software architecture

This chapter explains the library organization and each task. In the first section, a global view of the system has been reported while, the second and third sections provide the reference concerning API and configuration.

2.1 Structure

Writing code for real-time systems with few resources (e.g.: memory and peripherals), almost always requires that time overheads and footprint size are as small as possible. For this reasons, the chosen architecture has been divided in a set of modules that are:

- static: meaning that components are linked at compile time and not at run time;
- independent: meaning that only the components that are used are loaded.

The software has been studied to be cross-platform and OS-independent (a new porting requires only the changing of the drivers). The module has been placed between part of the operating system and part of the application level as reported in Figure 2.1.

Figure 2.2 shows all the components and the internal architecture of the library. The architecture is an improvement of the idea proposed in [5]. Within the library there are three macro areas: GLOBAL API, CPU and DEVICES. The API block is on the top level containing all the functions available for the user applications. It offers an unique and simple way to communicate with the modules. The three blocks on the right side implement the CPU support, focusing on its consumption. The CPU manager implements and

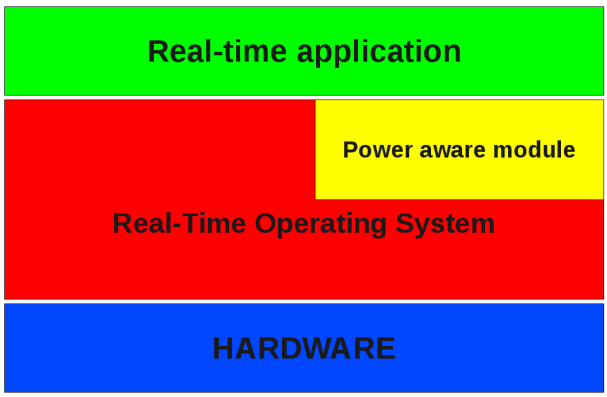

Figure 2.1: Global view

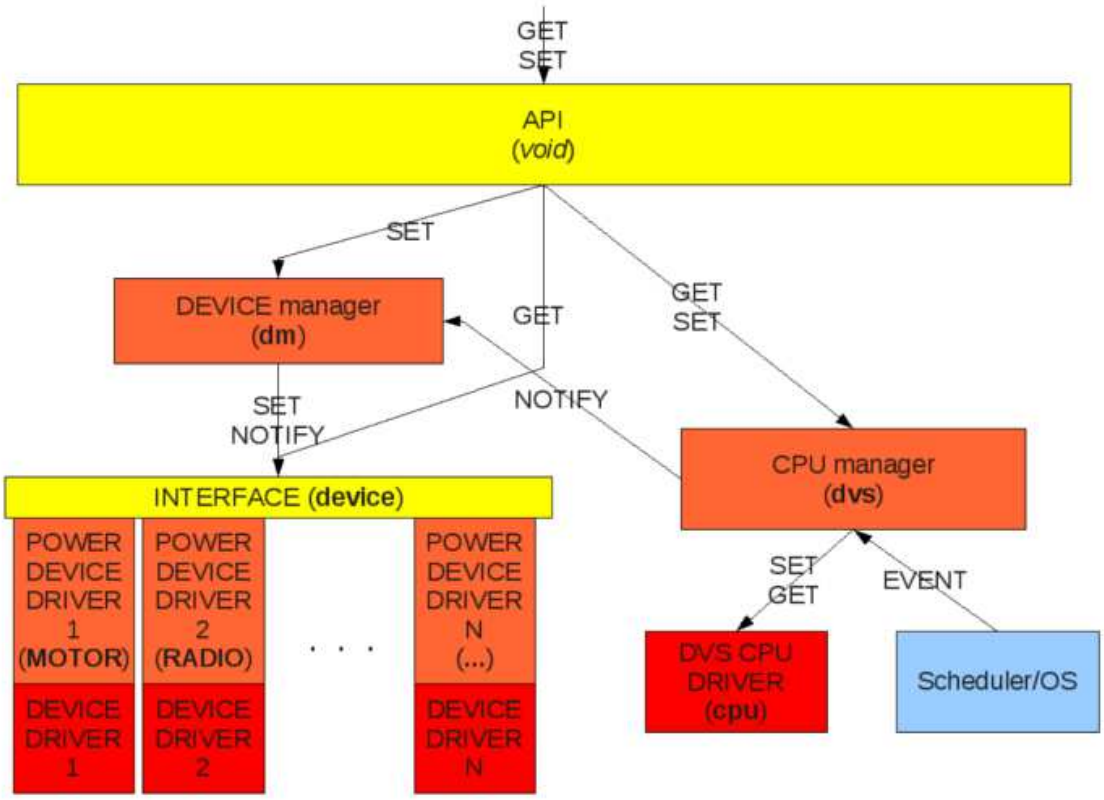

Figure 2.2: Software architecture

executes the power aware policies, for instance it can slow down the CPU speed according to the actual workload. The DVS CPU DRIVER depends on the CPU used and executes the frequency and voltage throttling and returns information about CPU features. Finally, Scheduler/OS represents the signals, from the scheduler, about task events (e.g.: when a task becomes ready or is preempted). On the left side there are the blocks representing the DEVICES area. As explained by their name, they are used to reduce the power consumption of the peripherals. The highest block of the DEVICES area is DEVICE manager which implements, for a particular peripheral, one or more power aware policies. This block is optional, therefore the programmer could enable only the bottom level to use directly the power states of the devices. The INTERFACE level introduces a gate useful to access to the power driver of devices exploiting an unique interface. Each supported device in the library has a small power aware stack, DEVICE DRIVER and POWER DEVICE DRIVER. The lowest level (DEVICE DRIVER) is the driver and is in charge of implementing the device functions. On the other hand, POWER DEVICE DRIVER has two important tasks:

- offering at the upper levels a vision of the power based on a set of states (each state is an operating mode with a known power consumption);
- adjusting the device, using the driver level, according to the power state chosen by the upper levels.

The adoption of a device power model based on a set of states makes their management easier.

2.2 How to use it

The module is configured by the following set of preprocessor declarations:

- __USE_PM__: enable the module;
- __USE_PM_DVS__: enable the CPU area (only one policy can be enabled);
  - __USE_PM_DVS_NOTHING__: the CPU manager does nothing;
  - __USE_PM_DVS_OLDVS__: the CPU manager executes OLDVS [6];
  - __USE_PM_DVS_BSDVS__: the CPU manager runs a OLDVS variant;

  - __USE_PM_DVS_BSSSDVS__: the CPU manager runs a OLDVS* variant;
  - __USE_PM_DVS_EXPS__: enable the EXtreme Power Saving feature;

- __USE_PM_DEVICE__: enable the *DEVICES* area;
  - __USE_PM_DEVICE_MANAGER__: enable the DEVICE manager;
  - __USE_PM_DEVICE_SERVO__: enable the power stack for the servo motor;
  - __USE_PM_DEVICE_RADIO__: enable the power stack for the radio module;
  - __USE_PM_DEVICE_TIMER__: enable the power stack for the timer.

The header to be included is:

```
#include <pm.h>
```

In the application, the programmer must call the hook functions of the power aware library when events happened. For example, with Erika Enterprise, the code is the following.

```
void StartupHook (void)
{
        /*... other code ...*/
        pm_hook_osStart();
        /*... other code ...*/
}
void ShutdownHook (StatusType Error)
{
        /*... other code ...*/
        pm_hook_osStop();
        /*... other code ...*/
}
void ErrorHook (StatusType Error)
{
        /*... other code ...*/
        if ( error in power aware library )
                pm_hook_error();
        /*... other code ...*/
```

```
}
void PreTaskHook (void)
{
        /*... other code ...*/
        pm_hook_taskStart ( this_task  );
        /*... other code ...*/
}
void PostTaskHook (void)
{
        /*... other code ...*/
        pm_hook_taskStop ( this_task );
        /*... other code ...*/
}
void ActivateTaskHook (void)
{
        /*... other code ...*/
        pm_hook_taskLaunch ( this_task );
        /*... other code ...*/
}
void DeActivateTaskHook (void)
{
        /*... other code ...*/
        pm_hook_taskFinish ( this_task );
        /*... other code ...*/
}
```

If you want to use CPU policies (OLDVS, BSdvs or BSSdvs), you must also specify some information about the tasks, such as the nominal frequency and the WCIN (Worst Case Instruction Number). The following files must contain the data:

- pm_dvs_dataStructures.h: containing two macros, PM_DVS_F_NOM and PM_DVS_NUMBER_OF_TASKS. The first one is the nominal frequency (Instructions Per Second) which satisfies the feasibility condition. The second one is the total number of tasks in the system (the background task is not considered);

- pm_dvs_dataStructures.c: storing, in the array pm_dvs_WCIN, the WCIN of each task. Indexes are the same of the sorting in the declaration in the *OIL* file (the first task has index 0, the second has index 1', and so on).

2.3 API

This section reports a complete list of the functions visible from the API level.

All the power-aware library functions start with the prefix *pm_*, followed by the action (*set*, *get*, *notify*, *hook*, *exe* or *force*) and the object.

The operating system uses the following block of functions to communicate an event. They are visible if *__USE_PM__* is defined.

```
//starting point of the power aware manager
void pm_hook_osStart ();
//finishing point of the power aware manager
void pm_hook_osStop ();
//error handler
void pm_hook_error();
//a new task is now ready
void pm_hook_taskLaunch (t_pm_taskID TaskID);
//a task has just finished
void pm_hook_taskFinish (t_pm_taskID TaskID);
//a task has become RUNNING
void pm_hook_taskStart (t_pm_taskID TaskID);
//a task has become NOT RUNNING
void pm_hook_taskStop (t_pm_taskID TaskID);
```

The following block of functions lets users retrieve information about CPU speed and to force a new one. They are available only if *__USE_PM__* and *__USE_PM_DVS__* are defined.

```
//return the actual dvs policy
t_pm_dvs_policy pm_get_dvsPolicy ();
//return the actual IPS (Instructions Per Second)
t_pm_ips pm_get_ips (t_pm_core core);
//return the actual frequency of the devices
//(it could be not equal to IPS)
t_pm_freq pm_get_f (t_pm_core core);
//set a new IPS value
t_pm_ips pm_set_ips (t_pm_core core, t_pm_ips ips);
//indicate if a EXtreme Power Saving policy is available
t_pm_bool pm_get_eXps ();
//force the system to go in EXtreme Power Saving mode
void pm_force_eXps ();
```

The following functions are used to manage the device consumptions. They are available if *__USE_PM__* and *__USE_PM_DEVICE__* are defined.

```
//set a parameter of a device (for DEVICE Manager)
void pm_set_conf (t_pm_device device, void *arg);
//number of power states of a device
t_pm_numState pm_get_numState (t_pm_device device);
//return the actual power state of a device
t_pm_state pm_get_state (t_pm_device device);
//set a new power state of a device
void pm_set_state (t_pm_device device, t_pm_state state);
//set the new callback function
void pm_set_callback (t_pm_callback callback);
//call the callback function
void pm_device_exe_callback (t_pm_device device, void *other);
```

The following functions whose behaviours depend on the active areas are usable if *__USE_PM__* and either *__USE_PM_DVS__* or *__USE_PM_DEVICE__* are defined.

```
//return the total power consumption
t_pm_power pm_get_power ();
//return the actual power consumption of a device
t_pm_power pm_get_power_1 (t_pm_device device);
//return the power consumption of a device in a state
t_pm_power pm_get_power_2 (t_pm_device device, t_pm_state state);
//return the thermal value
t_pm_heat pm_get_temperature (t_pm_device device);
```

Chapter 3

Device manager

This chapter describes how the supported devices are handled. The devices taken into account are timers, radio modules and the servo motors. For each of them, it has been reported how the device works, the implementation details (driver, power driver and part of Device Manager) and the relative consumption results.

3.1 Timer

Timers are the simplest managed device, even though they require negligible energy, as they are in charge of managing the system time, keeping the system consistency. As reported in Figure 3.1, the timer driver is provided by the proposed library for managing all the available timers. The power driver introduces two possible states for the device, *ON* and *OFF*. User applications can switch on and off the timer through this library. The most important aspect is the linking between the INTERFACE and CPU block, meaning that when a speed is scaled, the module adjusts the timer period in such a way the timer interrupt will still occur as desired (after the same amount of time). If the module does not find a valid new period, the callback function (whose implementation is provided by the user) to manage such unacceptable events is called.

3.2 Servo

A servomechanism, or servo, is an automatic device that uses error-sensing feedback to correct the performance of a mechanism [7].

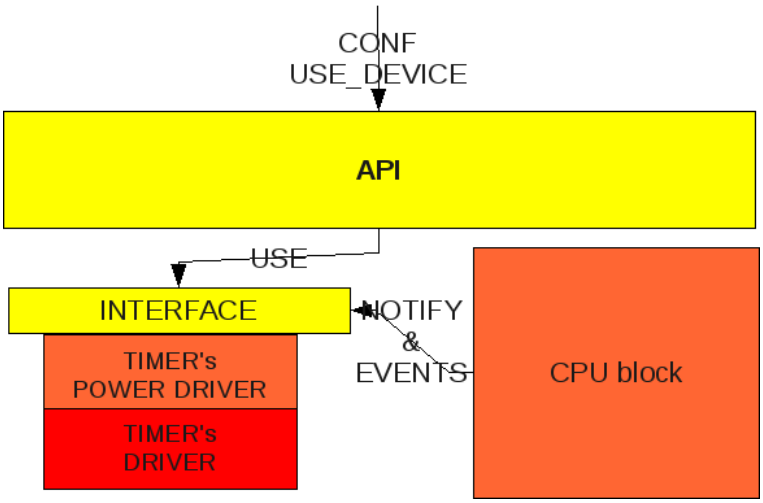

Figure 3.1: Timer's stack

Figure 3.2: Example of servo motor use

3.2.1 How it works

These devices are used in many small-scale applications, such as car steerings, airplane flaps and boat rudders, as depicted in Figure 3.2.

In the experiments, the HS-645MG, produced by Hitec® and reported in Figure 3.3, has been used. Such device has a maximum torque of $7.7Kg * cm$ (with a power supply of $4.8V$), the maximum speed is $0.24sec/60°$ and all the gears are made in Alumite/MP. Electrically, a servomechanism has three wires: power supply (red), ground (black/orange) and command line (yellow).

Inside the servo motor, four parts can be identified. The first part (from the left side in Figure 3.4) consists of the DC motor and the electronic con-

Figure 3.3: Servo motor HS-645MG

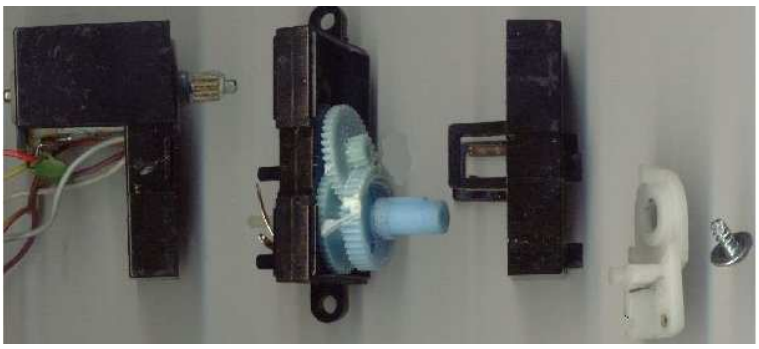

Figure 3.4: Inside a servo motor

troller (for managing motor power supply). Next part is the position feedback potentiometer which informs the control system about the current arm position. The third component is the reduction gear for reducing the motor speed and increasing the engine torque. The last part is the actuator arm.

The servo is moved by sending a pulse signal via the control wire. The width of the pulse informs the servo about the wished angle. A pulse of $1.5ms$ sets the servo at the central neutral position (90 °) Pulses long $1.25ms$ and $1.75ms$ set the servo at 0 ° and 180 °, respectively. Physical limits and timings of the servo hardware depend on brands and models, but generally servo's angular motion is more or less 180 ° and the neutral position is for $1.5ms$ pulse. Controllers update the arm position every time that receive a pulse. Updating periods are within the range $[10, +\infty]ms$, even though a period of $20ms$ is almost always used. Obviously, the more frequent the updating is, the lower the error is. When the controller receives a pulse signal, the PID algorithm executes the following operations:

1. subtracting the required angle to the read value (from the potentiometer);

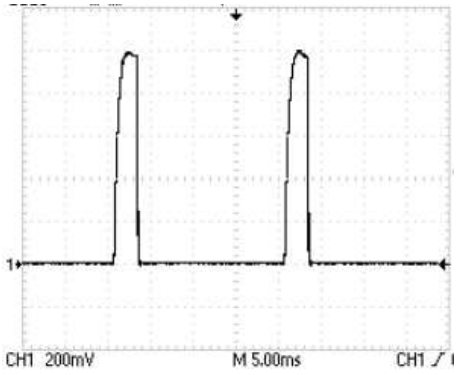

Figure 3.5: Example of current consumption for correcting angular error

2. comparing the result, if it is 0 (or less than ϵ), the system will wait a new pulse, going back to the previous step;

3. feeding the DC motor with current for a time proportional to the detected angular error;

4. if the previous execution had the same trend, the controller will decrease the mean value of the provided current, otherwise it will increase it.

In Figure 3.5 an example of the absorbed current has been reported, considering an updating period of $20ms$ and a small load (leading to a small error). The consumption is present only in the first part of the period and the arm is in a stable state as the mean current value is constant.

Another interesting example has been reported in Figure 3.6 concerning a configuration where the arm has been blocked, leading to a constant high angular error. Also in this case, the updating period is $20ms$. In the first picture part, the initial required current is really high (almost $1A$). On the other hand, in the second part, after several periods, the provided current is significantly decreased as the error has not changed.

3.2.2 Implementation

The servo manager aims at selecting an appropriate period according to the actual load.

The driver level, exploiting one of the output compare modules of the micro controller, is composed of the following four functions:

- *void servo_init()*, initialize the servo;

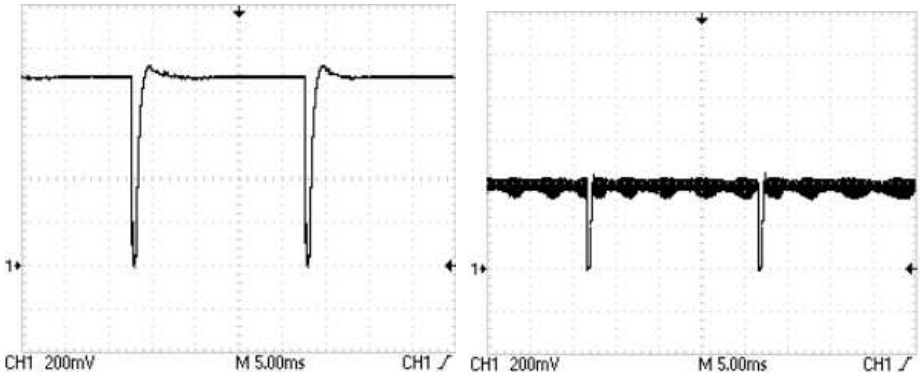

Figure 3.6: Example of current consumption with constant high error

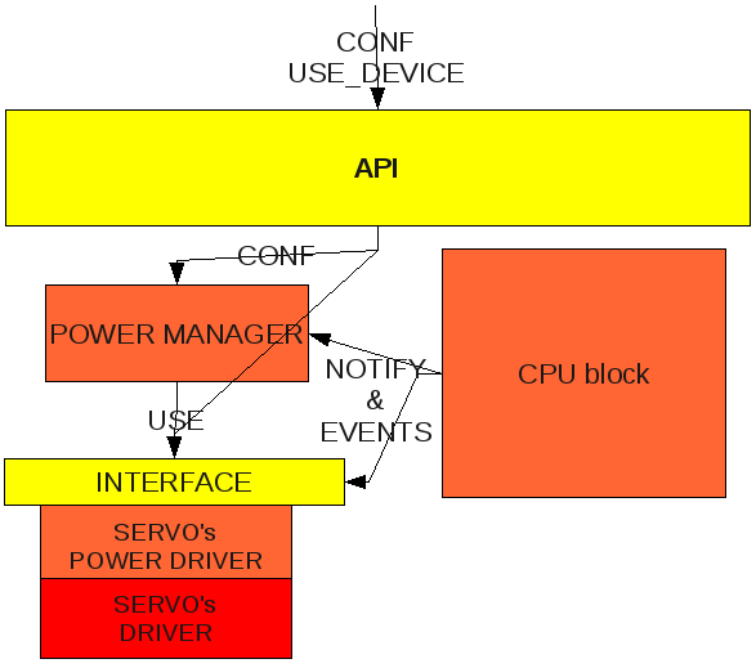

Figure 3.7: Servo stack

- *void servo_set_period(EE_UINT32 IPS, EE_UINT8 period_ms)*, set a new period;
- *void servo_set_angle(EE_UINT32 IPS, EE_UINT8 angle)*, set a new angle;
- *void servo_close()*, disable the servo.

The upper level offers n states. The first one indicates that the servo is off. The other states activate the motor setting a particular period.

If the Device manager is active, the application layer can inform it about the torque applied to the actuator arm. The policy uses such information to look for in an internal data structure the most appropriate period for minimizing the energy consumption. The applied torque can be detected by particular sensors that, unfortunately, are not included in the servo motor in use.

Notifications, from the CPU block concerning speed scalings, let the module modify the period of the output compare device in order to guarantee the performance.

3.2.3 Results

The torque applied to the servo mechanism arm has been obtained as reported in Figure 3.8.

The servo motor is controlled by the output compare of the Flex board at $3.3V$. The power supplier ($5V$) is taken from an external source, and the current sense amplifier monitors the absorbed current through such line. The first result has been that the electronic on board the servo motor consumes $12.56mA$.

The first idea for reducing the power consumption has been use the longest updating period to hold the load that:

1. does not let the arm oscillate excessively;
2. would maintain the load for an infinite time;
3. guarantees an angular error lower than 5.

Results which have been obtained are shown in the Figure 3.9. The maximum torque is $2.54kg * cm$ (and not $7.7kg * cm$) as greater values are supported only for few seconds.

According to these data, eight states have been created for matching the applied torque with the longest period that properly hold the load, as

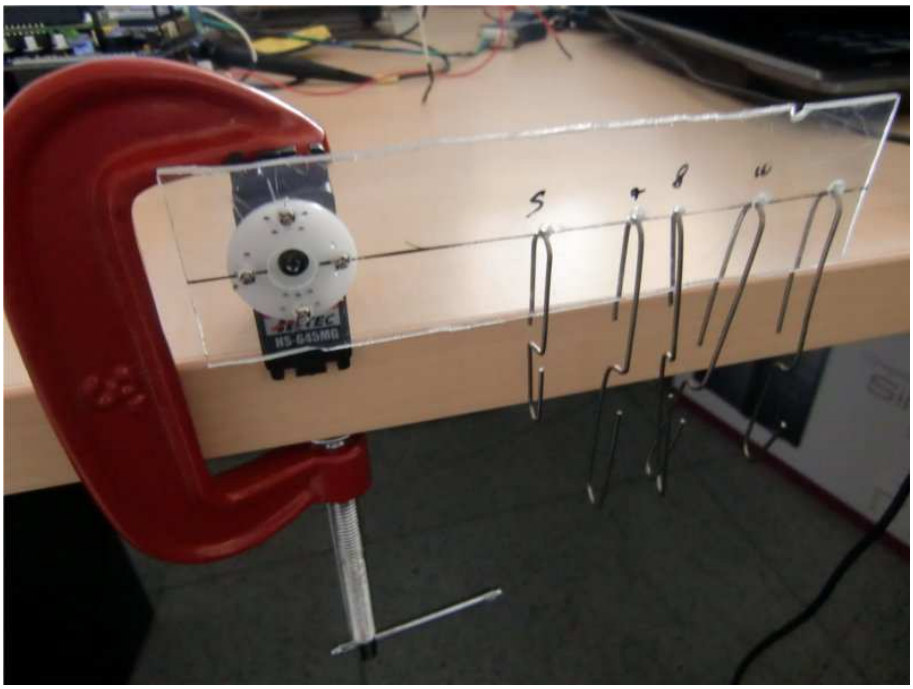

Figure 3.8: Arm extension to apply the desired torque

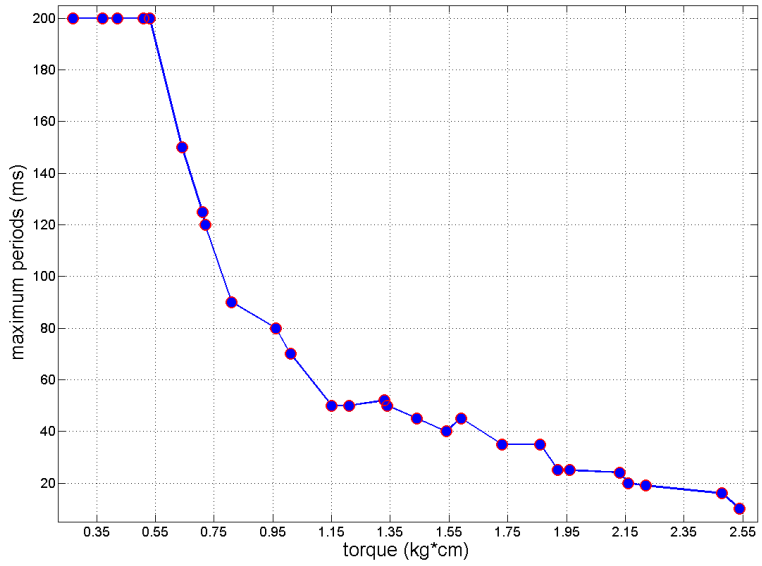

Figure 3.9: Longest usable updating period according to the applied torque

State ID	max torque (kg*cm)	period (ms)
0	OFF	OFF
1	0.53	200
2	0.72	120
3	1.01	70
4	1.59	40
5	1.86	35
6	2.16	20
7	Infinite	10

Table 3.1: Servo states which exploit the maximum period for each torque

State ID	max torque (kg*cm)	period (ms)
0	OFF	OFF
1	0.50	10
2	0.80	40
3	1.01	70
4	1.06	40
5	1.45	20
6	1.53	40
7	1.67	35
8	Infinite	10

Table 3.2: Updated servo states

reported in Table 3.1. Each row means that, up to the indicated torque, the servo can use the indicated period. In the first state, the servo is off and in the last it uses the minimum period (maximum force) for all the heavier loads.

To test the effectiveness of such an approach, the relative power consumptions have been measured. Lines not associated to the policy manager use a fixed updating period.

According to these data, the first idea seems not to be the best strategy, and for this reason, the state set has been updating in order to exploit for each possible torque, the updating period that actually minimize the power consumption. The new states are reported in Table 3.2 and the comparison between the previous and final state set in Figure 3.11.

In such scenario, a trade-off may be introduced concerning the number of states. On one hand, an high number of states would further reduce the

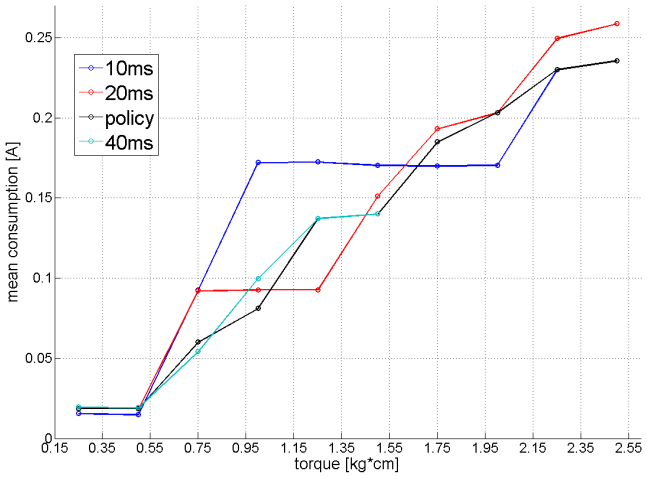

Figure 3.10: Power consumption of the states in Table 3.1

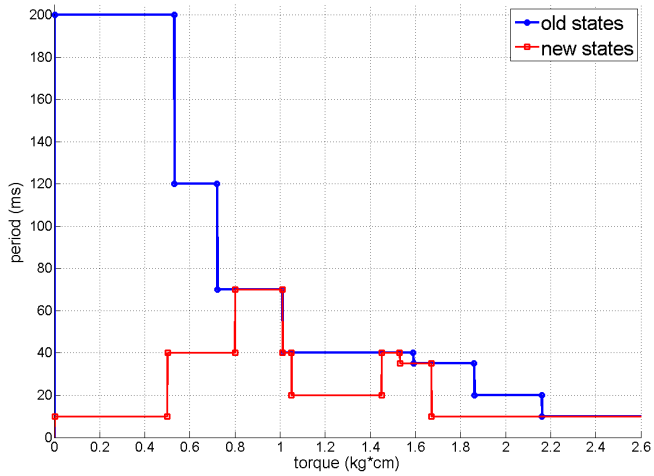

Figure 3.11: Comparison between states in Table 3.1 and Table 3.2

Figure 3.12: EasyBee module

energy wasting, but on the other hand, looking up within the data structure would take longer.

As it has been proved, the initial idea turned out to be wrong, leading to a new version entirely based on the actual power consumption. In conclusion, in several cases (especially with heavy loads) the sum of many small errors is less than the sum of few big errors. With torques lighter than $1kg * cm$, it is more suitable to use the shortest period ($10ms$) as the reactivity to new loads is higher.

3.3 Radio module

3.3.1 How it works

For the wireless communication, the EasyBee module, reported in Figure 3.12, has been used, embedding the cc2420 transceiver.

The module is a DSSS baseband modem at $2.4GHz$, with maximum bit rate equal to $250Kbps$. The device implements the physical layer of the IEEE 802.15.4 communication stack as reported in Figure 3.13. The overhead is made up of six bytes: four bytes (Preamble Sequence) for the synchronization and two bytes (Start of frame and Frame Length) for marking the packet.

The module communicates with the main board through the SPI interface. The main system can access at the functions of the cc2420 reading and writing its registers. The transceiver offers also buffering services for both transmitting and receiving, detaching the main application from the effective managing of the low level operations.

There are many operating modes that the device may use, and each of them is characterized by a particular task and power consumption. The lowest consumption is obtained in Sleep Mode is obtained when all the components are off, including the Voltage Regulator and the oscillator. The

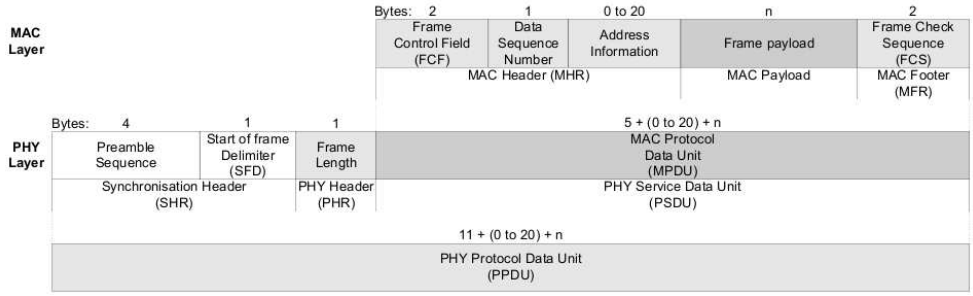

Figure 3.13: PHY and MAC layers of IEEE 802.15.4

drawback is the required time to wake the module up, indeed it is not negligible in real-time applications. Other low power states are Power Down and Idle. Although the first solution is similar to sleep mode, it allows the access to the registers and stack. The Idle mode switches off only the antenna. Concerning the active states, there are a receiving mode and five transmitting states (each of them with a different maximum distance of transmission and BER). Other features which have not been used in this thesis are:

- hardware encryption with the algorithm *AES-128*;
- battery monitoring to retrieve information related to the external power supplier;
- RSSI (Received Signal Strength Indicator) capability, useful to set an appropriate transmission power according to the distance among the nodes;
- LQI (Link Quality Indicator) detector;
- 16 channels with steps of 5 MHz.

3.3.2 Implementation

Radio module stack follows the main guideline as shown in Figure 3.14.

The driver is already available with the Erika Enterprise OS. Three states have been implemented: SLEEP, IDLE and ON. The first state puts all the components off, except for the waking up circuit. The IDLE state disables the RF circuits but the other parts are fully powered. The last one represent the fully operative mode. The Device Manager does not implement a real policy but manages:

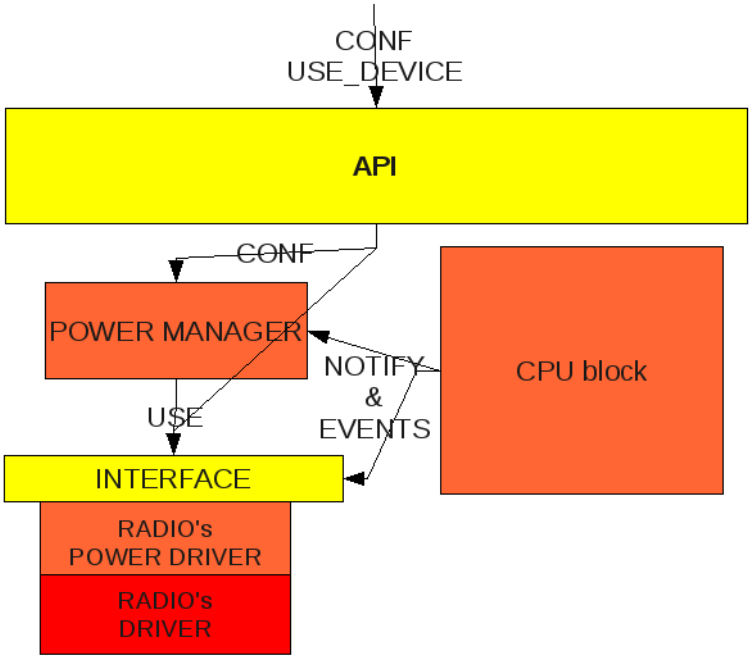

Figure 3.14: Radio stack

- transitions from ON to either SLEEP or IDLE, according to the estimated time of the next utilization;
- the waking up from either SLEEP or IDLE so that the module will be ON at the end of the estimated time of the next utilization (it is a crucial point as the waking up operation may take many milliseconds).

Since this is a layered architecture, the component do not know the required time from the application layer and communication protocols so, the user software must inform Device Manager when such applications need to use the radio module again. Notifications, from the CPU block concerning speed scalings, are used to set a new period for the waking up timer and a new scaler for the SPI interface.

To test the blocks, a simple protocol has been developed. Such protocol is TDMA (Time Division Multiple Access) considering a star topology (n nodes and only one coordinator in the network). Each time slot is divided in two parts. The first part consists of a waiting phase (for avoiding overlapping) and the second is the transmission time. All the communications are unsorted by the coordinator. Each node knows statically what its time slot is (the slot is allocated also when the node is off). The first slot starts when the

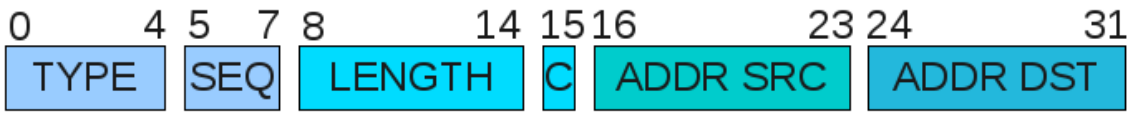

Figure 3.15: Packet header of the proposed protocol

transmission of the coordinator ends. When all nodes end their slots, there is again the slot of the coordinator. In the implementation, a transmission phase longer than the time necessary to send an entire transmission buffer has been used. All the messages have a header of 4 bytes, with the structure described in Figure 3.15. Where the fields have the following meanings:

- TYPE: explains what the message contains. Possible values are:
  - DATA: the message contains data for the application layer;
  - ACK: this is an acknowledge related to the received data messages;
  - ARE_YOU_ALIVE: the coordinator asks if the node is alive;
  - I_AM_ALIVE, a node answers that is ok.
  - TX_POWER_DOWN: the coordinator informs the node to reduce the transmission power;
  - TX_POWER_UP: the coordinator informs the node to increment the transmission power;
- SEQ: sequence number of the message in the stream;
- LENGTH: number of bytes of the payload ($[0, 123]$);
- C: control bit (parity check);
- ADDR SRC: source address;
- ADDR DST: destination address.

3.3.3 Results

The overheads due to the waking up operation are $1.25ms$ and $4.65\mu s$ from SLEEP and IDLE, respectively. In order to read the absorbed current, the power supply pin of the transceiver has been unsoldered and then connected to the current sensor input, leading to the system depicted in Figure 3.16. In such a way, the measured current is only due to the transceiver.

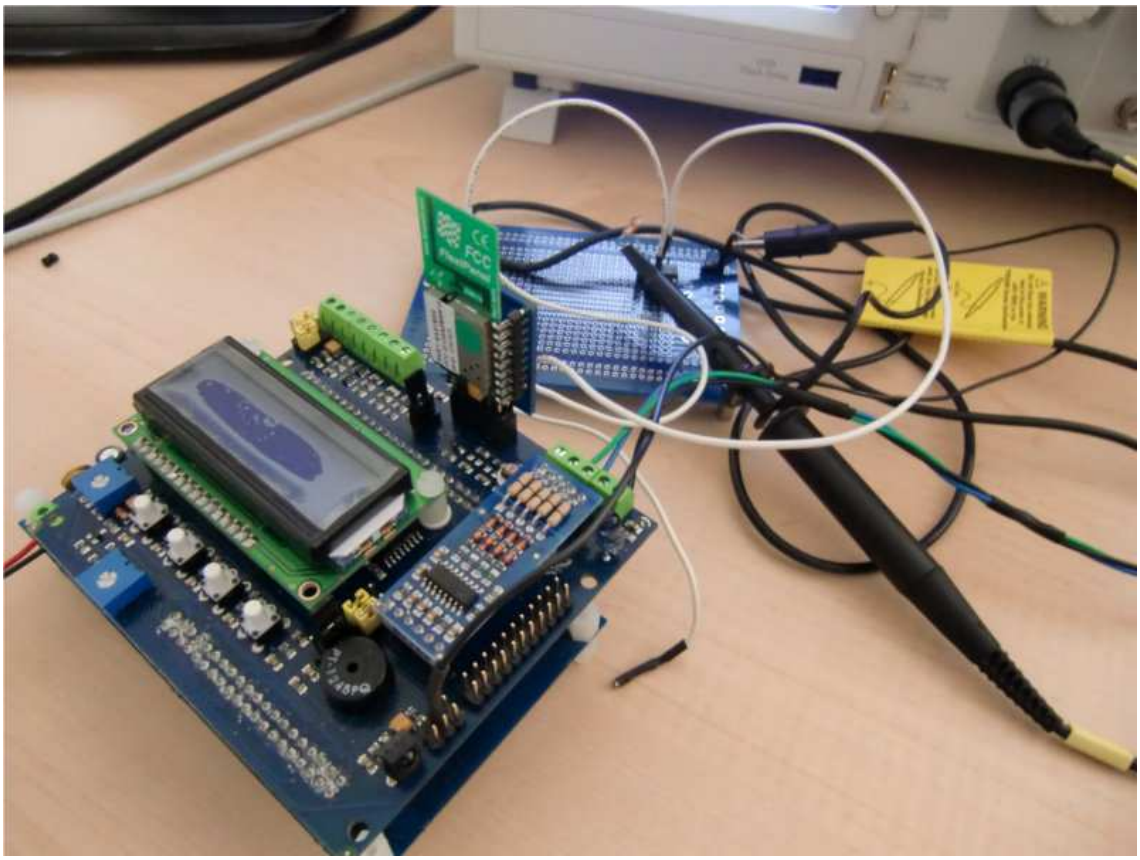

Figure 3.16: Hardware related to the measurement and measured systems

state	datasheet (mA)	measured (mA)	difference (mA)
Sleep	0.00002	6.03	6.03
RX mode	18.80	23.60	4.80
TX mode (max)	17.40	19.25	1.85

Table 3.3: Radio module consumptions

Parameter	time (s)	symbol
Safety interval	0.0015	T_{safety}
Max transmission (node)	0.0045	$T_{max\_tx\_node}$
Min transmission (node)	0.0015	T_{tx_node}
Min transmission (coordinator)	0.0003125	T_{tx_coord}
Hyper period	0.1728125	
T_{TX_bit}	$3.9 * 10^{-6}$	
T_{TX_byte}	$31.25 * 10^{-6}$	
T_{TX_buffer}	$3968.75 * 10^{-6}$	

Table 3.4: Time parameters

First of all, the measured data related to the consumption in the various states have been compared with the ones in the datasheet of the cc2420 transceiver, as reported in Table 3.3. Although the consumptions in TX and RX modes and IDLE state are similar, in sleep mode the measurements are significantly different.

In next test, 29 nodes and a coordinator have been considered. Every $0.1s$, each node generated an amount of data to be transmitted characterized by a Gaussian distribution with mean value and variance equal to 20 and 4, respectively. Table 3.4 reports all the time parameters of the wireless network. The Safety interval, that represent the first part of the communication slot, is longer than the waking up time, letting the first node switch off after the coordinator transmission. The Max transmission time is greater than the time useful to send a buffer of 127 bytes ($0.00396875s$).

Analytically the mean power consumption, considering and not considering low power state, can be computed exploiting the two following formulas. $P_{WITHOUT-SLEEP} = [T_{safety} + T_{tx\_coord} + (T_{safety} + T_{tx\_nodo}) \times (\#nodes - 1) + T_{safety}] \times P_{rx} + T_{tx\_nodo} \times P_{tx}$ $P_{WITH-SLEEP} = (T_{safety} + T_{tx\_coord} + T_{safety}) \times P_{rx} + T_{tx\_node} \times P_{tx} + (T_{safety} + T_{max\_tx\_nodo}) \times (\#nodes - 1) \times P_{sleep}$

Results are computed and measured considering a horizon of analysis of $1s$. Table 3.5 reports the obtained consumptions disabling (second row) and

Sleep	From formulas Datasheet(mJ)	Measurement(mJ)	Measured(mJ)
Unused	62.00	77.74	77.98
Used	1.69	21.37	16.64
Ratio	36.74	3.64	4.69

Table 3.5: Overall consumptions in $1s$

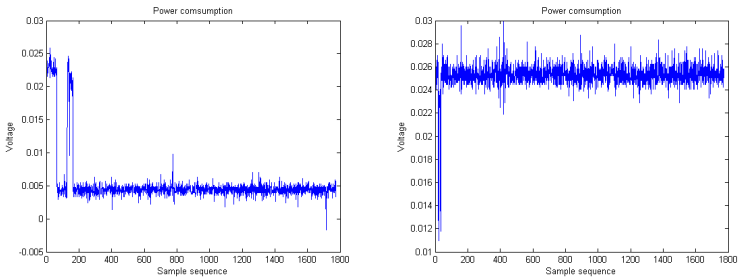

Figure 3.17: Absorbed current of the second node with and without exploiting the sleep state in a hyper period

disabling (third row) the low power feature. The results computed analytically using the consumptions in the datasheet and measured are reported in the second and third columns, respectively. The consumptions entirely measured from the hardware are noted in the last column. Concerning the ratios related to the last two column, the last one is greater as the estimating formula consider null waking up overhead. The difference between the ratios is significant due to the difference between the absorbed currents in sleep mode.

Figure 3.17 reports to screenshots about the power consumption of the second node during the hyper period with and without exploiting the sleep feature of the module.

According to the results of this experiment, assuming to feed the transceiver with 4AA batteries in series (each one characterized from 2000mA, 1.2V), the resulting life-time of the radio module would be the ones in Table 3.6.

	From formulas		
Sleep	**Datasheet(h)**	**Measurement(h)**	**Measured(h)**
Unused	154.84	123.49	123.11
Used	5688.35	449.13	577.04

Table 3.6: Transceiver life-time assuming 4 batteries (2000mA, 1.2V)

Chapter 4

CPU manager

4.1 Introduction

Power management in hard real-time systems is a classical optimization problem that consists of:

- objective: minimizing overall energy consumption or maximizing saved energy;
- constraints: guaranteeing real-time deadlines.

The most general power consumption model has been proposed by Martin et al. [8], reported in Equation 4.1 representing the overall system consumption.

$$P(s) = K_3 \times s^3 + K_2 \times s^2 + K_1 \times s^1 + k_0 \tag{4.1}$$

Where, the meanings of the components are:

- s: normalized frequency $s = f/f_{max}$;
- K_3: component that depends on both the voltage and frequency;
- K_2: component that includes the non-linearity of the voltage regulator;
- K_1: component that consider all the components that could vary only with the frequency;
- K_0: component that consider effects not depending on the speed and voltage.

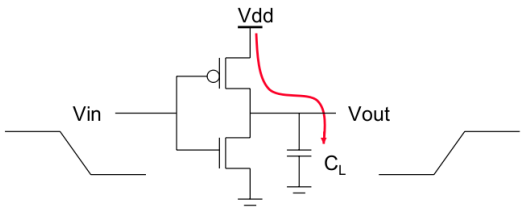

Figure 4.1: Dynamic component of the power consumption

Even through this model is generic, it fits really well the actual CMOS technology. More precisely, the formula of the power consumption in a transistor is:

$$P_{gate} = C_L \times V_{DD}^2 \times f_{clk} \times p_{switch} + t_{sc} \times V_{DD} \times I_{peak} \times f_{clk} \times p_{switch} + V_{DD} \times I_{leakage}. \tag{4.2}$$

Then, the total power consumption of a circuit can be formulated as:

$$P_{tot} = \sum_{i=1}^{\#gates} P_{gate_i}. \tag{4.3}$$

Where p_{switch} is the *activity factor*, indicating the probability that a switching event occurs in a transistor. As reported in Figure 4.2, the power consumption has three components:

- Dynamic power (Figure 4.1), the necessary power to load and unload the output capacitors (C_L, depending on the fan-out and wire lengths) of the gates. This component is independent from the transistor size;
- Short circuit power (Figure 4.2), consumed power during the gate switching as, during such event, the power source is linked with the ground. In the formula, the consumption is greater than the real one because we have used a constant current and equal to I_{peak} in the whole period $t_{sc} = [t_a, t_c]$ (or $[t_d, t_f]$). This component depends on the temperature, the size and the process technology;
- Leakage power (Figure 4.3), *it is a quantum phenomenon where mobile charge carriers (electrons or holes) tunnel through an insulating region* ([7]). This contribution is independent from the *switching activity* and the frequency as it is always present if the circuit is powered on. This is further divided in three causes: Gate leakage (from gate to source), Drain junction leakage (into junctions) and subthreshold current (from drain to source).

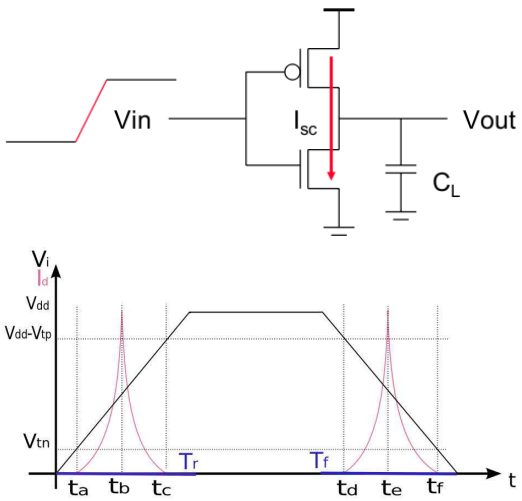

Figure 4.2: Short circuit component of the power consumption

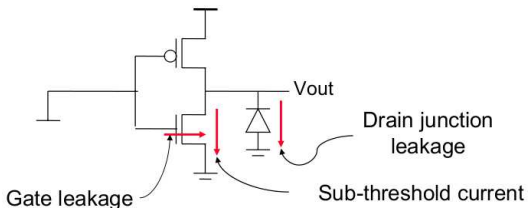

Figure 4.3: Leakage component of the power consumption

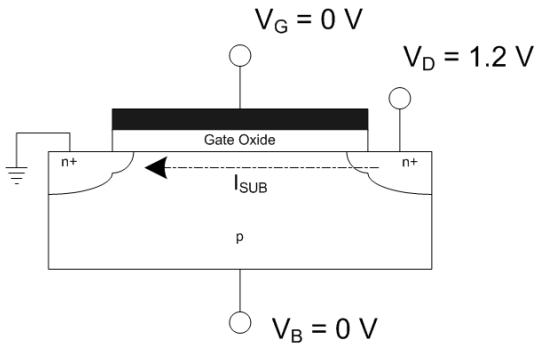

Figure 4.4: Subthreshold leakage

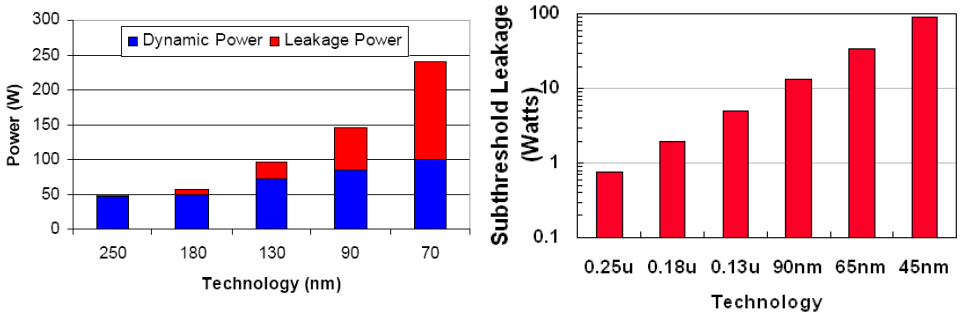

Figure 4.5: Trends of the leakage and subthreshold currents

The most important component of the leakage effect is the subthreshold current, reported in Figure 4.4. Such dissipation is given by the current that flows from the source to the drain of a MOSFET when the transistor is in the subthreshold region (i.e.: $V_{GS} < V_T$).

In Figure 4.5 several results concerning the leakage dissipation and provided by Intel have been reported. The first one highlights that the leakage power has become greater than the dynamic dissipation in recent technologies. The second analysis shows that the subthreshold current according to different technologies (note that the vertical axis has a logarithmic scale).

The parameters that affects this leakage components are the threshold voltage and temperature. On one hand, the dependence with the threshold voltage is direct (Figure 4.6 and, on the other hand, the relation with the temperature is inverse (Figure 4.7). Since the miniaturization of transistors and lower power supply have led to lower threshold voltages, the subthreshold current has been amplified. With a threshold voltage of 0.2V the subthresh-

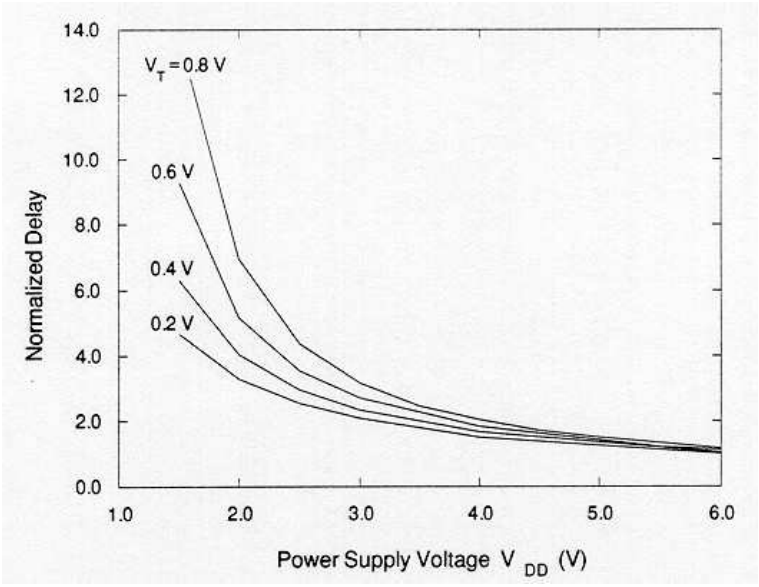

Figure 4.6: Dependency between leakage current and threshold voltage

old power causes more than the 50% of the whole power consumption.

A possible solution consists of increasing V_T for reducing the leakage power, but the main drawback is that in such a way the circuit delay increases as it relays on the following formula

$$\text{Circuit delay} = \frac{V_{DD}}{(V_{DD} - V_T)^2}. \tag{4.4}$$

Concerning electronic design, there are many techniques for reducing the energy consumption, according to when they are applied: design-time or run-time.

- the solutions at design-time are exploited by the electronic designer and are statical:
  - logic design and sizing: reducing the active power by an analysis of the switching activity (the power consumption is data dependent), the glitch effect and the optimal size of the gates;
  - reduced V_{DD} and multi V_{DD}: exploiting the lowest power supply voltage and dividing the circuit in areas (multi paths) with different V_{DD} (which affects the gate delays);
  - multi V_T: for reducing the leakage dissipation by exploiting different V_T for different paths according to their performance;

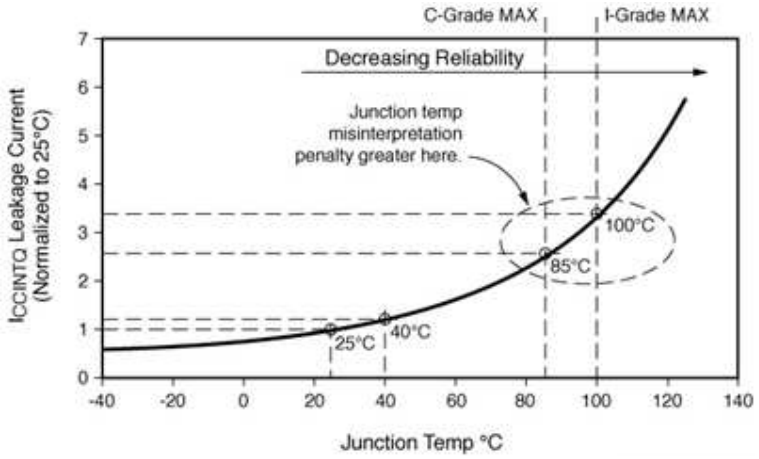

Figure 4.7: Dependency between leakage current and temperature

  - trading off between shared resources (multiplexed) or dedicated resources: shared resources guarantees lower costs and consumes but lower performances (higher access time) than a dedicated approach;

- run-time solutions are used to maximize the energy saving when the system is on:
  - clock gating: disabling the clock signal to reduce dynamic and short circuit powers (especially in pipelined architectures);
  - variable V_{DD} and variable V_T: setting the voltage of a electronic path according to the actual use;
  - DVFS (Dynamic Voltage and Frequency Scaling): varying dynamically the power supply voltage and/or the clock frequency of the system;
  - DPM (Dynamic Power Management): putting the system in to a low power state when possible, postponing task execution.

4.2 State of art

The first contribution of this work, concerning energy saving algorithms for processors, is the following taxonomy whose main features are reported in Figure 4.8 and described as follows:

- information: it specified what kind of data is used

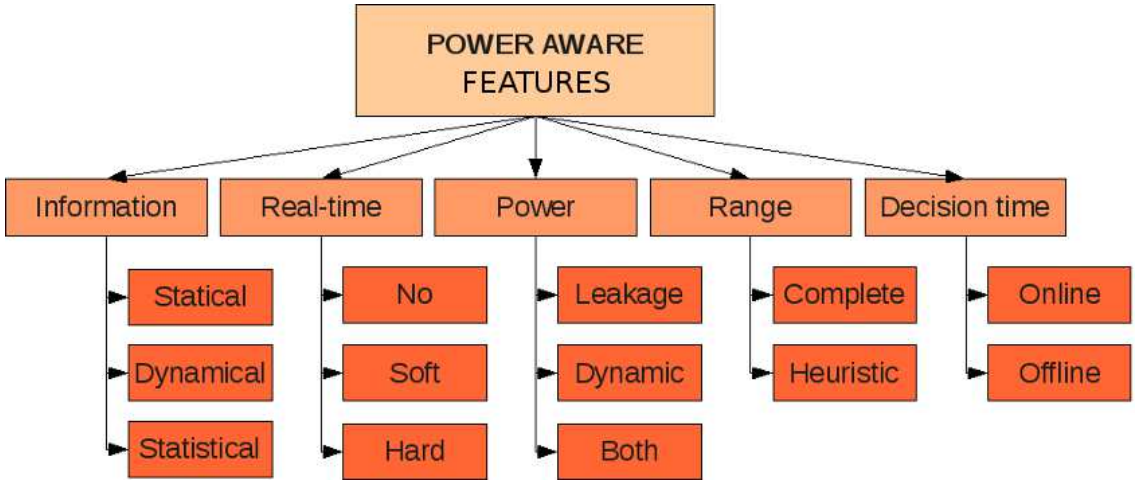

Figure 4.8: Energy saving algorithm taxonomy

  - statical: the algorithm uses only data that are known at design time and does not change in the time (such as WCET, periods and deadlines);
  - dynamical: dynamic data are known only at run-time (such as actual execution time of a job);
  - statistical: the algorithm uses a predictive model to estimate tasks' parameters;

- real-time criticality
  - hard real-time: if the algorithm can be used when a high reliability is required;
  - soft real-time: if the algorithm can be used only when a high reliability is not mandatory;
  - no real-time: if the algorithm does not manage real-time constraints;
  - mixed criticality: a combination of the previous bullets;

- power: the power consumption that the algorithm aims at minimizing
  - dynamic power: this power depends on the voltage and the frequency. The algorithm may implement either a DPM or DVS strategies;
  - leakage power: the algorithm exploits DPM techniques;

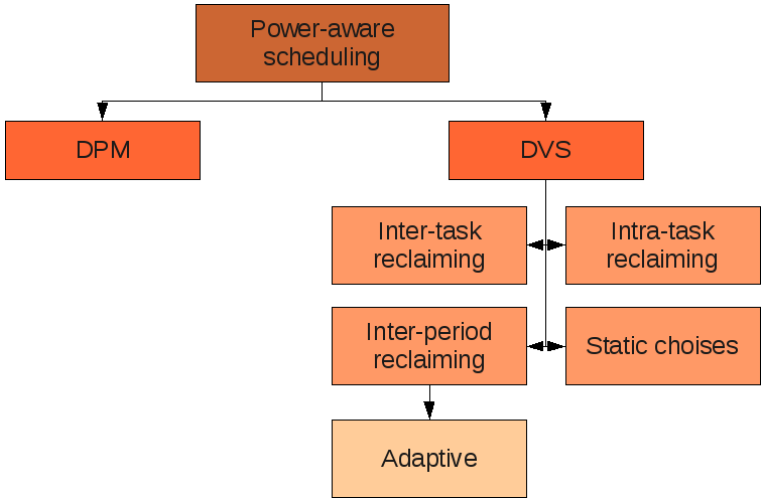

Figure 4.9: Taxonomy of energy aware algorithms

  - both;

- range: the amount of data that the algorithm uses to make chooses
  - complete: if algorithm analyzes all the available information for retrieve the optimal solution. This kind of algorithm may be time consuming;
  - heuristic: the software returns a no optimal solution, taking short time;
- decision time: when the algorithm runs
  - online: the algorithm runs when the system is on;
  - offline:, the algorithm computes a scheduling when the system is not running.

Energy-aware algorithms can be divided according to the criteria reported in Figure 4.9, firstly considering the energy component that those algorithms aim at reducing. In case of DVFS algorithms, they are further divided according to the tasks/jobs involved in the analysis.

The following analysis of the state of art is sorted according to the publication years. All the algorithms are studied for independent tasks (without shared resources).

Task	First a_j	Deadline	Period	WCET	Execution times	Color
τ_1	0	10	10	5	4, 3, 5	yellow
τ_2	0	15	30	7	6	orange

Table 4.1: Task set used for the example

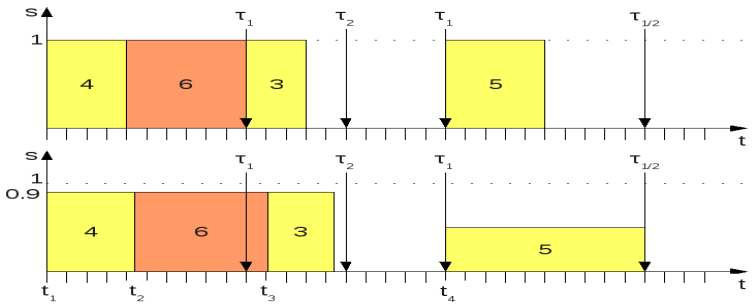

Figure 4.10: Schedules using and not using AVR

One of the first paper was published in the 1995 by Yao F. et at. [9]. The paper proposed three algorithms: an offline mathematical model, AVR (Average Rate) and OA (Optimal Available). AVR exploits the sum of densities of the active tasks in every instant, while OA recomputes the optimal schedule using the newly arrived tasks and the remaining workload. The density of a task j is defined as:

$$d_j(t) = \begin{cases} \frac{R_j}{d_j - a_j} & \text{if } j \text{ is active at } t \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Where R_j, d_j and a_j are the task's workload, deadline and arrival time of the task j, respectively.

Table 4.1 shows the task set used in the following example. The first graph in Figure 4.10 shows the execution without using AVR, as done in the second schedule. The algorithm is dynamical, heuristic and online for hard real-time systems, focusing on dynamic power However, such algorithm does not manage overheads (such as preemption costs and speed scaling), assuming an infinite frequency range.

The computation events are:

- t_1, t_2 and t_3: set $\frac{6}{15} + \frac{5}{10} = 0.9$ as running speed;
- t_4: set $\frac{5}{10} = 0.5$ as speed.

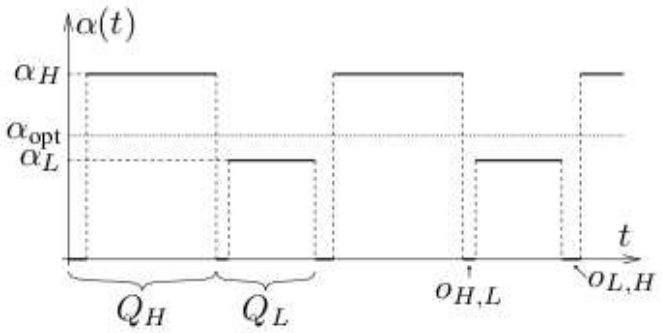

Figure 4.11: Speed modulation example

A similar approach has been proposed by Bini et al. [10]. The proposed algorithm computes an optimal frequency by the offline feasibility analysis, considering the worst-case parameters. Next step choses two speeds (among the supported ones by the processor) and executes a speed modulation between them in order to obtain a mean speed equal to the optimal one, as reported in Figure 4.11. α_L and α_H are chosen to reduce both energy and overhead $o_{L,H}$ and $o_{H,L}$. This algorithm aims at reducing the dynamic power and is a statical, complete and offline for hard real-time tasks.

As has been shown, the target of this kind of algorithms consists of selecting the most appropriate frequency according to the worst case execution. In fact, task early terminations may be useful for further reducing energy consumption.

In 1998, Okuma et al. [11] supposed that to minimize the power consumption, tasks have to finish their executions at the deadline time. Three theorems come from the previous assumption:

- with a continuous voltage range $[V_{min}, V_{max}]$, the voltage value that minimizes the consumption is only one, V_{good};
- with a discrete voltage range, the voltage values that minimize the consumption are two and they are the adjacent of V_{good}: $V_{adj\_min} < V_{good} < V_{adj\_max}$;
- the power consumption decreases if the number of available frequencies increase.

Task	Arrival time	Deadline=Period	WCET	Color
τ_1	4	10	5	yellow
τ_2	6	15	7	orange
τ_3	9	20	10	green

Table 4.2: Task set parameters

These statements can be considered the formal starting point of the DVFS algorithms.

Next algorithms taken into account exploit a different approach which relies on scaling the speed according to the actual remaining workload. In such a case, if a task instance ends its execution earlier than the worst case, next tasks will have an additional time to execute and the system may further slow the frequency down. This kind of algorithms exploit inter-task reclaiming strategies as the (positive or negative) extra budget is given to the other active jobs.

In the 2000 and 2001, Swaminathan and Chakrabarty proposed L-EDF [12] (Low-energy EDF) and EL-EDF [13] (Extended Low-energy EDF). The latter is an improvement of L-EDF, introducing the management of temporal and energy overheads due to frequency scaling. The algorithms assumes that the system has a discrete speed/voltage range, and when a task become RUNNING, the framework checks whether the task meets the deadline with the lowest frequency. If so, the speed scaled, otherwise the CPU runs at the fastest speed. These two algorithms are dynamical, heuristic and online for soft real-time systems (no global feasibility condition is checked), aiming at reducing the dynamic power component. Let us consider the example reported in Figure 4.12, representing an example of L-EDF schedule, with the task set reported in Table 4.2 and speed set $\{0.5, 1\}$.

The computation events are the followings:

- t_1: with speed 0.5, the first task meets its deadline and so, a speed scaling happens;
- t_2: the second task cannot meet the deadline at speed 0.5, so the highest speed is used;
- t_3: the highest speed is used;
- t_4: deadline miss as the previous speed downscaling has delayed excessively.

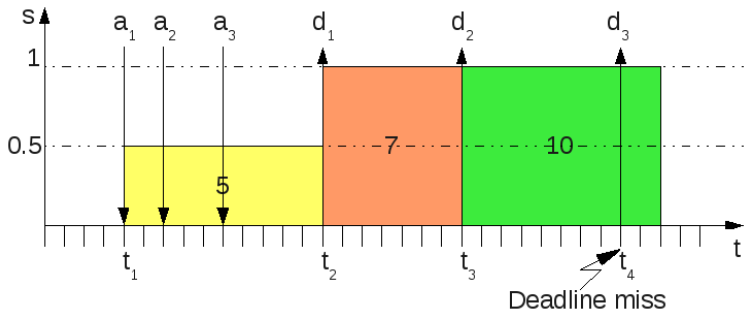

Figure 4.12: L-EDF schedule

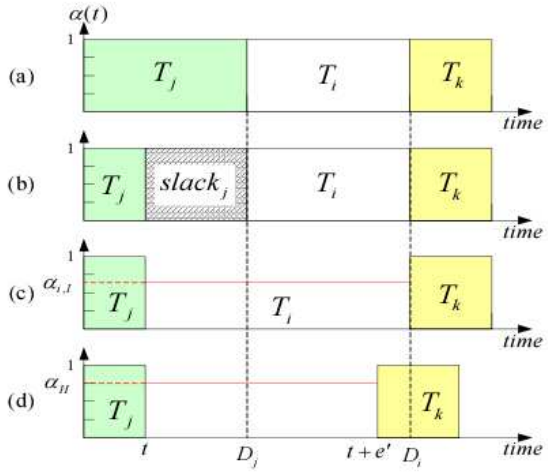

Figure 4.13:

In the 2007, Gong at al. proposed OLDVS and OLDVS* (On-Line Dynamic Voltage Scaling) [6]. When a task ends before its worst case execution time, the slack is given to the next job which can use such additional budget for further reducing the speed. The schedule reported in Figure 4.13 shows an example. The first part represents the worst case execution, without providing additional slack. In the second block, T_j finishes earlier its computation but the slack time is not used. The third example shows that T_i exploits the additional slack to slow the CPU down, terminating the task execution always at D_i. The last example shows that also T_i can finish early and T_k would have a bonus (from T_i and the remaining from T_j) but the example does not represent this case.

Task	Arrival time	Deadline=Period	WCET	Execution times	Color
τ_1	0	10	5	3, 2, 4	yellow
τ_2	0	15	6	4, 6	orange

Table 4.3: Task set parameter

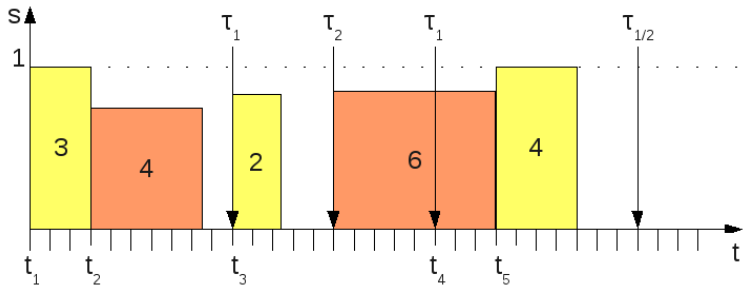

Figure 4.14: Example schedule considering OLDVS

The main difference between the previous algorithms is that OLDVS considers the worst case execution times and not deadlines, meaning that the algorithm is dynamical, heuristic and online, useful in soft real-time environment (as it does not consider switching overhead) and oriented to reduce the dynamic power.

An example is reported in Figure 4.14, considering the parameter in Table 4.3.

The scheduler behavior is the following:

- at t_1: since no tasks are executing, there is no extra slack time. So, the first task starts at the nominal speed;
- at t_2: τ_1 ends after 3 time units instead of 5. τ_2 exploits $8 = 6 + (5 - 3)$ time units for computing 6 work units. The speed is scaled down to $6/8 = 0.75$;
- at t_3: τ_1 ends and the bonus will expire at 11. So, τ_3, when starts, adds a bonus of 1 time unit. It runs at speed $5/6 = 0.83$;
- at t_4: τ_1 gets a bonus of 1 and scaled up to 0.86;
- at t_5: since τ_2 has used the whole available execution time, τ_1 executes at the nominal speed.

For the sake of completeness, the pseudo-code of OLDVS has been reported in the following block.

```
(2)  begin
(3)    a_{i,j} = calculate(t);
(4)                                   Upon context switch to each task T_i at time t
(5)    a_i = SelectFrequency(a_{i,j});
(6)    SetFrequency(a_i);
(7)    proc calculate(t)                                  (r_i = s_i = t and d_i > d_j)
(8)      if T_i preempted T_j
(9)        D_i = t + C_i;
(10)       R_i = C_i;
(11)       R_j = R_j − a_i ∗ (t − l);        l : the previous context switch time
(12)   else if T_i resumes after some task T_k
(13)         D_i = D_i − (D_k − t_p);                    T_i was preempted at t_p
(14)     else                                 T_i starts execution after some task T_k
(15)         if d_k > d_i or D_k < t
(16)           D_i = t + C_i;
(17)       else
(18)           D_i = D_k + C_i;
(19)       fi
(20)       R_i = C_i;
(21)     fi
(22)     return R_i/(D_i − t);                          return the scaling factor
(23)     .
(24)     proc SelectFrequency(a)
(25)       return min(f_1/f_max, ..., f_max/f_max || f_i/f_max >= a);
(26)     .
(27)     proc SetFrequency(a_i)
(28)       f_i = a_i ∗ f_max;
(29)     .
```

The code at lines 12 and 14 (colored in red) contains two corrected mistakes.

So far, OLDVS has been considered. OLDVS* is an improvement which splits the computation time (WCET and bonus time) in two parts, executed at two different speeds. Those two speeds are the adjacent of the on chosen by *OLDVS*. An example is reported in Figure 4.15.

The first segment runs at the lower speed, during which probability that the task terminates early is high. The second part concludes the remaining work at the higher speed. The OLDVS* pseudo-code is reported is the following.

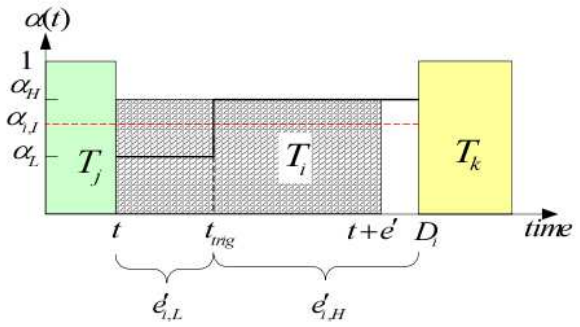

Figure 4.15: Example of OLDVS* execution

```
(2)  begin
(3)    a_{i,I} = calculate(t);      Upon context switch to each task T_i at time t
(4)    a_H = SelectFrequency(i,I);          Select frequency levels a_H and a_L
(5)    Select a_L;                                                  a_L < a_{i,I}
(6)    t_trig = t + (D_i − t) * (a_H − a_{i,I})/(a_H − a_L);
(7)    SetFrequency(a_L);
(8)    SetTimer(t_trig);
(9)    proc SetTimer(t)
(10)     Set Timer to be triggered at t;
(11)   .
(12)   proc OnTrigger()
(13)     R_i = R_i − a_L * (t − l);
(14)     SetFrequency(a_H);
(15)   .
```

The previous example, used to show the OLDVS behavior, is now tested with OLDVS* in Figure 4.16.

The scheduling events are the followings:

- at t_1: there is no bonus, so the task is executed at the maximum speed;
- at t_2: τ_2 inherits 2 time units as bonus. OLDVS returns the speed 0.75, so OLDVS* choses 0.6 and 1. The periods are computed with respect the chosen speeds;
- at t_3: the bonus is 1, so the algorithm exploits the speeds 0.7 and 1 (instead of 0.83);
- at t_4: OLDVS returns 0.85;

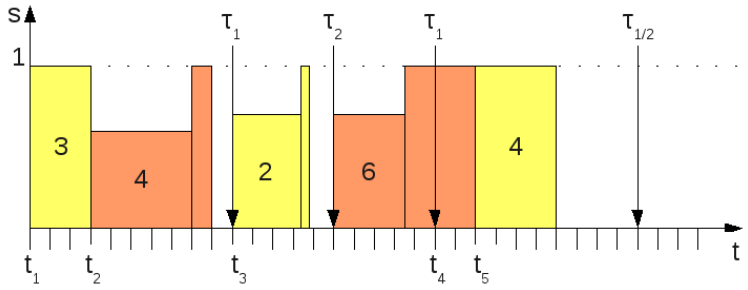

Figure 4.16: Example of OLDVS* execution

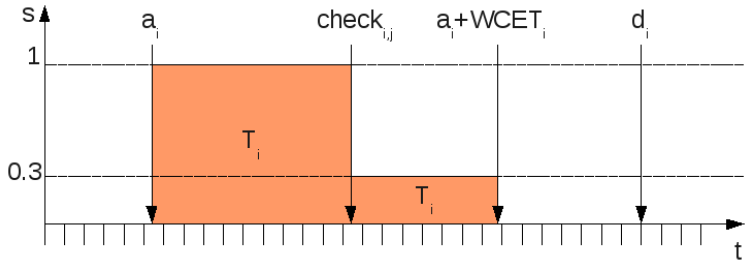

Figure 4.17: Example of DRA execution

- at t_5: there is no bonus, so τ_1 runs at the maximum speed.

Bini and Scordino [14] proposed an optimal algorithm to find the two frequency and managing the overheads to work with *hard real-time* systems.

OLDVS and OLDVS* are inter-task reclaiming algorithms as the extra slack time is exploited by the job in execution. On the other hand, intra-task reclaiming solutions consumes the slack time within the task itself. Aydin et al. proposed DRA [15] (Dynamic Reclaiming Algorithm) which assumes that when a task discovers that the actual job will finish earlier than the worst case, it slows the CPU down, prolonging the execution until the worst case finishing time.

Such algorithm is dynamical, heuristic and online for hard real-time systems coping with dynamic power. In the example reported in Figure 4.17, the task T_i executes a check during its execution ($check_{i,j}$), discovering that it will terminate earlier. Then, it computes the necessary speed to end earlier the worst case limit, in the example such speed is 0.3, and makes such frequency operative.

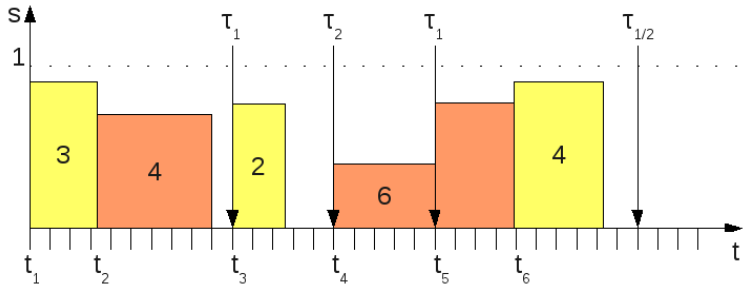

Figure 4.18: Example of ccDVS execution

A well-know class of solutions recomputes the optimal frequency when the ready queue change. Such algorithms are not labeled with an unique name and are referred as Intra-periods reclaiming policies. Basically, they exploited the slack time considering the whole task set, even next arrival tasks.

In 2001, Pillai et al. proposed ccDVS (Cycle Conserving DVS) [16]. At every task arrival or termination, the algorithm calculates the lowest speed that satisfies the requirements according to the remaining workload. The algorithm is dynamical, heuristic and online for soft real-time systems, focusing on dynamic power.

Under the EDF scheduler, the feasibility test is (assuming $T_i = D_i$):

$$\sum_{i=1}^{\#active\_tasks} \frac{C_i(s)}{T_i} \leq 1. \tag{4.6}$$

The algorithm uses as computational time C_i the worst case (if the task has not executed yet) or the execution time of the last instance (if the task has already finished).

The feasibility condition can be rewritten as follow:

$$\frac{f}{f_{max}} = \sum_{i=1}^{\#active\_tasks} \frac{\#INSTRUCTIONS_i}{T_i} \tag{4.7}$$

Considering the same task set of the previous example, Figure 4.18 reports an example of ccDVS execution.

The scheduling events are the followings:

- at t_1: all the tasks are active, the normalized speed is $\frac{5}{10}+\frac{6}{15} = \frac{27}{30} = 0.9$;

- at t_2: τ_1 ends after 3 time units instead of 5. Only τ_2 is active. The speed is set to $\frac{3}{10} + \frac{6}{15} = 0.7$;
- at t_3: a new job of τ_1 arrives, setting the speed at $\frac{5}{10} + \frac{4}{15} = 0.77$;
- at t_4: a new job of τ_2 arrives, setting the speed at $\frac{2}{10} + \frac{6}{15} = 0.6$;
- at t_5: a new job of τ_1 arrives, setting the speed at $\frac{5}{10} + \frac{4}{15} = 0.77$;
- at t_6: τ_1 ends, adjusting the speed at $\frac{5}{10} + \frac{6}{15} = 0.9$.

The pseudo-code is reported in the following block.

```
(1)  begin
(2)    proc select_frequency()
(3)      return min (f_1/f_max, ..., f_max/f_max || f_i/f_max >= U);
(4)    .
(5)    proc upto_task_release(T_i)
(6)      U_i = WCET_i/T_i;
(7)      select_frequency();
(8)    .
(9)    proc task_completition(T_i)
(10)     U_i = previous_execution_time_i/T_i;
(11)     select_frequency();
(12)   .
```

Another example of this kind of algorithms is Look-Ahead RT-DVS which aims at exploiting as long as possible the low speeds, until the waiting jobs are manageable considering their deadlines. The algorithm is dynamical, heuristic and online for soft real-time systems, minimizing the dynamic power.

The algorithm pseudo-code is reported in the following.

```
(1)  begin proc select_frequency(x)
(2)          return min(f_1/f_max, ..., f_max/f_max || f_i/f_max >= x);
(3)    .
(4)    proc upon_task_release(T_i)
(5)      left_i = C_i;
(6)      defer();
(7)    .
(8)    proc task_completion(T_i)
(9)      left_i = 0;
(10)     defer();
(11)   .
```

```
(12)  proc during_execution(T_i)
(13)    decrement left_i;
(14)  .
(15)  proc defer()
(16)    U = C_1/T_1 + C_2/T_2 + ... + C_n/T_n;
(17)    s = 0;
(18)    for i = 1 to n                    Reverse EDF order of the tasks
(19)        U = U - C_i/T_i;
(20)        x = max(0, left_i - (1 - U)/(d_i - d_n));
(21)        U = U + (left_i - x)/(d_i - d_n));
(22)        s = s + x;
(23)    end
(24)    select_frequency(s/(d_n - current_time));
(25)  .
```

The algorithm is explained thanks to the example reported in Figure 4.19. In each schedule, the highlighted time instant is the computation point in question. The first task has a worst case execution time of 2, an actual execution time of 1 and a period of 10 time units. The second task has the worst-case and actual execution equals to 4 and period of 13.

The scheduling events are the followings:

- at t_1: τ_1 and τ_2 are active. Next deadline belong to τ_1l, so it allocates the CPU bandwidth:
  - the algorithm reserves resources for next τ_1 job;
  - the algorithm reserves resources for next jobs of the other tasks (not the actual instances);
  - for the active tasks, the algorithm aims at spreading execution times (using the actual remaining computation execution times) from next deadline until next one;
- at t_2: τ_1 ends earlier that the worst case. Since next deadline belongs to τ_1, result does not change;
- at t_3: a new job of τ_1 arrives, but next deadline belongs to τ_2. So, it allocates the next execution in the next period, allocates other next instances and spreads the worst case execution times of actual instances.

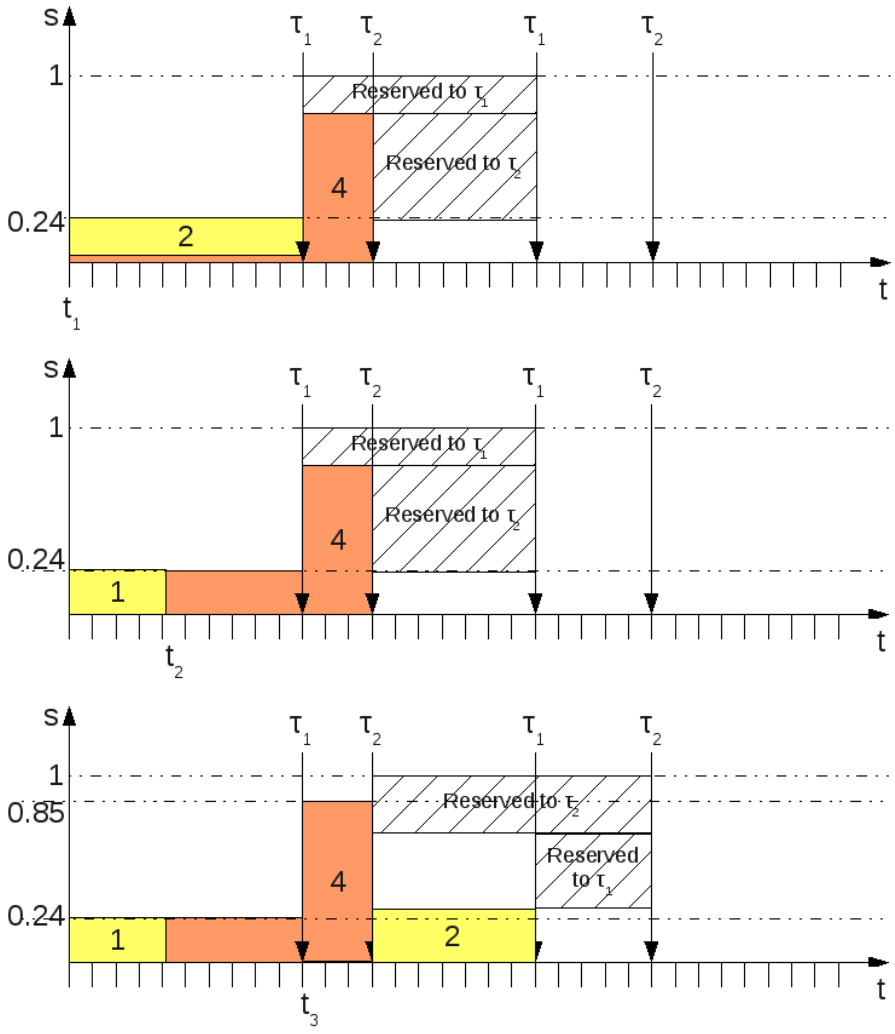

Figure 4.19: Example of Look-Ahead RT-DVS execution

Another kind of algorithms aiming at reducing the dynamic power are the adaptive ones which improve the previous algorithms by exploiting dynamic parameters. One of the most interesting example was published by Aydin et al., AGR [15] (AGggressive speed Reduction). The proposed algorithm uses as execution times the execution time of the previous task instance.

The mechanism may be improved by adding a PID estimator of the computation times [17] or exploiting a predictive module based on a statistical task model (especially in event-driven system) [18].

Concerning the algorithms ([18–20]) that are aimed at reducing the leakage power consumption, there are becoming really popular in the community as in almost all the actual hardware the static dissipation dominate dynamic power consumption.

All the presented algorithms assume an independent task model, without any shared resources, as it is a new research topic [21].

4.2.1 Temperature management

A new research topic related to the energy minimization concerns temperature management. Instead of minimizing the average power consumption, it aims at avoiding power dissipation peaks that may significantly increase the system temperature. Moreover, as has been discussed into the chapter introduction, the subthreshold current is significantly affected by the temperature and, in addition, it may reduce significantly the processor life-time causing electro migration phenomena.

More precisely, the temperature management problem can be formulated as follow:

- target function: ensuring real-time constraints;
- cost function: minimizing power density (the generated temperature is proportional).

This problem is different with respect to the power management issue as in several cases, executing at high speed for short time can be more suitable for the energy, but it would increase significantly the temperature.

Nowadays, almost all the advanced processors are equipped with an internal thermal sensor (or more than one), to acquire data useful for the thermal management policy. On the other hand, in the simulation, the mathematical formulation that represents the heat conduction is given by the Fourier's law.

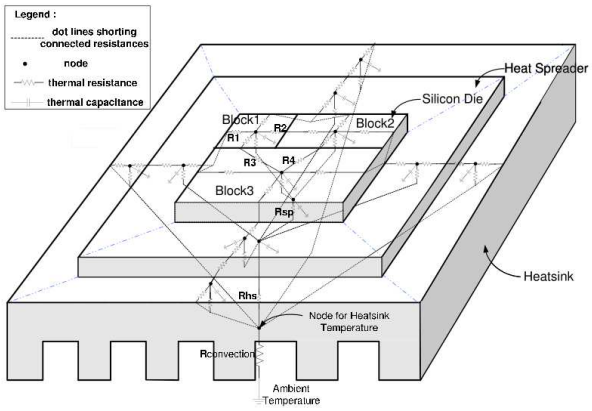

Figure 4.20: Example of the Hotspot library

The most important work, concerning simulations and heat transmission, is the *Hotspot* library [22, 23] which lets users model the processor's heat conduction model similarly to an electric circuit, as reported in Figure 4.20.

A classification useful to collect thermal aware algorithm is the followings:

- reactive: if a thermal problem happens (e.g. overheating), the algorithm tries to solve it;
- proactive: the algorithm aims at avoiding thermal problems. The algorithm is executed not only when the problem happens;
- both: the algorithm combines the previous approaches together.

The first option is more dangerous and less deterministic than the second. An algorithm that follows such approach is [24] which computes two CPU speeds (they must satisfy time requirements) and switches between them according to the temperature. Lower speeds produce less heat, letting the processor cool down.

Concerning the second approach, the algorithm proposed in [25] implements and solves a MILP model (Mixed Integer Linear Programming) model.

Thermal management in multi-core systems is still an open problem and it will keep the researchers' minds in gear for the next years. On of the few papers that have already faced the problem on CMP (Chip Multi Processor), are [26] and [27] which exploit thread migrations in order to reduce power density on the various cores.

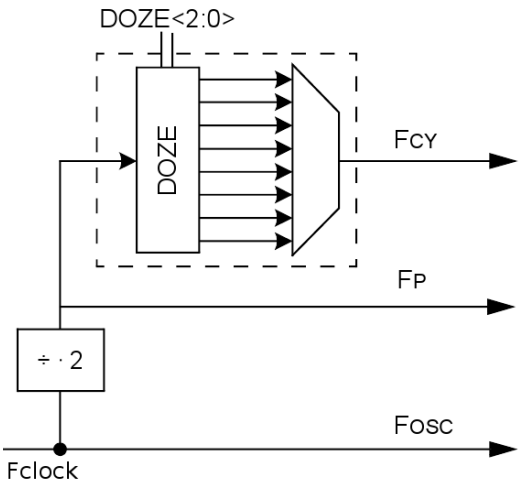

Figure 4.21: DOZE circuit

4.3 Implementation

Three policies have been implemented within the proposed framework *OLDVS*, *BSdvs* and *BSSdvs*.

4.3.1 OLDVS

OLDVS (On-Line Dynamic Voltage Scaling) is the algorithm proposed in [6], already detailed in Section 4.2, while BSdvs and BSSdvs are analyzed in Section 4.3.2 and Section 4.3.3, respectively.

Concerning low-power states, *EXPS* (Extreme Power Saving Mode) is a feature of the module that switches the CPU in a power saving mode when the user wants. Generally, microcontrollers have two or more special states, idle and sleep. The first state disable the clock signal for some parts of the microprocessor suspending the code execution. The second is deeper and switches off almost all the components, including oscillators. The wake-up time is proportional to the amount of saved energy, lower consumptions mean longer waking-up time. The dsPic® family has an additional hardware that allows to slow down (not stopping) the instruction clock (nor the device clock). Such a feature is called DOZE (shown in Figure 4.21 and is useful to avoid the adjusting of the device's parameters when the user needs to change only the code speed. The overhead is negligible, as only the clock pre-scaler is adjusted. The EXPS module may exploit both the idle state and doze, but the used feature is chosen at compile-time.

4.3.2 BSdvs

BSdvs stands for Budget Sharing Dynamic Voltage Scaling, which is an inter-task reclaiming algorithm. Basically, it is an improvement of OLDVS [6] for hard real-time environments considering switching overhead. The pseudo-code is the following.

```
(1)  begin proc system_starts()
(2)          bonus = 0;
(3)          switch to f_nominal
(4)    .
(5)    proc new_task(T_i)
(6)      left_R_i = WCIN_i;              //Worst Case Instruction Number;
(7)      left_T_i = WCET_i;      //Worst Case Execution Time at f_nominal;
(8)    .
(9)    proc task_ends(T_i)
(10)     if left_T_i > 0 then
(11)                          bonus = left_T_i;
(12)                   else
(13)                          bonus = 0;
(14)     fi
(15)     if T_i was the last task then
(16)                                  switch to f_nominal
(17)     fi
(18)     reset_timer
(19)   .
(20)   proc task_starts(T_i)
(21)     stop_timer
(22)     bonus = bonus − (timer_value at f_actual);
(23)     if bonus < 0 then                          //Is the bonus expired?
(24)                          bonus = 0;
(25)     fi
(26)     for j = 1 to number_of_speeds    //From the slowest to the fastest
(27)         if ((left_R_i at f_j)+
(28)           overheads_to_j + overheads_to_nominal)
(29)           < left_T_i + bonus then
(30)                                  switch to f_j
(31)                                  left_T_i = left_T_i + bonus;
(32)                                  bonus = 0;
(33)                                  break;
(34)         fi
(35)     end
```

```
(36)    reset_timer
(37)  .
(38)  proc task_stops(T_i)
(39)    stop_timer
(40)    left_R_i = left_R_i − timer_value;
(41)    left_T_i = left_T_i−
(42)    (timer_value at f_actual);
(43)    clear_timer
(44)  .
```

The speed $f_{nominal}$ is the frequency for which the schedule is feasible. The variable *bonus* contains the remaining time of the previous task due to early termination that it has not been used with respect the worst case execution time. The algorithm behavior can be summarized as follows:

- when the system starts for the first time, *system_starts* initializes the variables and scales to the nominal speed;
- *new_task* is called when a new instance of a task is created and it initializes the task's variables to their initial values:
  - $left\_R_i$: contains the remaining number of instructions than the worst case;
  - $left\_T_i$ contains the remaining execution time than the worst case;
- when a task becomes *RUNNING*, *task_starts* checks if the previous task have left a bonus execution time and if so, the actual task will use it (if it has not expired yet). Afterwards, it checks (from the slowest available speed to the fastest), if the task can execute its remaining load at that frequency considering the remaining execution time ($left\_T_i$) and bonus. If possible, the following components are also considered:
  - the switching time from actual speed to the new one;
  - the switching time from the new one to the nominal as the next task may require to speed up and the relative overhead must be considered taken into account;
- when a preemption happens, a task becomes *WAITING* or before a task instance's end, *task_stops* subtracts from $left\_R_i$ the elapsed instructions and from $left\_T_i$ the elapsed time.

- when an instance ends its execution, *task_ends* updates the *bonus* variable with the remaining execution time that it has not exploited. Moreover, if there are no tasks in the *READY* queue, restores the nominal speed;

Instructions at lines 19 and 33 allow to work with hard real-time tasks as they manage overhead and change the frequency if and only if such operation does not affect the execution at the nominal speed.

4.3.3 BSSdvs

BSSdvs (Budget Sharing and Splitting Dynamic Voltage Scaling) is equivalent to OLDVS* [6] with the difference that BSSdvs is suitable for hard real-time systems. In the following box, the pseudo-code has been reported.

```
(1)  begin proc system_starts()
(2)          bonus = 0;
(3)          switch to f_nominal
(4)    .
(5)    proc new_task(T_i)
(6)      left_R_i = WCIN_i;              //Worst Case Instruction Number;
(7)      left_T_i = WCET_i;      //Worst Case Execution Time at f_nominal;
(8)    .
(9)    proc task_ends(T_i)
(10)     if left_T_i > 0 then
(11)                          bonus = left_T_i;
(12)                   else
(13)                          bonus = 0;
(14)     fi
(15)     if T_i was the last task then
(16)                                  switch to f_nominal
(17)     fi
(18)     reset_timer
(19)   .
(20)   proc task_starts(T_i)
(21)     stop_timer
(22)     bonus = bonus − (timer_value at f_actual);
(23)     if bonus < 0 then                          //Is the bonus expired?
(24)                          bonus = 0;
(25)     fi
(26)     for j = 1 to number_of_speeds    //From the slowest to the fastest
(27)         if ((left_R_i at f_j)+
```

```
(28)            overheads_to_j + overheads_to_nominal)
(29)            < left_T_i + bonus
(30)            then
(31)                  left_T_i = left_T_i + bonus;
(32)                  bonus = 0;
(33)                  tmp = left_R_i * (f_j+1 − f_j)
(34)                  /(f_j+1 − f_j-1);
(35)                  t_low = tmp/f_j-1;
(36)                  t_high = (left_R_j − tmp)/
(37)                  f_j+1;
(38)                  if power_j * left_T_i >
(39)                    (t_low * power_j-1 + t_high*
(40)                    power_j+1)&&
(41)                    (t_low + t_high + overheads
(42)                    _to_j-1 + overheads_to_j+1+
(43)                    overheads_to_nominal)
(44)                    < left_T_i&&j < number_of_speeds then
(45)                                                    switch to f_j-1
(46)                                                    set next event after t_low
(47)                                                  else
(48)                                                    switch to f_j
(49)                  fi
(50)          fi
(51)      end
(52)      reset_timer
(53)  .
(54)  proc task_stops(T_i)
(55)      stop_timer
(56)      left_R_i = left_R_i − timer_value;
(57)      left_T_i = left_T_i−
(58)      (timer_value at f_actual);
(59)      clear_timer
(60)  .
(62)  proc timer_event(T_i, f_high)
(63)      stop_timer
(64)      left_R_i = left_R_i − timer_value;
(65)      left_T_i = left_T_i−
(66)      (timer_value at f_actual);
(67)      switch to f_high
(68)      reset_timer
(69)  .
```

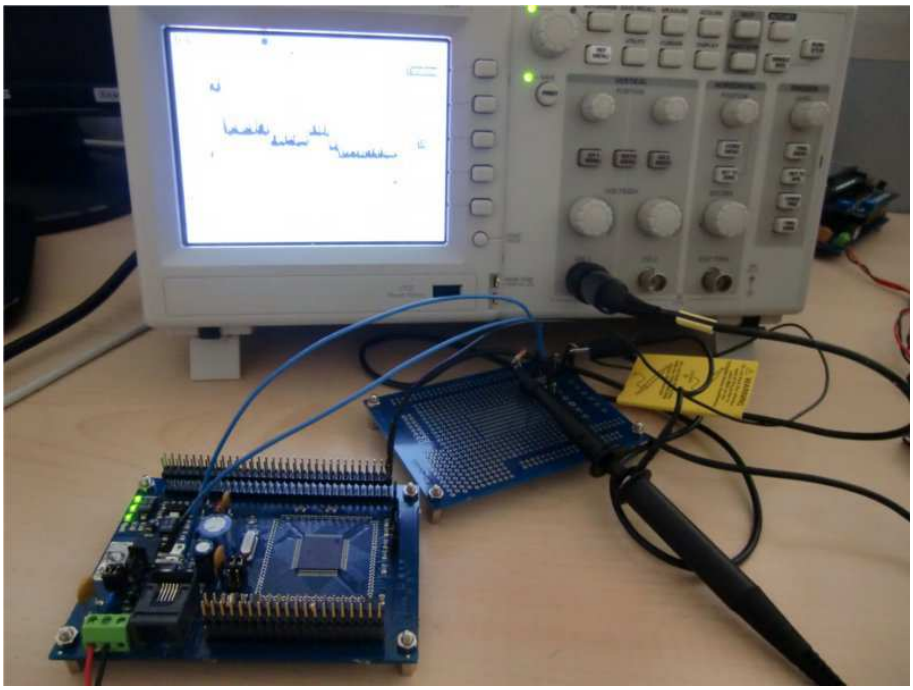

Figure 4.22: Energy consumption measurement system

In the code, there are two differences with respect to BSdvs. The first one concerns the function *task_starts* at line 40 as the algorithm checks if the splitting (exploiting the two close speeds) satisfies time requirements (time executions, switchings to the two frequencies and then to the nominal speed) and the energy consumption is lower than executing using a single speed. The second difference is how the the middle event is handled to switch from the lower speed to the higher. Such operation is done by *timer_event*.

4.4 Results

All the measurements have been carried on with the system in Figure 4.22.

The output and ground pins of the voltage regulator (7805) of the Flex Base board were unsoldered and connected to the current-sense amplifier. Consumption values were read by an oscilloscope and an additional Flex board (sampling with an ADC at 1 Million samples per second).

The first important measurement concerns the comparison between the measured board consumption (at $24C$) and the values in the datasheet (assuming $25C$), with respect different operative modes. Notice that the mea-

State	From datasheet Max (mA)	Typ (mA)	Measured (mA)
40 MIPS	90.00	84.00	86.12
35 MIPS	——	——	79.67
30 MIPS	70.00	65.00	73.00
20 MIPS	50.00	46.00	60.14
16 MIPS	40.00	37.00	54.46
10 MIPS	30.00	27.00	46.41
8 MIPS	——	——	43.20
2 MIPS	——	——	34.79
Doze at 40 MIPS	30.00	25.00	60.63
Doze at 2 MIPS	——	——	33.38
IDLE	25.00	16.00	56.38
SLEEP	0.50	0.21	26.40

Table 4.4: CPU consumption comparisons

sured data refers to the whole board and the data in the datasheet only to the core. The CPU has been configured as reported in the datasheet, so:

- external clock in use on OSC1 pin;
- all pins configured as input and pulled to V_{SS} (when possible);
- all peripherals were off (but clocked);
- program and data memories were on;
- the code was an infinite loop (without conditions and memory accesses);
- WDT (Watchdog) and FSCM (to change the frequency) were disabled.

The consumption comparisons are reported in Table 4.4 and then plotted in Figure 4.23.

The power model confirms the equation proposed in [8] (without voltage scaling and non-linearity components). The final model concerning the measured consumptions is summarized in Equation 4.4, where f is expressed in MIPS (Million Instructions per second) and A (mA).

$$A(f) = 1.35 \times f + 32.12$$

Since the formula refers the absorbed current, for computing the required power the value must be multiplied by the voltage ($5V$). Different slopes

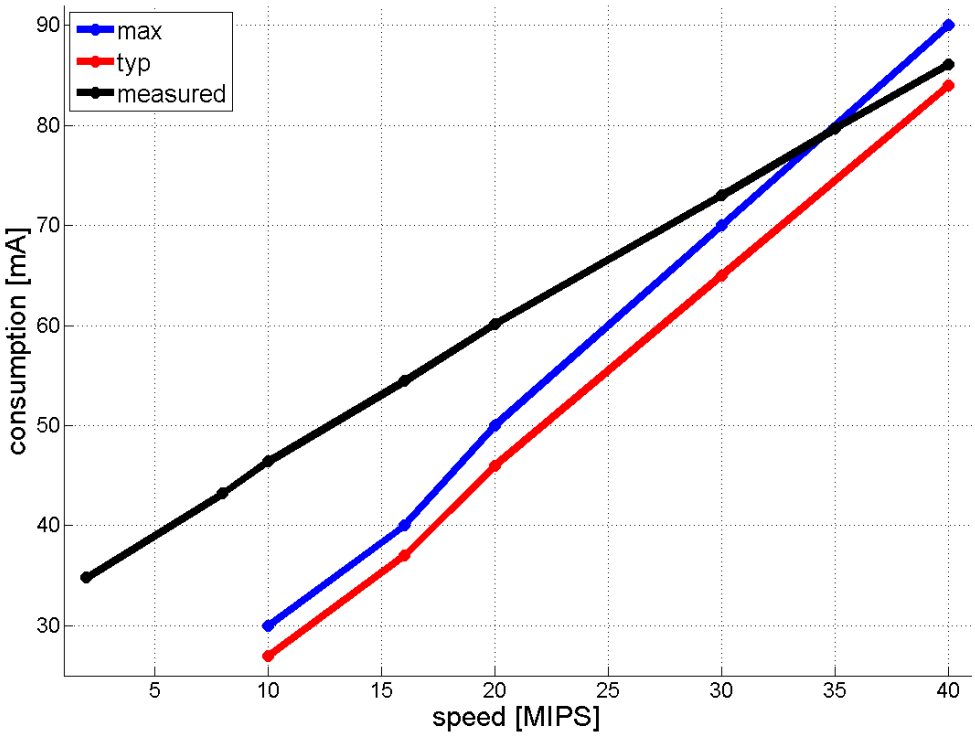

Figure 4.23: CPU consumption comparisons

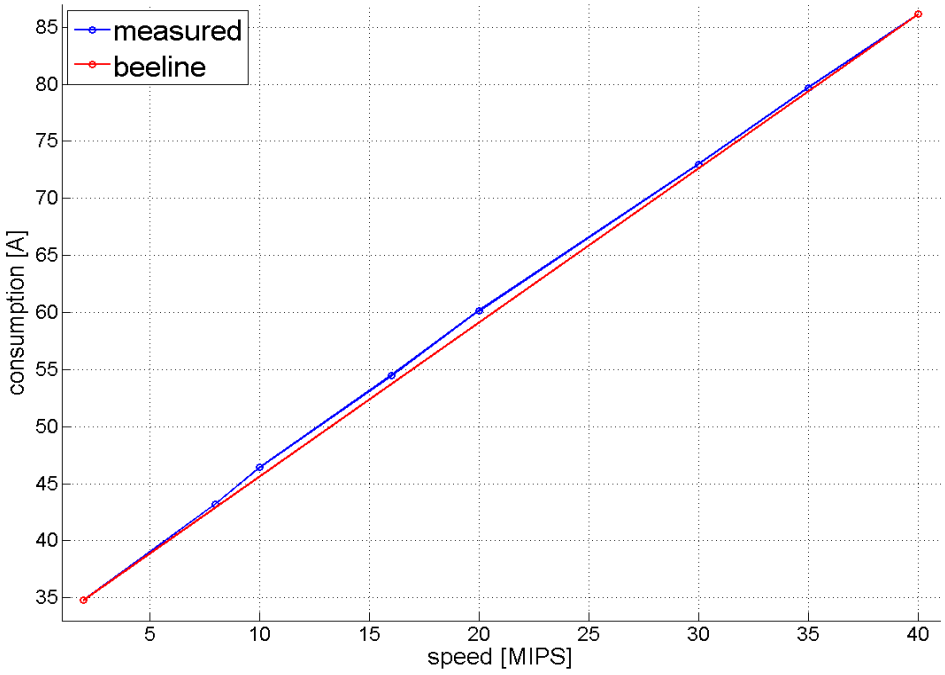

Figure 4.24: Comparison between measured consumption and approximation function

between the plotted functions might be caused by several components (on the board) that change with the frequency. In Figure 4.24, the comparison between the measured consumption and the approximated function in Equation 4.4 has been reported.

In addition, to conclude this brief profile of the board in use, Table 4.5 reports the power consumption of several devices on the board. The first and second entries have been read from the datasheet (typical values) while the last was measured. However, they does not depend on the running frequency.

The CPU driver has been implemented to use all the speeds reported in Table 4.4: 2, 8, 10, 16, 20, 30, 35 and 40 MIPS.

Since we have decided to deal with hard real-time systems, all the temporal overhead has been measured. All the clock configurations use either the crystal (2 MIPS) or the crystal combined with the PLL. The datasheet reports the following formulas of the maximum overheads for the frequency switchings:

$$T_{overhead_{from\_off\_to\_XT\_PLL}} = T_{OSCD} + T_{OST} + T_{LOCK}$$

Device	Consumption (mA)
Voltage regulator	5.00
Primary oscillator	2.20
Two green leds	7.38

Table 4.5: Consumption of several devices on the board

From state	Waking up time	Unit
Sleep	19.30	ms
IDLE	8	instruction cycles
DOZE	4	instruction cycles

Table 4.6: Overhead to recover the system for a low-power state

$$T_{overhead_{from\_off\_to\_XT}} = T_{OSCD} + T_{OST}$$

Where the meanings of the components are the followings (their values are not constants):

- T_{OSCD} (Oscillator start-up Delay) is the time to stabilize the clock signal (it depends on the crystal). It must be considered when the crystal is switched on;
- T_{OST} (Oscillator Start-up Timer delay) is a safety time of 1024 cycles (independent from the running speed);
- T_{LOCK} is the necessary time to stabilize the PLL loop (the nominal value is 1 ms).

As reported in Table 4.6, waking-up the system from the sleep state is time expensive as the procedure lasts for the worst case, taking the whole overhead plus another delay T_{VREG} (the nominal value is 130μs) which is the standby-to-active transition time. IDLE and DOZE imply short overhead as they disable the clock signal only for few modules or postscale the instruction frequency.

In Table 4.7 the overhead for scaling the frequency from all to all possible configurations. Cells contain values in microseconds (μs). The configuration which provides 2 MIPS uses only the primary oscillator and the other settings use the primary oscillator and the PLL. Maximum overheads are obtained when the system switches from the first configuration (2 MIPS) to one of the others as it must activate the PLL and T_{LOCK} is the maximum. The

		TO							
	MIPS	2	8	10	16	20	30	35	40
FROM	2	21.53	1012.05	1011.98	1010.48	1012.35	1010.48	1011.60	1011.53
	8	18.75	7.95	15.23	11.03	11.10	8.93	9.38	9.68
	10	17.70	14.03	6.30	8.93	8.93	7.13	7.43	7.80
	16	16.28	11.20	10.65	8.03	7.95	5.48	5.78	5.78
	20	15.75	10.20	9.08	6.38	3.15	4.28	4.50	4.65
	30	15.30	9.98	9.15	5.93	5.85	1.88	3.98	4.35
	35	14.85	9.38	8.33	5.18	5.18	3.60	1.73	3.75
	40	14.70	8.55	7.73	4.58	4.58	3.23	3.15	1.65

Table 4.7: Speed scaling overhead

opposite cases (to 2 MIPS) are not expensive because they only turn off the PLL. When the changing requires only a variation of the PLL (e.g.: from 10 MIPS to 40 MIPS), the operation takes a time proportional to the speed difference. On the diagonal (from a speed to the same), there are the time to execute the library code at a particular speed (without frequency changing, values are proportional).

To test the system, three task sets have been created, containing 10 tasks each. Table 4.8 reports the utilization, for each test bed, of each task, where $U_{task_i} = WCET_i/T_i$. The last row contains the total utilization factor of the three task sets. The tasks are sorted from the one with highest priority to the lowest. The hyperperiod (least common multiple of the periods) is set as $1s$. For each task set, nine tests have been carried out, and for each of them, the real utilization factor (i.e.: using the real execution time and not the worst case execution time) was scaled from 10% to 90% of the worst case, with steps of 10%. The reduction has been been applied uniformly to all the tasks. Finally, all the task sets are feasible at 40 MIPS with FP and EDF.

In the rest of the chapter, the reported results assume Fixed-Priority as scheduler. To test the module, the board is configured without caring of small power dissipation due to devices and peripherals, such ass in traditional applications.

The first computation concerns the average power **during the executions of the tasks** (not in the whole hyper period). Table 4.9, Table 4.10, Table 4.11, Table 4.12, Table 4.13 and Table 4.14 reports the absolute and relative (ratio between the values and *Nothing*'s one) results.

Figure 4.25, Figure 4.26 and Figure 4.27 plots in a graphical way the percentages in Table 4.12, Table 4.13 and Table 4.14, respectively.

In the previous analysis, the algorithms work without any problem, ob-

	Task set		
Task	**1**	**2**	**3**
1	9.80%	17.82%	19.60%
2	9.80%	16.04%	19.60%
3	9.80%	14.25%	19.60%
4	9.80%	12.47%	5.39%
5	9.80%	10.69%	5.39%
6	9.80%	8.91%	5.39%
7	9.80%	7.13%	5.39%
8	9.80%	5.35%	5.39%
9	9.80%	3.56%	5.39%
10	9.80%	1.78%	5.39%
Total	98.00%	98.00%	96.53%

Table 4.8: Task sets used in the experiments

	U_{real}/U_{worst_case} (W)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.498	0.501	0.501	0.502	0.501	0.502	0.499	0.502	0.502
Oldvs	0.320	0.357	0.375	0.387	0.415	0.443	0.459	0.477	0.485
BSdvs	0.285	0.296	0.315	0.335	0.354	0.378	0.409	0.439	0.451
BSSdvs	0.286	0.298	0.314	0.345	0.356	0.376	0.414	0.432	0.451

Table 4.9: Mean power of the first task set

	U_{real}/U_{worst_case} (W)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.501	0.502	0.501	0.501	0.504	0.503	0.506	0.502	0.502
Oldvs	0.272	0.289	0.336	0.360	0.396	0.409	0.448	0.453	0.482
BSdvs	0.242	0.258	0.283	0.310	0.339	0.365	0.394	0.425	0.446
BSSdvs	0.243	0.260	0.285	0.327	0.347	0.366	0.398	0.422	0.448

Table 4.10: Mean power of the second task set

	U_{real}/U_{worst_case} (W)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.501	0.508	0.508	0.507	0.506	0.509	0.508	0.508	0.509
Oldvs	0.244	0.282	0.324	0.337	0.389	0.432	0.454	0.470	0.486
BSdvs	0.228	0.252	0.281	0.307	0.328	0.360	0.387	0.417	0.447
BSSdvs	0.228	0.253	0.277	0.308	0.329	0.360	0.392	0.414	0.446

Table 4.11: Mean power of the third task set

	U_{real}/U_{worst_case} (%)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Oldvs	64.29	71.33	75.04	77.12	82.69	88.22	91.97	95.18	96.63
BSdvs	57.20	59.07	62.96	66.77	70.70	75.37	82.06	87.49	89.95
BSSdvs	57.44	59.38	62.75	68.63	70.94	74.92	83.00	86.14	89.89

Table 4.12: Percentages of the first task set

	U_{real}/U_{worst_case} (%)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Oldvs	54.40	57.66	67.11	71.90	78.57	81.30	88.64	90.23	96.16
BSdvs	48.40	51.46	56.51	61.90	67.15	72.50	77.84	84.62	89.00
BSSdvs	48.57	51.77	56.98	65.21	68.83	72.80	78.64	83.96	89.33

Table 4.13: Percentages of the second task set

	U_{real}/U_{worst_case} (%)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Oldvs	48.70	55.42	63.79	66.51	76.88	84.97	89.26	92.49	95.48
BSdvs	45.52	49.59	55.32	60.52	64.76	70.83	76.23	82.00	87.77
BSSdvs	45.45	49.80	54.51	60.71	65.01	70.73	77.10	81.50	87.62

Table 4.14: Percentages of the third task set

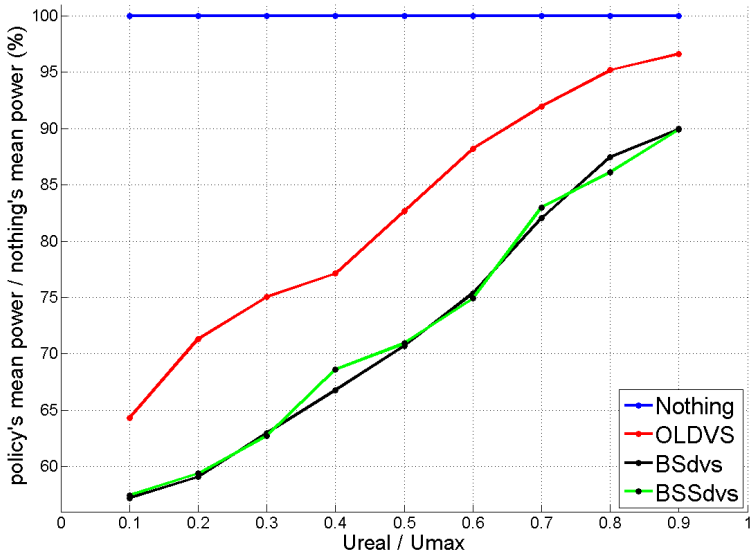

Figure 4.25: Percentages of the first task set

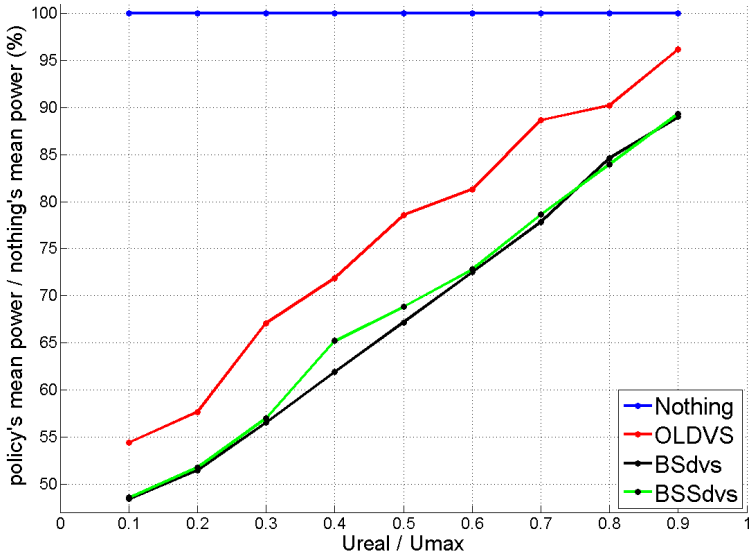

Figure 4.26: Percentage of the second task set

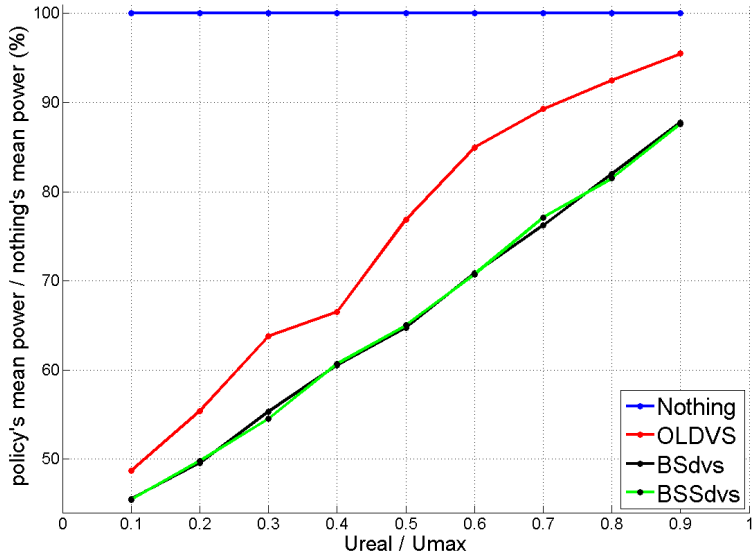

Figure 4.27: Percentage of the third task set

	U_{real}/U_{worst_case} (s)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.179	0.261	0.309	0.423	0.505	0.587	0.668	0.749	0.831
Oldvs	0.360	0.429	0.468	0.613	0.660	0.691	0.748	0.793	0.861
BSdvs	0.450	0.606	0.635	0.782	0.844	0.883	0.888	0.900	0.959
BSSdvs	0.447	0.600	0.643	0.748	0.852	0.895	0.876	0.928	0.957

Table 4.15: Execution time of the first task set

taining a mean power lower than Nothing (i.e.: without policies and always at the maximum frequency). Results of BSSdvs are not better than BSdvs as, when the first applies the splitting, the task uses more time than the case with BSdvs and so, the next jobs have less extra budgets, leading to use higher speeds. The same happens with OLDVS which consumes the bonus quickly (for the first task), no considering overhead that limit the use of lower frequencies.

So far, only the mean power consumption has been considered. Now, let us analyze in Table 4.15, Table 4.16 and Table 4.17 how the execution time is affected from the policies (keep in mind that the hyperperiod is $1s$).

The previous results have been plotted in Figure 4.28, Figure 4.29 and Figure 4.30. In almost all the measurements, BSdvs is characterized by a longer execution time, proving the previous notes.

	U_{real}/U_{worst_case} (s)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.172	0.253	0.334	0.415	0.496	0.577	0.658	0.739	0.820
Oldvs	0.473	0.612	0.613	0.677	0.700	0.775	0.771	0.848	0.861
BSdvs	0.617	0.780	0.847	0.881	0.908	0.930	0.941	0.943	0.971
BSSdvs	0.615	0.774	0.839	0.811	0.880	0.928	0.941	0.957	0.965

Table 4.16: Execution time of the second task set

	U_{real}/U_{worst_case} (s)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.150	0.232	0.315	0.396	0.478	0.561	0.643	0.724	0.806
Oldvs	0.532	0.595	0.623	0.729	0.700	0.707	0.750	0.805	0.860
BSdvs	0.631	0.758	0.817	0.863	0.930	0.936	0.951	0.959	0.970
BSSdvs	0.634	0.754	0.841	0.870	0.930	0.950	0.945	0.973	0.977

Table 4.17: Execution time of the third task set

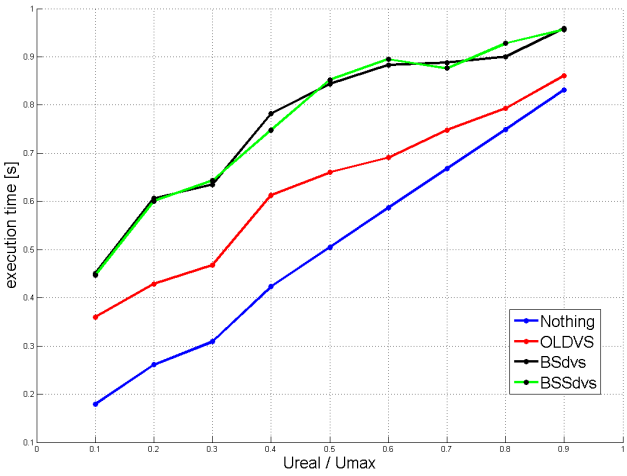

Figure 4.28: Execution time of the first task set

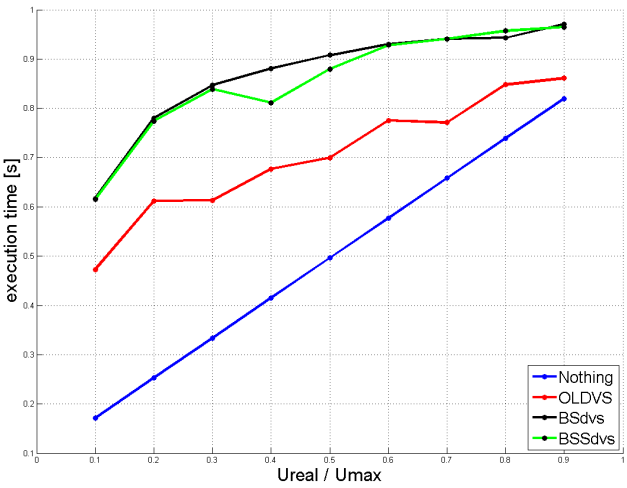

Figure 4.29: Execution time of the second task set

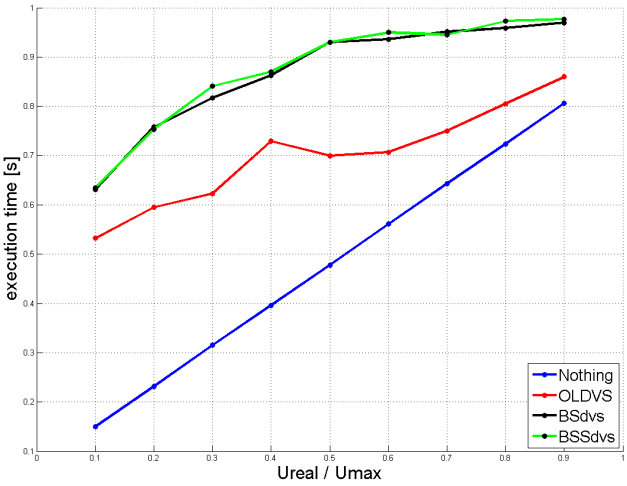

Figure 4.30: Execution time of the third task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.089	0.131	0.155	0.213	0.253	0.295	0.333	0.376	0.417
Oldvs	0.115	0.153	0.176	0.237	0.274	0.306	0.343	0.379	0.417
BSdvs	0.128	0.179	0.200	0.262	0.299	0.334	0.364	0.395	0.432
BSSdvs	0.128	0.179	0.202	0.258	0.303	0.337	0.362	0.401	0.432

Table 4.18: Energy of the first task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.086	0.127	0.167	0.208	0.250	0.290	0.332	0.371	0.411
Oldvs	0.129	0.177	0.206	0.244	0.277	0.317	0.346	0.384	0.415
BSdvs	0.150	0.201	0.240	0.273	0.307	0.339	0.370	0.401	0.433
BSSdvs	0.149	0.201	0.239	0.265	0.305	0.340	0.374	0.404	0.432

Table 4.19: Energy of the second task set

Although policies reduce the mean power, the execution time is make longer and this affect badly the overall energy consumption to execute the workload. When a DVFS algorithm is active, it uses a frequency slower than Nothing for a longer time. According to the power model, a DVFS policy is less convenient than Nothing as if the frequency halves, the execution time doubles but the same does not happen for the energy consumption. For example, at 40 MIPS the power consumption is $86.12mA$ but at 20 MIPS it is $60.14mA$, so if at the higher speed it executes for $1s$ the required energy is $k \times 86.12mJ$ and at the lower speed it is $k \times 2 \times 60.14mJ$ which is higher. Otherwise, with another consumption models, results may be different. The measurements reported in Table 4.18, Table 4.19, Table 4.20, Table 4.21, Table 4.22 and Table 4.23 report the energy required for **executing the tasks** (without considering what happen in the rest of the hyperperiod).

Figure 4.31, Figure 4.32 and Figure 4.33 plot the previous results in a graphical way.

Now, let us consider the energy required during the **entire hyper period** reported in Table 4.24, Table 4.25, Table 4.26, Table 4.27, Table 4.28 and Table 4.29. In the next measurements, the EXPS mode (exploiting the DOZE feature) has been used where indicated.

Figure 4.34, Figure 4.35 and Figure 4.36 plot the previous results in a graphical way, showing that OLDVS works well as it leaves the speed at the

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.075	0.118	0.160	0.201	0.242	0.285	0.327	0.368	0.410
Oldvs	0.130	0.168	0.202	0.246	0.272	0.306	0.340	0.379	0.418
BSdvs	0.144	0.191	0.229	0.265	0.305	0.337	0.368	0.399	0.433
BSSdvs	0.144	0.191	0.233	0.268	0.306	0.342	0.370	0.403	0.435

Table 4.20: Energy of the third task set

	U_{real}/U_{worst_case} (%)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Oldvs	133.97	118.84	115.26	112.61	108.82	104.93	103.87	101.43	100.95
BSdvs	143.55	137.27	129.59	123.30	118.13	113.41	109.11	105.03	103.79
BSSdvs	143.37	136.74	130.68	121.23	119.6	114.26	108.81	106.63	103.56

Table 4.21: Percentages of the first task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Oldvs	149.49	139.44	123.08	117.33	110.93	109.15	103.91	103.58	101.02
BSdvs	173.40	158.65	143.34	131.40	123	116.83	111.27	108.00	105.36
BSSdvs	173.30	158.28	143.18	127.52	122.20	117.01	112.47	108.75	105.16

Table 4.22: Percentages of the second task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Oldvs	172.32	142.08	126.36	122.39	112.59	107.14	104.24	102.82	101.80
BSdvs	191.17	161.88	143.67	131.71	125.87	118.20	112.78	108.51	105.54
BSSdvs	191.76	161.82	145.79	133.19	126.42	119.80	113.35	109.41	106.12

Table 4.23: Percentages of the third task set

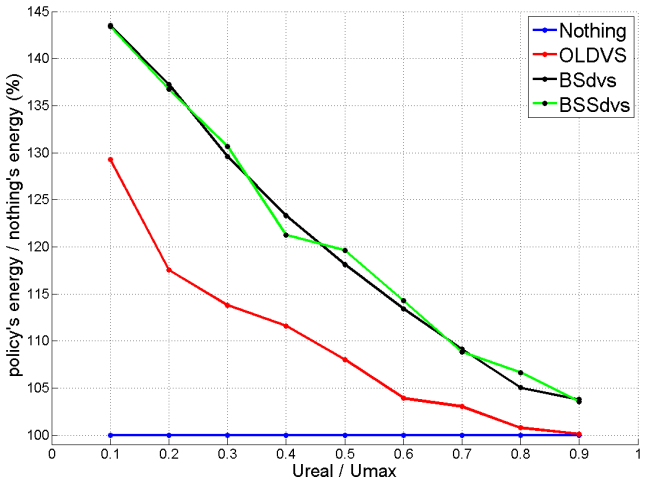

Figure 4.31: Analysis of the energy of the first task set

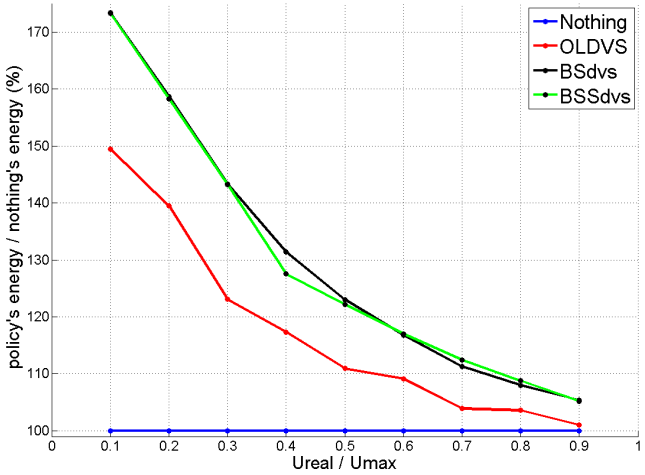

Figure 4.32: Analysis of the energy of the second task set

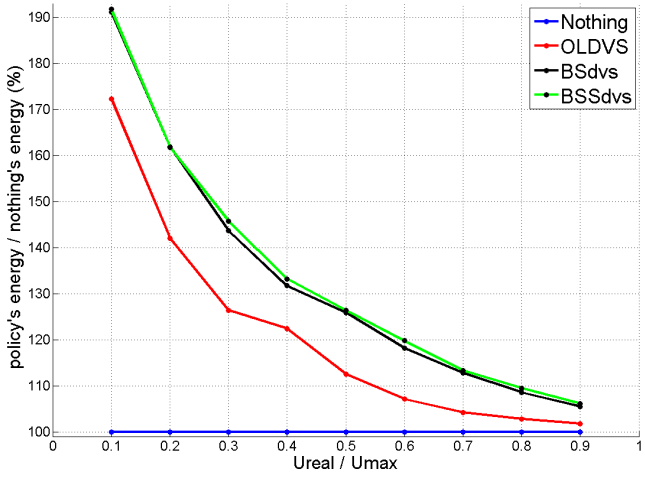

Figure 4.33: Analysis of the energy of the third task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.482	0.485	0.486	0.489	0.491	0.494	0.494	0.497	0.500
Nothing+EXPS	0.350	0.365	0.374	0.395	0.410	0.425	0.437	0.453	0.469
Oldvs	0.241	0.306	0.318	0.341	0.408	0.429	0.444	0.477	0.484
Oldvs+EXPS	0.229	0.276	0.290	0.321	0.366	0.391	0.412	0.448	0.463
BSdvs	0.392	0.370	0.378	0.370	0.382	0.399	0.422	0.451	0.458
BSdvs+EXPS	0.305	0.310	0.322	0.336	0.359	0.383	0.407	0.436	0.453
BSSdvs	0.394	0.374	0.375	0.383	0.379	0.398	0.429	0.444	0.462
BSSdvs+EXPS	0.281	0.274	0.292	0.329	0.345	0.364	0.396	0.420	0.446

Table 4.24: Energy required from the first task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.484	0.487	0.488	0.491	0.494	0.498	0.500	0.501	0.501
Nothing+EXPS	0.344	0.361	0.377	0.393	0.410	0.423	0.440	0.453	0.500
Oldvs	0.270	0.282	0.312	0.336	0.398	0.413	0.441	0.451	0.486
Oldvs+EXPS	0.241	0.260	0.289	0.314	0.358	0.378	0.409	0.427	0.461
BSdvs	0.332	0.307	0.315	0.331	0.352	0.374	0.400	0.429	0.448
BSdvs+EXPS	0.272	0.272	0.290	0.311	0.337	0.361	0.389	0.418	0.444
BSSdvs	0.328	0.307	0.317	0.359	0.364	0.375	0.406	0.428	0.453
BSSdvs+EXPS	0.268	0.272	0.288	0.320	0.339	0.360	0.386	0.413	0.438

Table 4.25: Energy required from the second task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	0.482	0.485	0.487	0.489	0.491	0.495	0.497	0.500	0.503
Nothing+EXPS	0.347	0.363	0.378	0.393	0.408	0.425	0.439	0.454	0.470
Oldvs	0.224	0.249	0.277	0.300	0.340	0.382	0.405	0.454	0.470
Oldvs+EXPS	0.212	0.239	0.267	0.293	0.329	0.365	0.390	0.427	0.450
BSdvs	0.326	0.307	0.317	0.331	0.339	0.368	0.393	0.419	0.447
BSdvs+EXPS	0.263	0.269	0.288	0.309	0.329	0.361	0.387	0.415	0.442
BSSdvs	0.317	0.310	0.312	0.333	0.343	0.369	0.399	0.417	0.447
BSSdvs+EXPS	0.261	0.270	0.284	0.310	0.330	0.359	0.389	0.411	0.442

Table 4.26: Energy required from the third task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Nothing+EXPS	72.47	75.29	76.93	80.78	83.36	86.02	88.59	91.21	93.76
Oldvs	49.96	63.06	65.50	69.78	83.11	86.92	89.90	96.03	96.71
Oldvs+EXPS	47.54	56.86	59.75	65.58	74.48	79.18	83.51	90.06	92.58
BSdvs	81.34	76.22	77.83	75.61	77.79	80.80	85.50	90.72	91.63
BSdvs+EXPS	63.30	63.88	66.21	68.77	73.08	77.59	82.40	87.67	90.55
BSSdvs	81.70	77.06	77.22	78.31	77.13	80.58	86.94	89.22	92.45
BSSdvs+EXPS	58.33	56.40	60.13	67.17	70.26	73.63	80.18	84.43	89.23

Table 4.27: Percentages of the first task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Nothing+EXPS	71.08	74.17	77.18	80.00	82.91	85.03	87.98	90.46	93.3
Oldvs	55.86	58.02	63.97	68.36	80.61	82.94	88.26	90.03	97.09
Oldvs+EXPS	49.88	53.37	59.16	63.95	72.56	76.03	81.86	85.23	92.16
BSdvs	68.70	63.13	64.47	67.44	71.30	75.09	79.95	85.71	89.54
BSdvs+EXPS	56.15	55.85	59.22	63.35	68.18	72.48	77.75	83.52	88.63
BSSdvs	67.80	63.08	64.89	73.11	73.74	75.41	81.20	85.53	90.46
BSSdvs+EXPS	55.49	55.83	59.06	65.26	68.65	72.39	77.17	82.49	87.51

Table 4.28: Percentages of the second task set

	U_{real}/U_{worst_case} (J)								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nothing	100	100	100	100	100	100	100	100	100
Nothing+EXPS	71.95	74.82	77.60	80.32	82.99	85.72	88.30	90.90	93.51
Oldvs	46.45	51.29	56.81	61.33	69.11	77.07	81.39	90.79	93.54
Oldvs+EXPS	44.02	49.23	54.77	59.84	66.84	73.72	78.48	85.56	89.52
BSdvs	67.59	63.35	65.11	67.68	68.95	74.33	78.93	83.85	88.9
BSdvs+EXPS	54.64	55.39	59.09	63.17	67.00	72.81	77.76	83.02	88.00
BSSdvs	65.85	63.87	63.99	67.94	69.74	74.58	80.35	83.52	88.93
BSSdvs+EXPS	54.21	55.70	58.37	63.25	67.15	72.54	78.28	82.27	87.82

Table 4.29: Percentages of the third task set

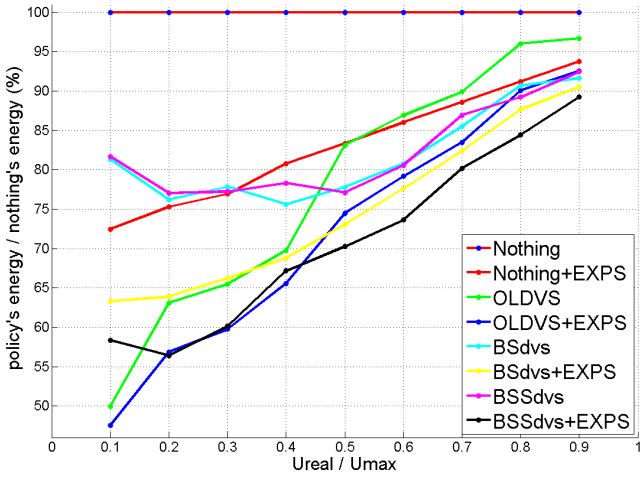

Figure 4.34: Analysis of energies with the first task set

last value when there are no tasks in execution. BSSdvs keeps guaranteeing a performance similar to BSdvs for the reason explained before. The gain given by EXPS decreases when the ratio U_{real}/U_{worst_case} increases.

The first consideration that comes from the presented measurements is that inter-period reclaiming policies are better than inter-task reclaiming algorithms as they use the bonus time to reduce the speed for many jobs and not only for the next.

Another note concerns the fact that, if we consider only the energy to execute the tasks, the consumption given by a policy is higher than Nothing. On the other hand, from a more general point of view which considers the whole hyperperiod, results are the opposite. This situation lets us realize that DPM algorithms may be more suitable on the architecture in use.

Under such considerations, the trade-off related to the energy becomes the following. On a side, we have DVFS policies, that allow us to have a lower mean power and a longer execution time. On the other side, Nothing+EXPS that is a simple DPM policy executes the workload quickly, while using a greater mean power. In such a scenario, the power model is the needle of the balance and the key for judging which is the best strategy. For example, with a consumption similar to the theoretical model $P(t) = s^3(t)$ (the most used in literature), DVFS policies works better than DPM solutions as slightly slowing the CPU down lets us save a significant amount of energy.

In the rest of this section, a methodology for selecting the most appropriate policy is presented.

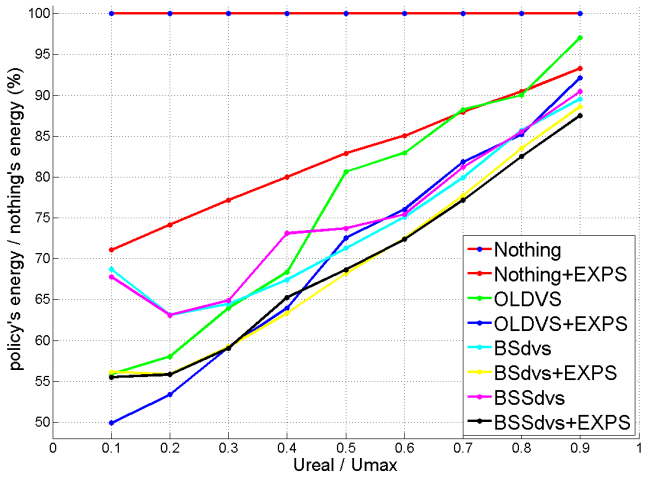

Figure 4.35: Analysis of energies with the second task set

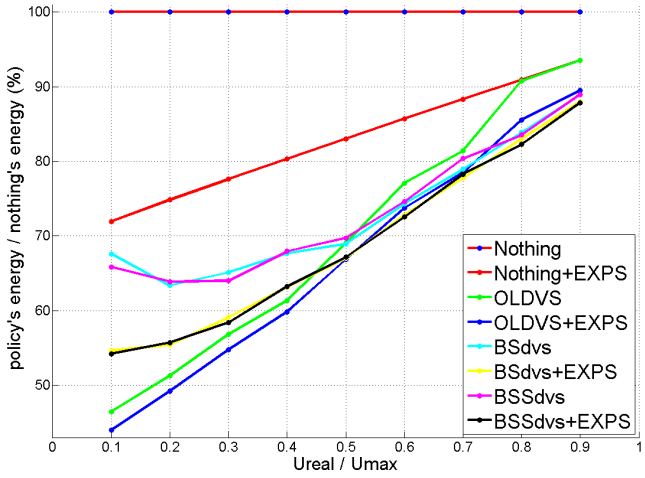

Figure 4.36: Analysis of energies with the third task set

In the power model, the consumption in the energy saving modes has been included as they are the other face of the medal for the DPM algorithms. If the core provides a special low-power mode with an extremely low consumption, then speeding up the execution to maximize the sleep time may be a valid solution. At this point, a new problem is introduces, given by the wake-up time which may forbid the use of such state for real-time reasons. For example, even though our board consumes $26.40mA$ ($56.38mA$ in IDLE mode), it requires $19.30ms$ to wake up the processor which is a really long time.

Let us formulate the problem in a more formal manner, as reported in Equation 4.4.

$$P_{mean\_power} \times T_{execution\_tasks} < P_{executing\_at\_F_{max}} \times T_{executing\_at\_F_{max}} + (T_{execution\_tasks} - T_{executing\_at\_F_{max}}) \times P_{power\_saving\_mode/nop}$$

The formula states that the energy, using any policy, must be less than the energy required by running always at the maximum speed. The first side is the energy that the policy requires, while the second is the energy that the system consumes at the maximum speed during the workload execution and during the inactive intervals. More precisely, the meanings of the variables are the followings:

- P_{mean_power}: mean power that the policy consumes during the task execution. It depends on the typical task execution time (U_{real}/U_{worst_case}), the policy, available speeds and overhead;
- $T_{execution\_tasks}$: amount of time in which the CPU executes the tasks (in a hyper period) with the policy;
- $P_{executing\_at\_F_{max}}$: mean power executing at the maximum speed;
- $T_{executing\_at\_F_{max}}$: amount of time in which the CPU executes the tasks (in a hyper period) at the maximum speed;
- $P_{power\_saving\_mode/nop}$: power consumption when there are not tasks in execution. It depends whether a power saving mode is used or not.

To better explain the idea, let us consider the simple example with two tasks, τ_1 (period of 10, WCET of 5) and τ_2 (period of 15, WCET of 6). The actual execution time is written in the task box. In the worst case, the schedule is reported in Figure 4.37. Frequencies are normalized with respect to the maximum one and the scheduler is assumed to be EDF. The hyper period is equal to 30 time units. The system runs at the highest speed for 27/30 time units and 3/30 time units are spent in a low-power state.

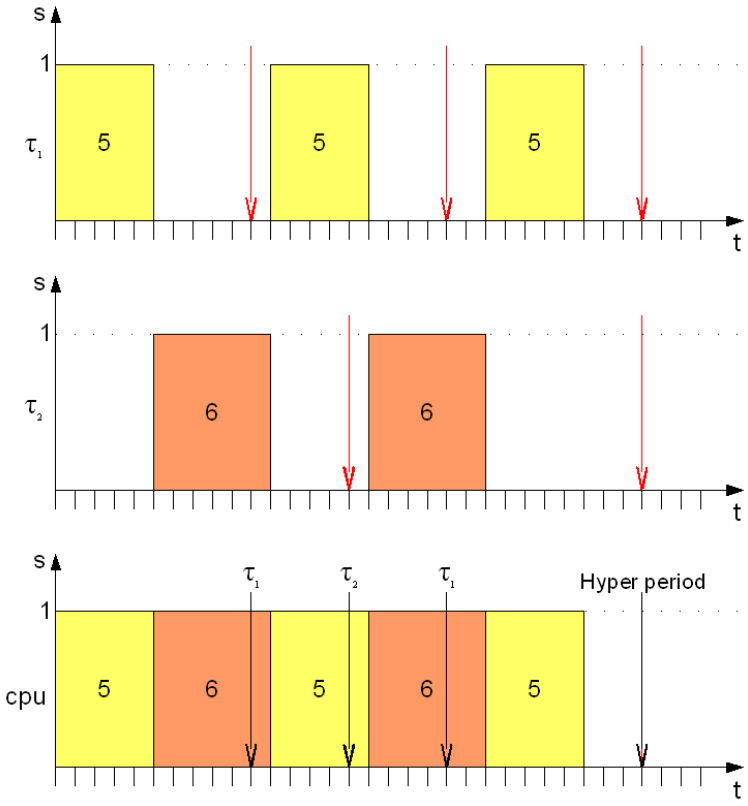

Figure 4.37: Schedule in the worst case

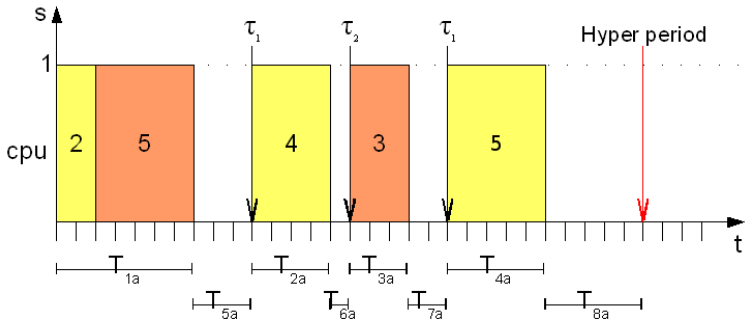

Figure 4.38: Schedule running always at the maximum speed

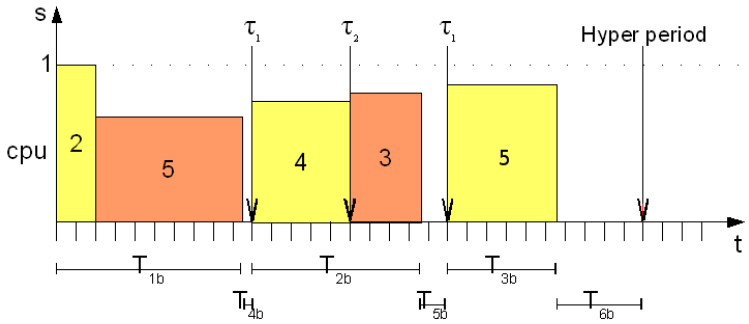

Figure 4.39: Schedule exploiting BSdvs

Now, suppose that the instances of the first task execute for 2, 4 and 5 time units and the instances of the second task execute for 5 and 3, the schedule assuming to run always at the maximum speed is reported in Figure 4.38. The example assuming BSdvs as energy saving policy is reported in Figure 4.39.

When the DVFS policy is used, the mean power is always lower or equal to the mean power running at the maximum speed, but requiring a computational time longer. In order to justify the usefulness of the policy, the overall energy consumption of Figure 4.39 must be lower than the overall energy consumption of Figure 4.38. The global execution time without any policy is $T_{1a} + T_{2a} + T_{3a} + T_{4a}$ and the time spent without execution is $T_{5a} + T_{6a} + T_{7a} + T_{8a}$. Having introduced the policy, the execution time is $T_{1b} + T_{2b} + T_{3b}$ while the inactivity time is $T_{4b} + T_{5b} + T_{6b}$. Rewriting Equation 4.4 according to the new values, this is what happens:

$$P_{mean\_power} \times (T_{1b} + T_{2b} + T_{3b}) + P_{power\_saving\_mode} \times (T_{4b} + T_{5b} + T_{6b}) < P_{executing\_at\_F_{max}} \times (T_{1a} + T_{2a} + T_{3a} + T_{4a}) + P_{power\_saving\_mode} \times (T_{5a} + T_{6a} +$$

$$T_{7a} + T_{8a})$$

The inactivity time can be expressed as difference between hyperperiod and total execution time:

$$P_{mean\_power} \times (T_{1b} + T_{2b} + T_{3b}) + P_{power\_saving\_mode} \times (HYPER\_PERIOD - (T_{1b} + T_{2b} + T_{3b})) < P_{executing\_at\_F_{max}} \times (T_{1a} + T_{2a} + T_{3a} + T_{4a}) + P_{power\_saving\_mode} \times (HYPER\_PERIOD - (T_{1a} + T_{2a} + T_{3a} + T_{4a}))$$

Assuming that $(T_{1a} + T_{2a} + T_{3a} + T_{4a}) > (T_{1b} + T_{2b} + T_{3b})$, the formula becomes:

$$P_{mean\_power} \times (T_{1b} + T_{2b} + T_{3b}) < P_{executing\_at\_F_{max}} \times (T_{1a} + T_{2a} + T_{3a} + T_{4a}) + P_{power\_saving\_mode} \times ((T_{1b} + T_{2b} + T_{3b}) - (T_{1a} + T_{2a} + T_{3a} + T_{4a}))$$

Writing execution times as the number of instructions divided by the frequency (in MIPS) ($T_x = \frac{\#INSTRUCTIONS}{f_{execution}}$):

$$P_{mean\_power} \times \frac{\#INSTRUCTIONS}{f_{mips\_mean}} < P_{executing\_at\_F_{max}} \times \frac{\#INSTRUCTIONS}{f_{mips\_max}} + (\frac{\#INSTRUCTIONS}{f_{mips\_mean}} - \frac{\#INSTRUCTIONS}{f_{mips\_max}}) \times P_{power\_saving\_mode}$$

This is the final version of the inequality:

$$\frac{P_{mean\_power} - P_{power\_saving\_mode}}{f_{mips\_mean}} < \frac{P_{executing\_at\_F_{max}} - P_{power\_saving\_mode}}{f_{mips\_max}}$$

The two important values are P_{mean_power} (depending on the policy and the tasks) and $P_{power\_saving\_mode}$ (depending on the system). If they are both unknown, their limits can be found and this gives us information about which is the most appropriate, DVFS or DPM. The frequency is not an issue as the power model is known which matches the consumption with the frequency and back ($f_{execution} = C \times Power + D$ where $C = \frac{1}{1.35} = 0.74$ and $D = \frac{-32.12}{1.35} = -23.79$). Since our data are in mA, the inequality is divided by the voltage (which is constant).

Figure 4.40 studies the maximum mean consumption that **a generic** can could consume in order to be more appropriate than *Nothing* strategy. In other words, what is the maximum consumption for having a DVFS policy more useful than the DPM approach. The analysis is obtained varying the consumption in the low-power state and the mean frequency. Such function is described by the curve with internal lines in Figure 4.40. The plane (smoothed curve) is the power model of the board (independent from the task set execution).

If the typical execution speed of a task set with a DVS policy and the absorbed current in low-power states are known then, it is possible to define:

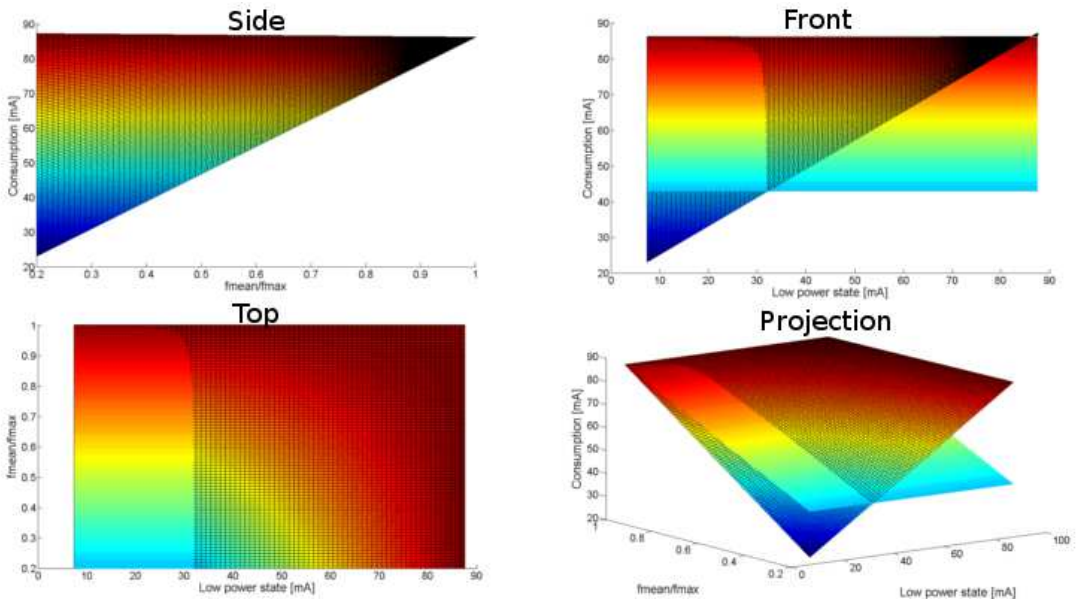

Figure 4.40: Trade-off analysis between DVFS and DPM approaches

- if the surface value is lower than the plane point, the DVFS policy is not convenient;
- if the surface value is greater than the plane point, the policy is more convenient;
- if they are equal, they are interchangable.

Now, let us focus our attention on the intersection line between the two curves. In Figure 4.41, the maximum values of the consumption in the low-power state so that Nothing is better than any policy are reported. Those results have been obtained varying the mean speed. As it has been already said, this analysis heavily depends on the adopted policy and available speeds. The result related to the model is a straight line as it directly depends on the power model which is linear. Real consumptions are different as they are affected by differences between theoretical and actual power model and overhead.

This let us claim that the theoretical values offer a valid upper bound.

Measurements with different frequency sets have not been carried out as only the consumption values would have been affected and not their trends.

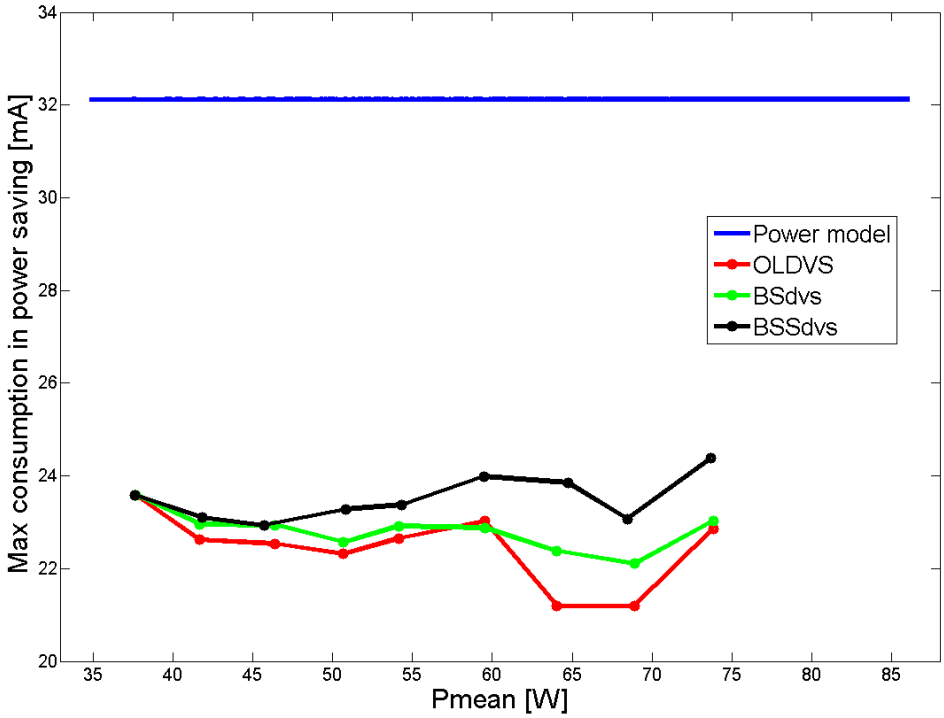

Figure 4.41: Maximum allowed consumption in low-power state with respect to the average consumption

4.5 Conclusions

In this section, the following list contains a set of advises (best practices) to speed up the decision process:

1. designers must know the power profile of the processor (or board);
2. the energy saving states must be chosen according to the trade-off between wake-up time power consumption;
3. the CPU driver should offer a frequency set as wide and dense as possible;
4. in many signal processing algorithms, the execution times are generally the same in every instances, but this is not always true. This has a heavy impact on pessimistic algorithms;
5. once designers have all the required information, they can draw a graph similar to Figure 4.40, for evaluating the most appropriate approach.

Chapter 5

Conclusion

This thesis has presented a framework for tiny real-time kernel which implements several energy aware algorithms for radio modules, radio modules and CPUs.

The simplest component which has been presented is the driver of the radio module which, exploiting the Sleep, Idle and Fully-operative modes, manages the transitions between the possible state, according to the radio use, for reducing the required energy while guaranteeing time constraints (considering the temporal overheads related to the waking-up times). A simple protocol has been implemented (based on TDMA and star topology), in order to test the proposed approach.

Servomotors are the muscles of many applications and for this reason attracted my attention. The manager select the most appropriate updating period according to the applied torque at the servo's arm. The load curve has been measured for the Hitec HS-645MG.

The main topic of this thesis has been the power management for the CPU. The analysis started proposing an review of the state of art and went on the studying seveal algorithms which aim at reducing of the mean power. Such algorithms have been implemented within a real operation system and tested on a real processor. According to the results, the main note concerns that lower mean power does not imply a lower energy consumption. The reason is that the execution time may increase significantly if the speed is slightly reduced (which in turn reduces the actual power). This leads to claiming that the usefulness of an energy saving algorithm is strictly related to the power model in use. In the end, this thesis has pointed out that a trade off between DPM approaches and DVFS policies may be the best strategy according to the considered hardware.

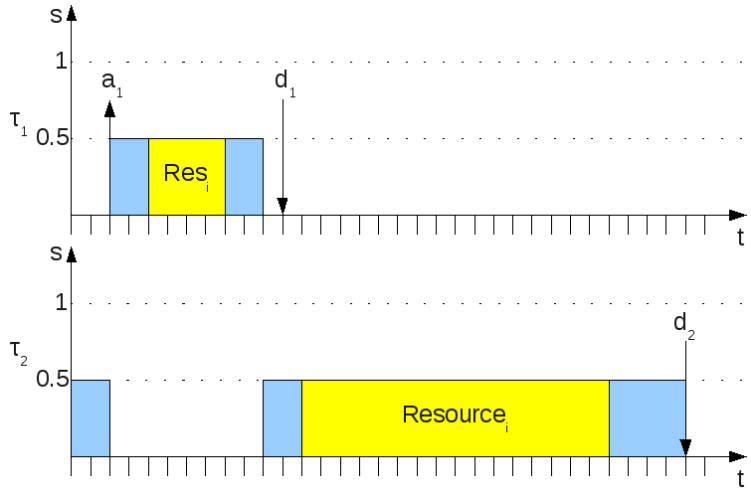

Figure 5.1: Running always at speed 0.5

5.1 Future improvements

Concerning the servomechanisms, the most important improvement consists of developing a system that do not require user notifications. It would be possible by adding a sensor, within the servo, that interacting with the module informs it about the applied torque on the arm.

The CPU management can be extended in three possible ways, explained in the followings:

- implementing inter-period reclaiming algorithms;
- considering a more realistic computational model. More precisely, this analysis assumed that computational times scales linearly with the frequency but it is not always true (memory accesses and the external I/O may be speed-independent). This issue may be addressed by dividing the times of the tasks in two types, the first scales with the speed and the second does not [10];
- managing shared resources, as deadline miss and data inconsistency may occur when different speeds are exploited [21] due to the introduced blocking time. For instance, consider the example in Figure 5.1 and Figure 5.2 where tasks are executed always at speed 0.5 and 1.0, respectively.

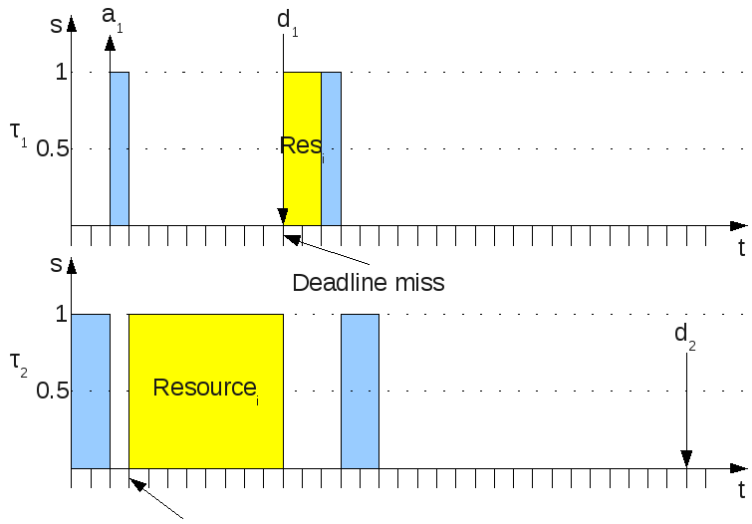

Figure 5.2: Running always at speed 1.0

In addition, another interesting improvement would be the integration of DPM and DVFS policies with band allocation constraints in wireless protocol [28]. Such solution would be especially useful for small wireless sensor networks where the nodes' main activity consists in sensing physical values and then sending them to a collector.

Bibliography

[1] "Scientific american," http://www.scientificamerican.com/.

[2] "Evidence srl," http://www.evidence.eu.com/.

[3] "Embedded solutions srl," http://www.es-online.it/.

[4] Microchip, *dsPIC33FJXXXMCX06/X08/X10 Data Sheet High-Performance, 16-Bit Digital Signal Controllers*, 2009, http://www.microchip.com/.

[5] F. Marinoni, Buttazzo and Franchino, "Kernel support for energy management in wireless mobile ad-hoc network," in *OSPERT 2005 Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2005.

[6] M.-S. Gong, Y. R. Seong, and C.-H. Lee, "On-line dynamic voltage scaling on processor with discrete frequency and voltage levels," in *ICCIT '07: Proceedings of the 2007 International Conference on Convergence Information Technology*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1824–1831.

[7] "Wikipedia," http://www.wikipedia.org/.

[8] T. L. Martin and D. P. Siewiorek, "Nonideal battery and main memory effects on cpu speed-setting for low power," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 1, pp. 29–34, 2001.

[9] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1995, p. 374.

[10] E. Bini, G. Buttazzo, and G. Lipari, "Speed modulation in energy-aware real-time systems," in *ECRTS '05: Proceedings of the 17th Euromicro*

Conference on Real-Time Systems. Washington, DC, USA: IEEE Computer Society, 2005, pp. 3–10.

[11] T. Okuma, T. Ishihara, and H. Yasuura, "Real-time task scheduling for a variable voltage processor," in *Proceedings of the 12th international symposium on System synthesis*, ser. ISSS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 24–. [Online]. Available: http://dl.acm.org/citation.cfm?id=857198.857958

[12] Swaminathan and Chakrabarty, "Real-time task scheduling for energy-aware embedded systems," 2000.

[13] V. Swaminathan and K. Chakrabarty, "Investigating the effect of voltage-switching on low-energy task scheduling in hard real-time systems," in *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*. New York, NY, USA: ACM, 2001, p. 251.

[14] C. S. E. Bini, "Optimal two-level speed assignment for real-time systems," 2009.

[15] H. Aydin, P. Mejía-Alvarez, D. Mossé, and R. Melhem, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 95.

[16] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2001, pp. 89–102.

[17] Y. Zhu and F. Mueller, "Feedback edf scheduling of real-time tasks exploiting dynamic voltage scaling," *Real-Time Syst.*, vol. 31, no. 1-3, pp. 33–63, 2005.

[18] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo, "Adaptive dynamic power management for hard real-time systems," in *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2009.

[19] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. Buttazzo, "Adaptive power management for hard real-time event streams," 2010.

[20] ——, "Periodic power management schemes for real-time event streams," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, 2009.

[21] F. Zhang and S. T. Chanson, "Processor voltage scheduling for real-time tasks with non-preemptible sections," in *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 235.

[22] U. of Virginia, *Hotspot library*, http://lava.cs.virginia.edu/HotSpot/.

[23] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2003, pp. 2–13.

[24] S. Wang and R. Bettati, "Reactive speed control in temperature-constrained real-time systems," *Real-Time Syst.*, vol. 39, no. 1-3, pp. 73–95, 2008.

[25] T. Chantem, R. P. Dick, and X. S. Hu, "Temperature-aware scheduling and assignment for hard real-time applications on mpsocs," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008, pp. 288–293.

[26] S. Heo, K. Barr, and K. Asanović, "Reducing power density through activity migration," in *ISLPED 2003: Proceedings of the 2003 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2003, pp. 217–222.

[27] Wu and XU, "Temperature-aware task scheduling algorithm for soft real-time multi-core system," *SIGBED Rev.*, 2009.

[28] V. Devadas and H. Aydin, "Real-time dynamic power management through device forbidden regions," in *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 34–44.