Master's Thesis

Operating System-Level Power Management Based on Power Estimation

Michael Düss

Institute for Technical Informatics
Graz University of Technology
Head: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Reviewer: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger

Advisor: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger
Dipl.-Ing. Andreas Genser

Graz, September 2010

Abstract

Today adequate power management is a focal point in the embedded systems domain. The ever rising complexity of embedded systems causes an increase in power consumption, which opens a gap between power supply technologies and the power consumption demands for mobile systems. Hence, the evolution of battery technology does not keep up with the power thirst of new microprocessor systems. Effective countermeasures to circumvent these trends are introduced by power management mechanisms. Dynamic power management is a popular technique to deal with required power optimizations.

In this thesis power management strategies will be designed and implemented on the operating system-level. Power information derived from an on-board power estimation unit will be exploited. The goal is to develop power management strategies for systems with limited power with the support of the acquired power information.

Kurzfassung

Heutzutage ist der Einsatz von Power Management Techniken eine Kernaufgabe im Bereich von Eingebetteten Systemen. Die Komplexität von Eingebetteten Systemen und somit auch die benötigte Leistung steigen stetig. Für mobile Systeme bewirkt dies eine größer werdende Lücke zwischen der notwendigen und der verfügbaren Leistung. Grund dafür ist der Umstand, dass die Weitereinwicklung der Batterietechnologien nicht mit dem stark steigenden Leistungsverbrauch von neuen Microprozessor Systemen mithält. Der Einsatz von Power Management Strategien ist eine effektive Gegenmaßnahme um diesem Trend entgegen zu wirken. Dynamisches Power Management ist dabei eine gängige Technik um geforderte Leistungsoptimierungen zu realisieren.

Diese Masterarbeit befasst sich mit dem Entwurf und der Implementierung einer Power Management Strategie auf Betriebssystem-Ebene. Ziel ist die Entwicklung eines Power Managements für Systeme mit limitierter Versorgungsleistung. Die Realisierung erfolgt mit der Unterstützung einer zusätzlichen Power Estimation Unit, welche Leistungsinformationen über das benutzte System zur Verfügung stellt.

STATUTORY DECLARATION

I declare that I have authored this thesis independently that I have not used other than the declared sources/resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, 1st September 2010

Michael Düss

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Power consumption is an important task since the beginning of embedded systems, especially in the division of mobile and nomadic systems [Ols08], [She08]. The main issue is the limited battery capacity. On the one hand, more and more computational power is needed and on the other hand the power supply should be stable as long as possible. The trend in the embedded world shows that the performance of new microprocessors is increasing continuously, but the improvement in battery technology is much slower [She08]. Figure 1.1 illustrates the gap between performance and operating time of mobile devices. Hence, the gap between power consumption and power supply is growing. There are two possibilities to work against this issue. First, battery technology improvements are required and second, measures to make systems more power-efficient are needed.
Another interesting section of embedded systems is energy harvesting. One goal of such energy harvesting systems is to stay within power constraints and to ensure system stability during a changing power budget [KHZS07].
Power consumption is also an important factor for all electric systems. The reason is simple and always the same, money. Energy costs have increased rapidly during the last years (Figure 1.2) and therefore customers and consumers are interested in reducing power consumption for a reduction of costs.
Furthermore the terms "Green Computing" and "Green IT" [Cam09], [Mur08], [Rut09] become more and more popular. This means that people start thinking more about the environment.

1.1 Motivation

A designer of a power management for mobile systems has to handle the following two problems:

- Energy is limited
- Power is limited

The problem of limited energy addresses common mobile devices. The goal is to reduce power and energy consumption with the aim to maximize the duration. Technologies for

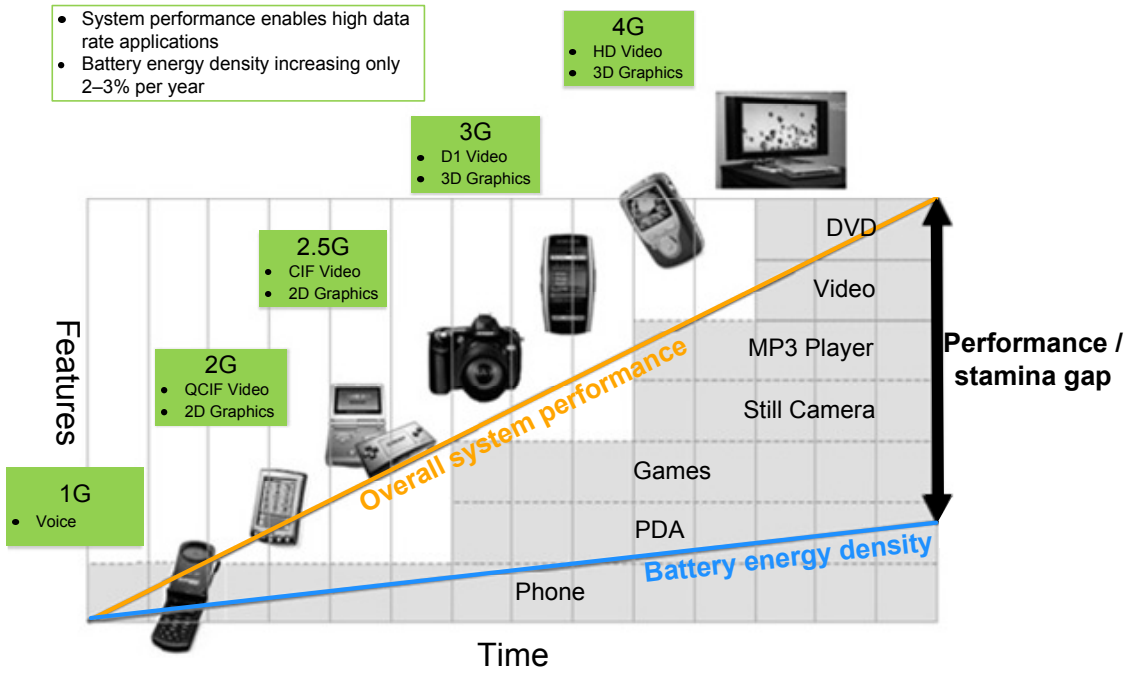

Figure 1.1: Performance/Stamina gap for mobile devices [She08]

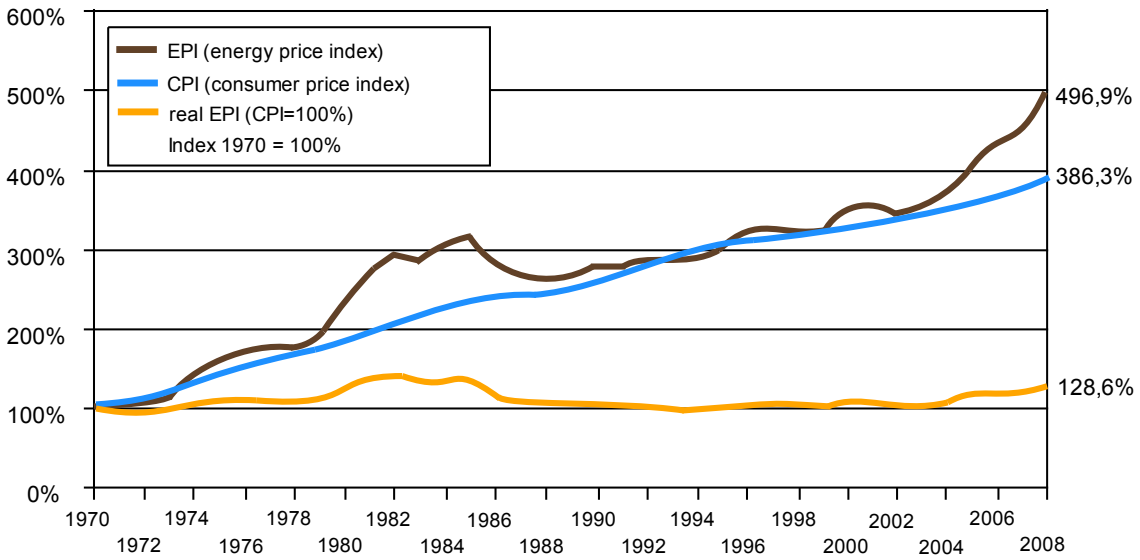

Figure 1.2: Trend of energy costs in Austria between 1970 and 2008; Source: Statistik Austria, calculated by Austrian Energy Agency

reaching this goal are Dynamic Voltage Scaling (DVS) and Dynamic Voltage and Frequency Scaling (DVFS), which have been introduced by Chandrakasa et al. [CSB92]. With DVS/DVFS it is possible to reduce the power consumption of a microprocessor by modeling the clock frequency and the supply voltage. Initially, these techniques were used

to support systems without real-time requirements. Later, special algorithms were developed which are designed for real-time systems and static workload. Today, it is also required to support real-time systems with a dynamic workload.
On the other hand, limited power is an issue [KHZS07]. To deal with that it is necessary to ensure a stable operation of the system. The area of energy harvesting systems is a common example. Given power constraints in form of power budgets require the system to adopt frequency and supply voltage to stay within these constraints. Power budgets are dependent on various power supply sources such as vibration energy, solar energy or thermal energy.
For this class of systems, DVS/DVFS or task rescheduling are countermeasures to avoid violations of a given power budget.
Power management is in general available on different abstraction levels: hardware-, compiler-, operating system- and application-level. This thesis aims at power management on operating system-level for systems with limited power.

1.2 Goals

The goal of this master thesis is a power estimation supported power management on operating system-level[1]. This mainly includes the design, the implementation and the evaluation of a chosen power management. To reach this main goal some subgoals are defined:

- Studying of literature, which is relevant for writing the chosen master thesis. The main topics are: Power management techniques, CMOS power consumption, operating systems (OS) for embedded systems and energy harvesting.
- Investigate a μcLinux OS and determine its potential for OS power management adaptations.
- Implement and integrate the OS on a LEON3 system on chip (SoC) platform.
- Establish a communication between the OS and the available power estimation unit.
- Design and implement a concept that enables power estimation based power management on OS-level.
- Evaluate the implemented power management for a set of benchmarking applications.

1.3 Structure

A summary of the necessary background is given in Chapter 2. It includes the topics: CMOS power consumption, energy harvesting and operating systems for embedded systems.

[1]This thesis is part of the POWERHOUSE project that is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 815193.

Chapter 3 shows an overview of the different power management variants. Reactive/static power management and proactive/dynamic power management are introduced and the relevant related work is given.
The concept of this master thesis is shown in Chapter 4. It includes an overview of the target system and its main components. Also a detailed description of the implemented power management is given.
Chapter 5 shows the implementation of the chosen power management. It includes the hardware and software setup of the target system and shows code snippets of the implementation.
The evaluation and the results of the implemented power management are shown in Chapter 6. It focuses on the evaluation setup, the outcome of the OS-level power management its impact on the LEON3 SoC's performance.
The conclusion is given in Chapter 7. It summarizes the results of the thesis and and provides an overview of future improvements.

Chapter 2

Background

CMOS power consumption, energy harvesting and real-time operating systems are important in the context of this master thesis. Knowledge about the power consumption of CMOS circuits is often used to realize power and energy reductions in embedded systems. The relation between power consumption, clock frequency and supply voltage is exploited. Energy harvesting is a technology which uses the energy of the environment to provide additional power for battery-driven systems. The power management, which is implemented in this thesis, is an alternative to support energy harvesting systems. Furthermore, the topic of real-time operating systems is relevant for this thesis, because the chosen power management is implemented on the operating system-level.

2.1 CMOS Power Consumption

It is important to understand the different factors of the CMOS power consumption to gain power and energy reductions. The power consumption of a CMOS circuit is determined by two components [Tex97]:

- Static power consumption
- Dynamic power consumption

2.1.1 Static Power Consumption

The basic element in the CMOS technology is an inverter [Tex97]. Figure 2.1 shows the two operating modes of a CMOS inverter circuit:

- **Case 1:** If the input is logical 0, the n-MOS device is OFF and the p-MOS device is ON. Then the output is logical 1 (Vcc).
- **Case 2:** If the input is logical 1, the n-MOS device is ON and the p-MOS device is OFF. Then the output is logical 0 (GND).

One of the transistors is always OFF and hence, theoretically no power is consumed. Only a small amount of power is consumed, because of the reverse-bias leakage between diffused regions and the substrate. Equation (2.1) describes the static power consumption (P_S) of

a CMOS circuit. It depends on the sum of the leakage currents (I_{CC}) and on the supply voltage (V_{CC}).

$$P_S = I_{CC} \cdot V_{CC} \tag{2.1}$$

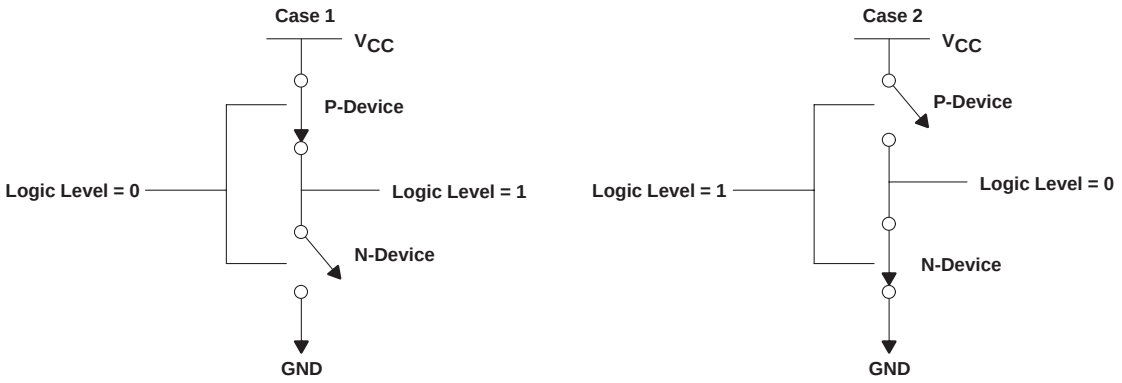

Figure 2.1: CMOS inverter modes for static power consumption [Tex97]

2.1.2 Dynamic Power Consumption

The dynamic power consumption of a CMOS circuit is the sum of transient power consumption (P_T) and the capacitive-load power consumption (P_L) [Tex97].

Transient Power Consumption

The transient power consumption of a CMOS circuit is caused by the switching of the transistors from on logical state to the other [Tex97]. The power is consumed through the current (switching current) which is needed to charge the internal nodes (Cpd - dynamic power-dissipation capacitance) and the current, which flows during the switching of the p-channel transistor and the n-channel transistor. So the switching frequency, the rise and fall times of the input signal and the internal nodes affect the dynamic power consumption. Equation 2.2 shows the transient power consumption (P_T).

$$P_T = C_{pd} \cdot {V_{CC}}^2 \cdot f_I \cdot N_{SW} \tag{2.2}$$

C_{pd} : *dynamic power − dissipation capacitance*
V_{CC} : *supply voltage*
f_I : *input signal frequency*
N_{SW} : *number of bits switching*

Capacitive-Load Power Consumption

Additionally, power is consumed through the charging of external load capacities [Tex97]. The amount of consumed power of this charging process also depends on the switching frequency. The capacitive-load power consumption (P_L) is described with Equation 2.3.

$$P_L = C_L \cdot {V_{CC}}^2 \cdot f_O \cdot N_{SW} \tag{2.3}$$

C_L : *external (load) capacitance*
V_{CC} : *supply voltage*
f_O : *output signal frequency*
N_{SW} : *number of outputs switching*

2.2 Energy Harvesting

Usually, the power supply of wireless and embedded systems is employed by using batteries. This can become a problem if these systems are expected to operate for long durations, because battery energy is limited. An alternative to support battery-driven systems is the technology of energy harvesting.
Basically, a harvesting node is a system which gains energy from its environment with the goal to provide additional energy to batteries. The harvested energy can be only a fraction of the required energy. A big advantage of energy harvesting in comparison to battery stored energy is the fact that the potential of environment energy is infinite. A disadvantage is that the maximum power which can be gained is limited. Therefore it is useful to develop a power management which is able to stay within power constraints. Figure 2.2 shows a typical graph of solar activity. It illustrates the harvestable power dependent on the daytime.

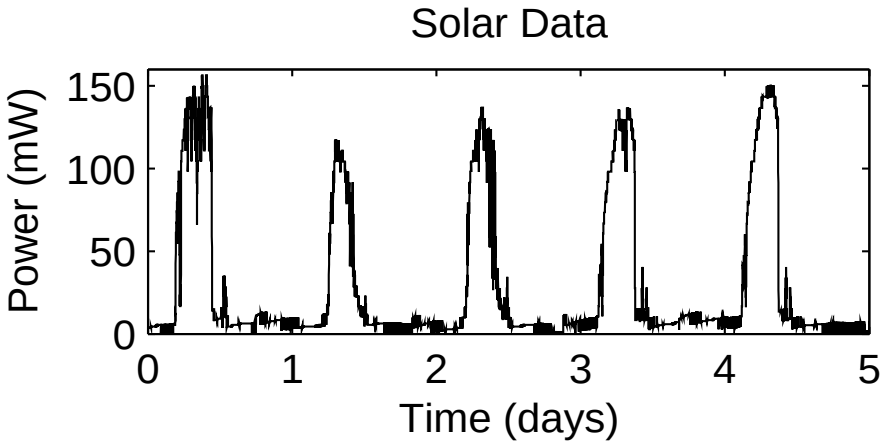

Figure 2.2: Solar activity graph [KHZS07]

A basic example for an energy harvesting device is a desk calculator with a solar cell. A more complex example is a network of harvesting nodes, where each node gains energy by

using the same or multiple technologies for harvesting, with the goal to harvest a maximum of energy.
The design of power management of energy harvesting systems is distinguished from battery supplied devices. The goal of power management at battery supplied devices is to minimize power and energy consumption or to maximize lifetime, while required performance constraints must be held. In the case of energy harvesting systems two different modes are common. First, harvesting nodes are used to support a battery supplied system aiming at the minimization of power consumption and the maximization of lifetime. Second, harvesting nodes are the only energy source and therefore the device is independent of battery energy. The life time of such a device is theoretically infinite. This mode is often called "energy neutral", because a requested performance level can be supported as long as necessary. Unsurprisingly, the power management design goals of such an energy neutral mode are quite different from the goals mentioned above. Designers have to deal with two main considerations [KHZS07]:

- **Energy neutral operation**
  The approach of energy neutral operation means that the required energy of a system is always lower than the harvested energy. It is important to guarantee that the system is able to operate stable. Often such a system consists of many nodes, and every node harvests its own energy. In addition to a stable operation, also a constant system performance over the whole network must be guaranteed.

- **Maximum performance**
  In energy harvesting systems also the aspect of the maximum performance has to be considered. If a system operates energy neutral it is also a question of which maximum performance can be reached in a given harvesting environment. As mentioned above, this depends again on the different network components. A possibility to solve the issue would be a harvesting system which is always able to support maximum required power. This means that the minimum power output of the harvesting system is high enough to support maximum system performance. A disadvantage of this approach are the excessively high costs to provide the necessary energy. Also in some cases a harvesting system is not able to deliver any energy. An example would be a system which harvests solar energy. If the sun does not shine, the power output is zero. A promising approach is to establish a power management system between the harvesting network and the consumer device. Then the task of the power management system is to hold balance between available and required power.

2.2.1 Conditions for Energy Neutral Operation

The mode of energy neutral operation is influenced by the power consumption on the one side and the power supply on the other side. The power output from the energy source is defined as $P_s(t)$, at time t, and the consumed power is defined as $P_c(t)$. Three different cases which show the possible operation modes for energy neutral operations can be distinguished [KHZS07]:

- **Harvesting system with no energy storage**
  Address energy harvesting systems which are able to use the harvested energy di-

rectly and without a storage buffer.
This kind of energy harvesting devices are able to operate when

$$P_s(t) \geq P_c(t) \quad \forall \ t \tag{2.4}$$

is fulfilled.
These systems have some disadvantages. In the case $P_s(t) < P_c(t)$ the provided power is not sufficient and the system is not able to operate correctly. Also at times when $P_s(t) > P_c(t)$ the power $P_s(t) - P_c(t)$ is lost.

- **Harvesting system with ideal energy buffer**
  Such systems are used if the profile of energy generation is strongly different from the profile of power consumption. It is suitable to use an energy buffer to store the harvested energy. In the ideal case the energy storage is a buffer where the energy can be stored without losses. The amount of energy inefficiency during charging and leakage will be unattended. The following equation describes the case of ideal energy buffering during operation:

$$\int_0^T P_c(t)dt \leq \int_0^T P_s(t)dt + B_0 \quad \forall \ T \in [0, \infty) \tag{2.5}$$

  B_0 it the initial energy of the buffer.

- **Harvesting system with a non-ideal energy buffer**
  A system without an energy buffer is not practical and an ideal energy buffer does not exist. Instead of these considerations a harvesting system with a non-ideal energy buffer will be used. The buffer could be a battery or an ultra-capacitor. The disadvantages of such energy buffers are: limited capacity, limited charging efficiency and leakage.

2.2.2 Harvesting Sources

Our environment offers several different harvestable energy sources. For a designer it is often difficult to decide which and how many energy sources should be used. The following energy sources are available [MM05]:

- **Kinetic energy**
  Two sources are used for harvesting kinetic energy: first, the motion of moveable parts and second, mechanical deformation. Three technologies are known to transform kinetic energy into electrical energy: piezoelectric effect, electrostatic generation and magnetic induction.
  The **piezoelectric effect** describes the effect that certain materials possess electrical polarizability, which is proportional to a subjected mechanical stress.
  The basic components of an **electrostatic generator** are an electrical field and a

moveable part. Energy will be generated if the moveable part moves against the field.
If a conductor is moved within a magnetic field and the conductor crosses magnetic field lines, **magnetic induction** happens and energy will be generated.

- **Electromagnetic radiation**
  Electromagnetic radiation can be used for energy harvesting in form of solar energy or radio frequency (RF) radiation.
  Solar energy is used in form of solar-powered photovoltaic systems, which are able to transform the electromagnetic radiation into electrical energy. For mobile devices solar energy is one of the most suitable options.
  RF radiation is a common source to power identification cards. The devices harvest the needed energy from the electromagnetic energy of the environment. There is a high potential of RF radiation in cities and areas with high population, because of the large number of RF sources. The issue with RF radiation technology is to convert the potential energy into useful energy.

- **Thermal energy**
  Another form of energy in our environment is thermal energy. Persons, animals, machines or other natural sources can be used to transform thermal energy into electrical energy. The technology behind is called thermovoltaic, especially thermal generators. A thermal generator consists of a couple of thermal conductors, which are made of different material. The conductors are connected. Based on a given temperature difference between the connected conductors electrical voltage will be generated. The level of the generated voltage depends on the material of the conductors.

Classification of Harvesting Sources

One way to classify harvesting sources is to divide them dependent on their characteristics regarding controllability and predictability. The following types are possible [KHZS07]:

- **Uncontrolled but predictable**
  These energy sources are not controllable, but it is possible to predict the behaviour in certain time ranges. This means that a forecast model can be made. Wind and solar energy for example cannot be controlled, but it is possible to predict the weather for certain areas during a time period within a certain error margin.

- **Uncontrollable and unpredictable**
  These energy sources are difficult to handle because they are not controllable and forecast models are often too complex and impractical. A representative for this kind of energy sources is vibration energy in an indoor environment. Harvesting this energy is possible, but a prediction is nearly impossible.

- **Fully Controllable**
  An example of these energy sources is a self-power flashlight. A user can shake the device to generate energy and use it whenever it is required.

- **Partially Controllable:**
  RF energy is an example for a partially controllable energy source. Such an RF source can be installed indoors and also a network of harvesting nodes. Then the total amount of harvested energy depends on RF propagation characteristics within the environment. This propagation cannot be controlled by a designer or a user. But the designer or user is responsible for the exact location of the harvesting nodes.

2.2.3 Practical Observations

Design and development of an energy harvesting system is a difficult task. Therefore, an accurate look at the target device is necessary. Table 2.1 gives an overview of common wearable battery-supplied devices. Also the average power consumption and the usual duration are mentioned.
It is also difficult to decide which energy source should be used for energy harvesting. Table 2.2 gives a short summary of available energy sources. Furthermore, different application possibilities and the potential power which can be gained are shown.
Finally a summary (Figure 2.3) of the research and development (R&D) results of the last decade is shown. The figure gives an overview of the power ranges of different harvesting technologies. Also the amount of the realized R&D projects is shown (red points).

	Device Type	
	Power Consumption	**Energy Autonomy**
Smartphone	1 W	5 hours
MP3 player	50 mW	15 hours
Hearing Aid	1 mW	5 days
Wearable Sensor Node	10 μW	Lifetime
Cardiac Pacemaker	50 μW	7 years
Quartz watch	5 μW	5 years

Table 2.1: Selected battery-operated systems and their average power consumption [VvSGH09]

Source		Device Type	
		Characteristics	**Harvested Power**
Photovoltaic	Indoor	0.1 mW/cm^2	10 μW/cm^2
	Outdoor	100 mW/cm^2	10 mW/cm^2
Vibration / Motion	Human	0.5 m@1Hz, 1 m/s^2@50Hz	4 μW/cm^2
	Industrial	1 m@5Hz, 10 m/s^2@1kHz	100 μW/cm^2
Thermal Energy	Human	20 mW/cm^2	30 μW/cm^2
	Industrial	100 mW/cm^2	1-10 mW/cm^2
RF Cell phone		0.3 μW/cm^2	0.1 μW/cm^2

Table 2.2: Characteristics of various energy sources and amount of typical harvested power [VvSGH09]

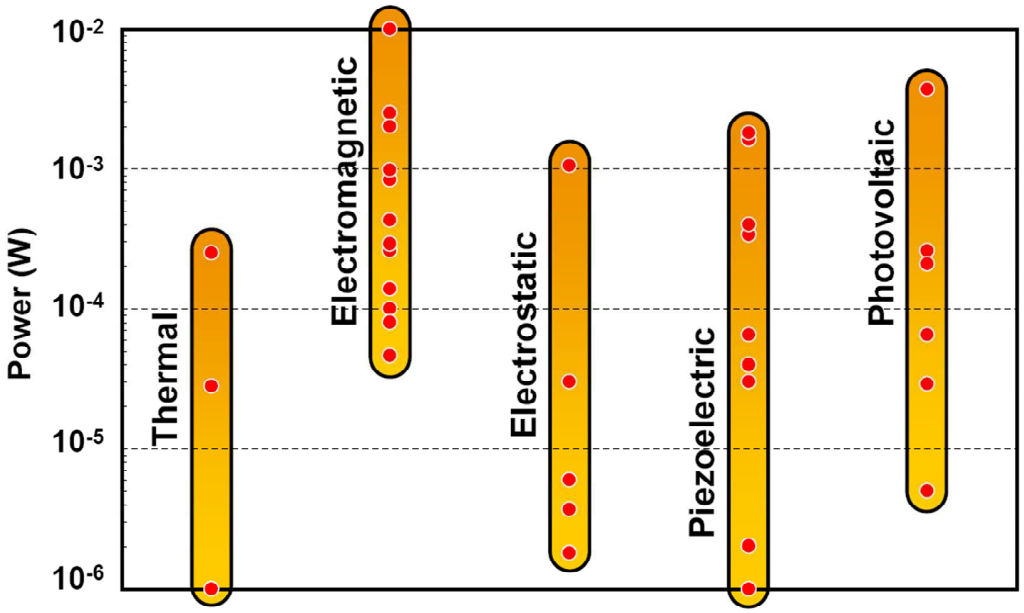

Figure 2.3: Power potential from the environment by the means of using several types of energy harvesters [VvSGH09]

2.3 Real-Time Operating Systems (RTOS)

As mentioned, power management can be done on several abstraction levels. In these thesis the focus is set on the OS-level, and therefore it is necessary to understand the basics of operating systems, especially real-time operating systems.

Real-time operating systems are a subsection of operating systems. To understand real-time operating systems it is necessary to know the basics of usual operating systems.

2.3.1 Operating Systems

The basic purpose of an operating system is to provide a buffer between the user and the hardware of a system [Hea03]. An operating system is a software environment that provides a constant interface and a set of utilities to enable users to utilize the system quickly and efficiently. Therefore, it is possible for programmers to write application programs, which can be moved to other systems, because of hardware independence. Normally, some debug tools are available, which help to speed up the testing process.

2.3.2 Multitasking Operating Systems

A single tasking operating system is not suitable for most embedded systems [Hea03]. They do not fulfill the requirement that multiple applications can run simultaneously and provide intertask control and communication. To handle this a suitable operating system for embedded systems must be able to handle multiple tasks.
A multitasking operating system works by dividing the processor time into discrete timeslices. For the completion of the execution, a task requires a certain amount of computation time (timeslices). The kernel of the operating system is responsible for the task scheduling and decides when and how long a task gets processor time. A task is not executing continuously until completion, it is is interleaved during execution with other tasks. This implies a sharing of processor time, because only one task can use the processor at the same time.

Context Switching

As mentioned, multitasking operating systems are based on a multitasking kernel, which controls the time slicing mechanisms [Hea03]. A timeslice of a task is a period of time which determines how long a task is allowed to run before an interruption occurs. This time period is triggered by the system timer. After an interruption happens, the task will be put on the “ready” list and wait for further execution. Before another task is allowed to run, the current processor registers must be saved in a special table of the current task (task control block). This information is needed for next time when the task gets execution time. Before a new task is allowed to run, its registers are loaded. This whole process, when one task is replaced by another, is called context switch. Figure 2.4 shows a time slicing pattern and Figure 2.5 summarizes a context switch.

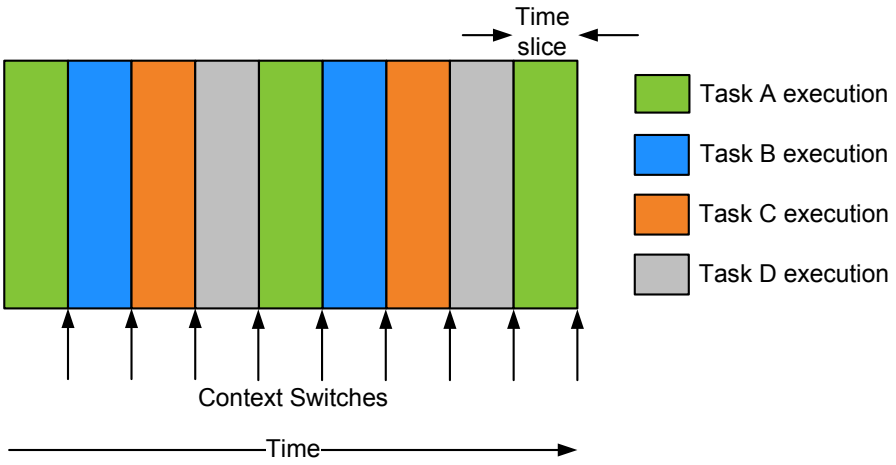

Figure 2.4: Timeslice model [Hea03]

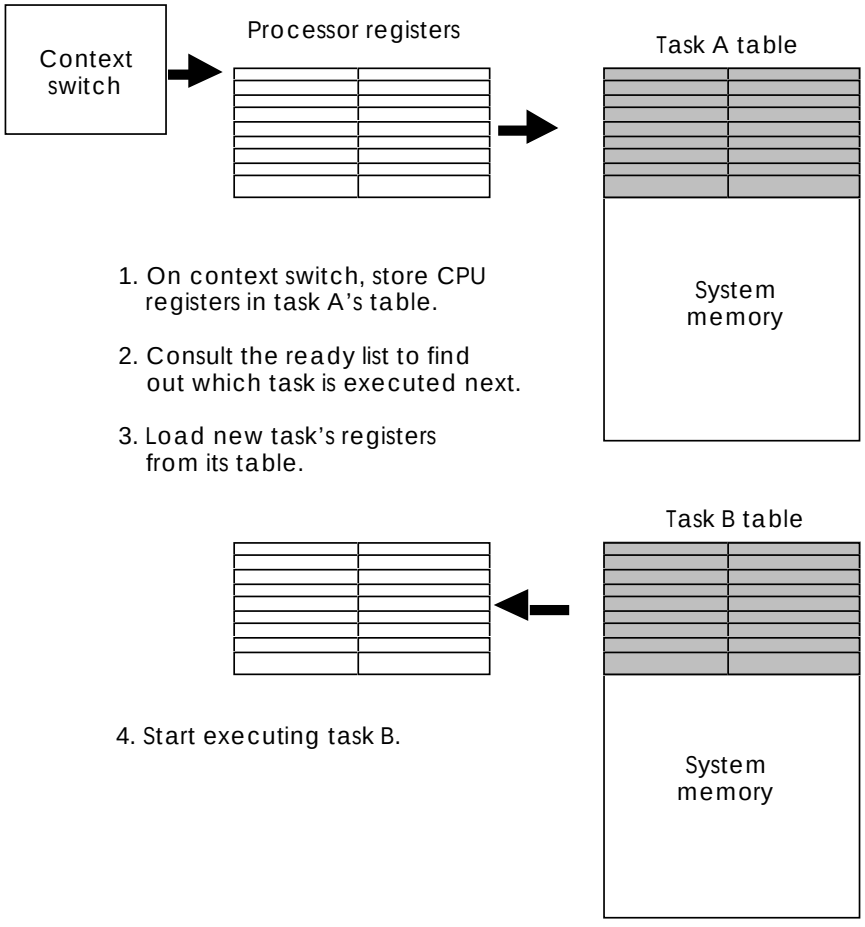

Figure 2.5: Model of context switching [Hea03]

Kernel

The kernel controls the scheduling of tasks, memory usage and prevents tasks from corrupting each other [Hea03], [Tan03]. If memory sharing is used, the kernel controls the share of program modules, such as high-level language run-time libraries. In the case of memory usage a set of memory tables is maintained. Access to memory is handled on the basis of these tables. It allows to protect the resources, such as physical memory and peripheral devices, from the user. This is very important to ensure the system's integrity. For communication between tasks, message passing can be implemented with the kernel as message passer. If task A wants to stop task B, a call to the kernel will be executed and task B will be stopped. Alternatively, task B can be delayed for a certain period of time or forced to wait for a message. On a typical real-time operating system two different types of messages exist:

- Messages (Flags) which can only control tasks, but cannot carry any implicit information. They are often called semaphores or events.
- Messages which can control tasks and can also carry information. They are often called messages or events.

Figure 2.6 shows a typical real-time kernel. There different lists can be seen, which display the different states of a task. The most important states are:

- run - task is in processing
- ready - task is ready for processing
- blocked - task is dormant, suspended, waiting for an event or waiting for an command, etc.

Additionally, several more service tasks are needed if a real-time kernel should work inside a full operating system. These are tasks to perform I/O services, file handling and file management services, task loading, user interface and driver software. Typical kernel size is less than 16 kbyte and will often grow with service tasks into a large 120 kbyte operating system.
The service tasks surround the kernel in form of layers and the result is a typical onion structure. User tasks build the outmost layer. Figure 2.7 shows such a structure.
In a typical system, all these service tasks and user tasks are controlled, scheduled and executed by the kernel.

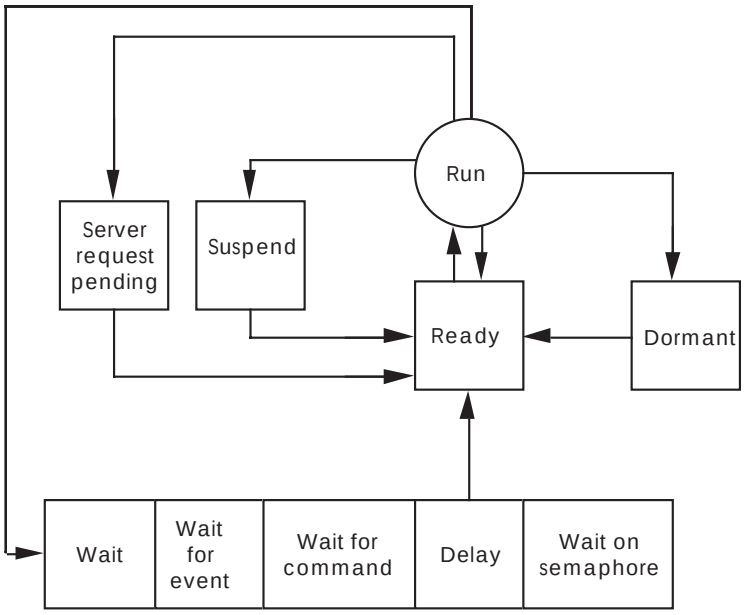

Figure 2.6: State diagram for a typical (real-time) kernel [Hea03]

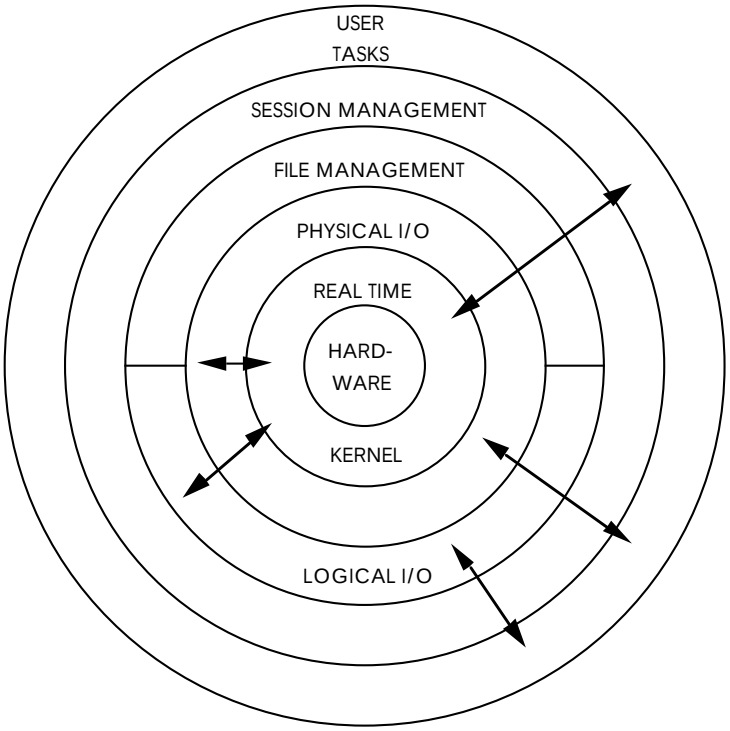

Figure 2.7: A typical operating system structure [Hea03]

2.3.3 What is a Real-Time Operating System?

Today many operating systems exists, which are also described as “real-time”. These operating systems provide some additional features to usual operating systems.
A basic characteristic of a real-time operating system is its defined response time to external units [Hea03]. It is essential for a real-time operating system to react within a maximum defined time after an interrupt occurs. The response time is dependent on system performance and current workload. Still, it is essential for a real-time operating system to not exceed a certain maximum time. Any operating system that is able to handle this requirement can be described as real-time.
Another example for a real-time application: As mentioned for industrial control it would have serious consequences if the system does not have real-time characteristic. It is easy to imagine what would happen if an automatic assembly line controlled by an embedded system does not respond in time. Figure 2.8 shows the case that the system has a certain time to stop the conveyor belt after the limit switch generate an interrupt. The response does not need to be instantaneous but in time.
As we can see in this example a real-time operating system's internal mechanism must be able to handle external interrupts in guaranteed times.
After an interrupt is generated, the current running task is halted and the interrupt handling will start.

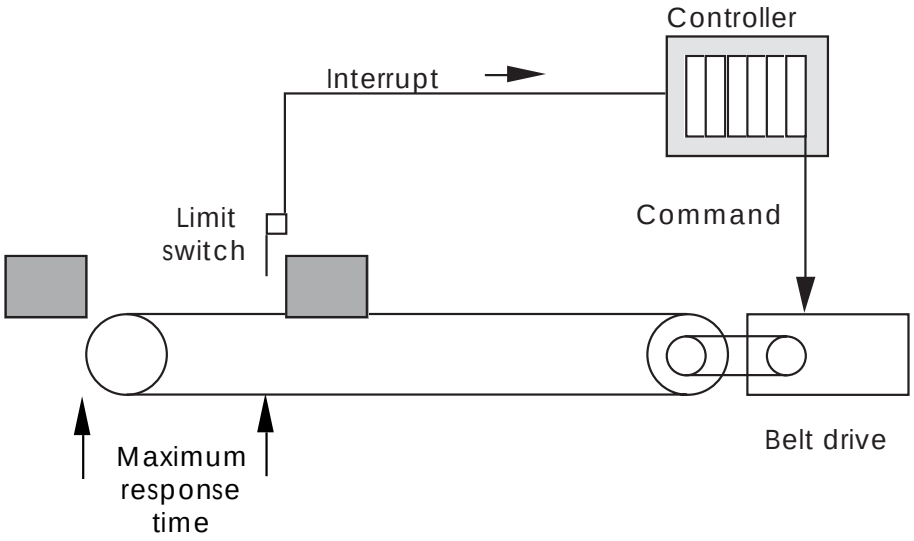

Figure 2.8: Example of a real-time response [Hea03]

2.3.4 Scheduling

The scheduler is responsible for the distribution of the processor time to the single tasks. It decides when and how long a process or task is allowed to use the processor for computing. To fulfill the different requirements for a scheduler a lot of scheduling algorithms exist. In case of simple flow control tasks, high throughput is necessary. To manage systems with user interaction, fast response time would be expected.
A simple classification for scheduling algorithms can be done by separation in non-preemptive and preemptive algorithms. Non-preemptive means that a running task cannot be interrupted by a task with higher priority. In case of preemptive scheduling a task with higher priority is allowed to halt the running task. The following scheduling algorithms are common:

Shortest Job First (SJF)

If the current running task is finished, the task with the shortest remaining computation time is allowed to run next [SGD09]. It is not possible to interrupt a running task and therefore the procedure is a non-preemtive scheduling. The algorithm is unfair, because for short tasks it is possible to overtake long tasks and thereby short tasks are privileged. The SJF scheduling is unusable for real-time systems.

First Come First Served (FCFS)

FCFS scheduling is also very simple and non-preemtive. Tasks get computation time in the order of arrival [SGD09]. Fairness is guaranteed, because every task gets computation time and a starvation of a task is excluded. The procedure is also unusable for real-time applications.

Round Robin

The computation time is separated in timeslices of equal length. Tasks will be put in a waiting queue and will be chosen with the First-In-First-Out (FIFO) procedure to execute [SGD09]. After the end of a timeslice the running task will be stopped and put back to the waiting queue. The round robin scheduling algorithm is fair, because the computation time is shared equally among all tasks. The algorithm can be used for applications with lossy real-time requirements.

Round Robin with Priorities

With the adoption of priorities an upgrading of the round robin scheduling is done [SGD09]. In this procedure the tasks get priorities. Tasks with same priority are sorted in a group and a separate waiting queue exists for each priority group. The algorithm always takes tasks from the queue with the highest priority. If this queue is empty, the queue with the next lower priority will be used. The problem of this scheduling is a possible starvation of tasks with low priority, because they get less computation time. A solution is to increase the priority of a task in relation to its waiting time. In this form the procedure is used in modern operating systems (Windows XP, Linux).

Rate Monotonic (RM)

RM scheduling is a static real-time scheduling algorithm which calculates the scheduling at compile-time for all possible tasks. The following assumptions are made [LL73]:

- Time critical tasks occur periodically.
- Current period must be finished before execution of next period starts.
- Dependences between different tasks are impossible.
- Execution for each task is constant.
- Non-periodic tasks are not time critical.
- Each task gets a priority. Tasks with higher priority are allowed to interrupt tasks with lower priority (preemtive system).

A RM scheduling exists if the following equation is fulfilled:

$$\sum_{i=1}^{n} \frac{\Delta t_i}{Per_i} \leq 1 \tag{2.6}$$

$$n: \textit{ number of tasks},\ \Delta t_i: \textit{ computation time of } i^{th} \textit{ task},\ Per_i: \textit{ period of } i^{th} \textit{ task}$$

The placing of priorities follows the rule that the task with the shortest period gets the highest priority.
Liu also shows in [LL73] that if the load μ of a processor is $\leq 70\%$, then a successful scheduling is guaranteed:

$$\mu = \sum_{i=1}^{n} \frac{\Delta t_i}{Per_i} \leq n(2^{n/1} - 1) \tag{2.7}$$

The right side of the equation converges against 0.693 if $n \to \infty$. A successful scheduling is also possible for higher workload, but not guaranteed. Under special requirements (all task periods are a multiple of the shortest period), systems with 100% load can be scheduled. The big advantage of the RM scheduling is the guaranteed schedule. Disadvantages of this algorithm are that the schedule is planned static and if load is higher than 70% no schedule commitment can be made.

Earliest Deadline First (EDF)

In comparison to RM scheduling EDF scheduling is a dynamic scheduling algorithm for real-time systems [SGD09]. The scheduling is calculated again and again at run-time for all executable tasks. Basis for the EDF scheduling is a preemtive system with dynamic priority distribution. The other requirements are equal to RM scheduling. EDF scheduling

follows the simple rule that the next task which is allowed to run is the task with the earliest deadline.
Advantages of the EDF scheduling are the simple implementation and that the processor can be used up to a load of 100%. A big disadvantage of the EDF algorithm is that a successful scheduling cannot be guaranteed for all cases.

2.3.5 Comparison between Common OS and RTOS

The following table 2.3 summarizes the main differences between a common OS and an RTOS.

	Common OS	**RTOS**
Calculation Results	logically correct results	logically and temporally correct results
Response time	no guarantee for a maximum response time: violation of time constraints are acceptable	response time is guaranteed; failure could be disastrous
Scheduling	slow scheduler, poor temporal resolution and accuracy; common Linux: 10ms	fast scheduler, high temporal resolution and accuracy; typical 20-200 μs
	common scheduling procedures: time-sharing, first come first served, round-robin, round-robin with priorities	deterministic scheduling procedures such as earliest deadline first; short and deterministic timing for task switching
	scheduling is efficient, but not predictable	scheduling is control- and predictable
Optimizations	optimization for maximum of data throughput	optimization for a minimum response time; typical 20 μs
	optimized to an average workload	optimized to a maximum workload (worst case)
Kernel mode	only system processes are allowed to run in kernel mode	system processes and time critical user processes are allowed to run in kernel mode
Interrupts	periodic timer-interrupt	periodic timer-interrupt is not imperatively, but has a high resolution (One-Shot-Timer)
	interrupts are partially locked (masked)	fast interrupt handling

Table 2.3: Comparison between common OS and RTOS [Abb06]

Chapter 3

Power Management

Today device manufacturers must handle the challenge that consumers expect rich-featured (mobile) devices. Full functionality is only one part of a successful device. It is also necessary to provide long-lasting battery life to fulfill consumer demands. Furthermore, the topics energy saving, green IT and carbon footprint have become popular in the last years. These reasons initiate manufactures to deal with power management. Not only consumers' reasons stimulate the development of sophisticated power management. In addition to the positive effects for our environment, energy savings help to save money. The fact of money savings is a strong motivation for manufactures to develop and establish power management.
For appropriate power management a coordination and cooperation of all parts within a whole system is necessary. To establish such a power management in an embedded system it is important to indicate the parts with the most power saving potential. In case of an embedded system often the CPU is a major source regarding power consumption. It is also a fact that embedded systems often have peripheral devices, which are sometimes inactive during the whole power-on time.
To consider a power management strategy it is necessary to categorize power management in two parts:

- **Reactive/static power management**
  Power consumption is influenced by switching of unused/inactive system components.
- **Proactive/dynamic power management**
  Power consumption is influenced continuously/dynamically during operation.

It is recommended to use both parts together to generate a maximum of power savings [Kit09].

3.1 Reactive/Static Power Management

Reactive/static power management means the control of the power consumption of a system by switching on/off power consuming system parts (Ethernet controller, USB port, etc.) [ACP09], [Kit09], [LDM01], [LLTW06], [Ols08]. Great value is given to system parts

with high power consumption, since they offer more power saving potential. To realize reactive/static power management, different working states for the system parts of an embedded system are defined. Some parts only have the states "On" and "Off", where "On" means 100% power consumption and "Off" means no consumption. A more complex reactive/static power management with five different working states is shown in Figure 3.1.
The standards Advanced Power Management (APM) and Advanced Configuration and Power Interface (ACPI) have been established for the area of personal computers. APM was the pre-standard of ACPI. ACPI defines global power states (Working, Sleeping, Soft Off, Mechanical Off) for the whole system. Also a set of sub-states for the "sleeping" state is provided. The states will be managed by the operating system.

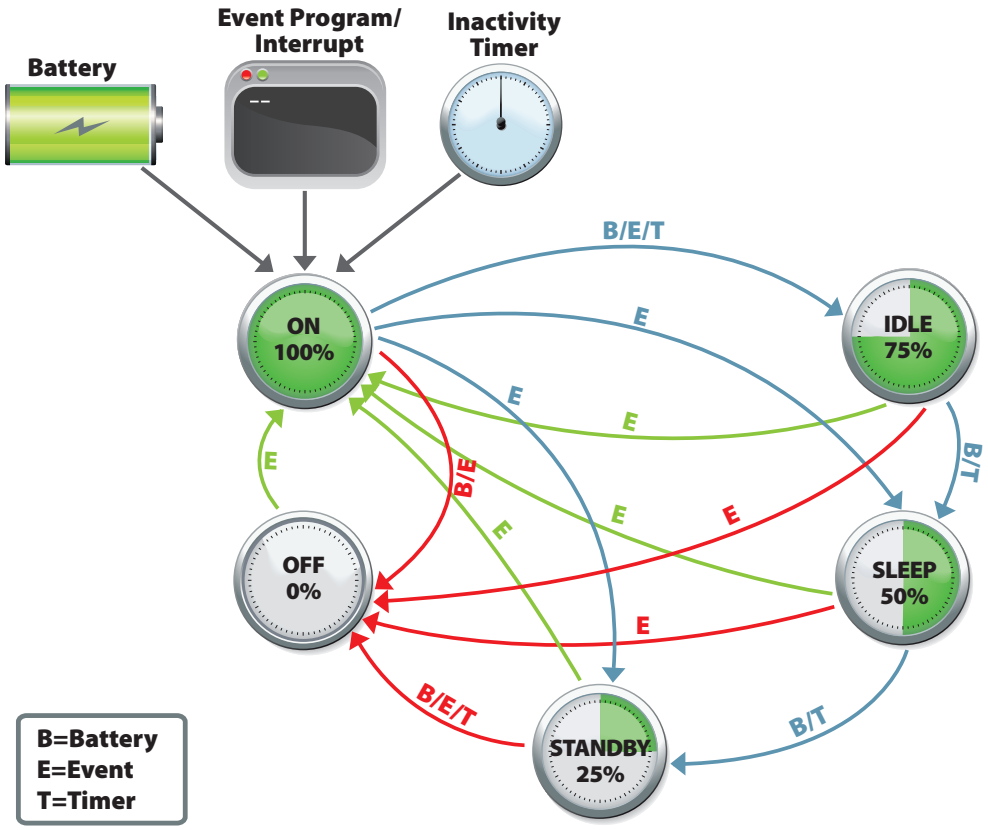

Figure 3.1: State diagram for a reactive/static power management model [Ols08]

3.2 Proactive/Dynamic Power Management

Proactive/dynamic power management is used to influence the power consumption of a system continuously/dynamically [CSB92], [Kim06], [She08], [SRH05], [TTD03] . Different techniques are used to enable Proactive/dynamic power management for the target sys-

tems: Dynamic Voltage Scaling (DVS), Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Process and Temperature Compensation (DPTC) and task rescheduling.

- **Dynamic Voltage Scaling (DVS) / Dynamic Voltage and Frequency Scaling (DVFS)**
  Originally, DVS works only with voltage scaling and DVFS uses both voltage and frequency scaling. Nowadays DVS/DVFS has become the standard technique for proactive/dynamic power management.
  DVS/DVFS uses the fact that the power consumption of a CMOS circuit depends linearly on its clock frequency and quadratically on its supply voltage (Chapter 2.1):

$$P \propto fV^2 \tag{3.1}$$

  P : *power*
  f : *clock frequency*
  V : *supply voltage*

  Furthermore, the number of cycles for completion of a task-set is independent of the clock frequency, and the total energy is quadratically dependent on the square of the supply voltage (Eq. (3.2)).

$$E \propto V^2 \tag{3.2}$$

  E : *energy*
  V : *supply voltage*

- **Dynamic Process and Temperature Compensation (DPTC)**
  DPTC is an improvement to DVS/DVFS. It is assigned that the maximum clock frequency of an electronic circuit depends on the process speed and the operating temperature [She08]. Normally, the supply voltage of a circuit is set to support the required frequency, also in worst case. Worst case means that the circuit operates on very high temperatures (maximum defined temperature) as well as special variations in the manufacturing process. If a "better case" (moderate temperature) is present, the supply voltage can be reduced and a stable operation is still guaranteed. When DPTC is used, a reference circuit is implemented, and this circuit measures the current frequency. Dependent on the process technology and the operating temperature the circuit lowers the supply voltage to a minimum, which is even high enough to support the required clock frequency. So the DPTC concept is used to adjust the supply voltage to match the SoC temperature and the process corner.

Proactive/dynamic power management is available on different levels of abstractions:

- **Hardware-level**
  Bennini et al. [BCMS01] show a battery-driven dynamic power management on hardware-level. Dependent on the battery charge, the discharging rate is adapted. Several policies are introduced and also the battery characteristics are taken into account. Lee et al. [LNS+07] introduce a hardware power management unit (PMU). It uses Dynamic Voltage and Frequency Scaling (DVFS) for a reduction of the power consumption. The PMU improves the load regulation of the supply voltage with co-locking the clock frequency and the supply voltage.

- **Compiler-level**
  Hsu and Kremer [HK03] design and implement a compiler algorithm for CPU power reduction. The compiler identifies code regions with low CPU load through memory access. Then DVFS is used to reduce the power consumption of the CPU while these code regions are computed.

- **OS-level**
  A division into two sections is helpful: (i) power management on OS-level for minimizing the power consumption (Section 3.2.1) and (ii) power management on OS-level for systems with limited power (Section 3.2.2).

- **Application-level**
  Gu et al. [GCO06] introduce DVFS for gaming applications. They show the power saving potential of interactive computer games.

3.2.1 Power Management on OS-Level for Minimizing the Power Consumption

Dynamic power management on OS-level is commonly done through special task scheduling algorithms in combination with DVFS.
Many power management algorithms on OS-level focus on hard real-time systems with multiple tasks. The main demand is to choose the optimal operating voltage for a task to stay within its timing constraint. Each task is operating on one single voltage level and will not change during execution. This technique is called "**inter-task DVFS scheduling**". Different implementations of inter-task DVFS scheduling are shown in Yao et al. [YDS95], Schmitz et al. [SAHE02] ,Zhang et al. [ZHC02], Varatkar and Marculescu [VM03] and Gorji-Ara et al. [GACB+04].
On the other hand a lot of power management algorithms on OS-level consider the power saving potential within the task boundary. This technique is called "**intra-task DVFS scheduling**", and the operation voltage of a task is adjusted dynamically during its execution. Studies on intra-task DVFS scheduling are introduced by Shin et al. [SKL01], Seo et al. [SKC04], Shin and Kim [SJK05], and Oh et al. [OKKK08].
All these related works on OS-level concentrate their efforts on minimizing the power consumption of a system or to maximize the life time of mobile systems.

3.2.2 Power Management on OS-Level for Systems with Limited Power

Power Management on OS-level for systems with limited power is another form of OS-level power management. The challenge is to stay within a given power budget and ensure system stability. To the best of my knowledge no related work exists on power management on OS-level for systems with limited power.
Kansal et al. [KHZS07] show a power management for energy harvesting sensor networks. They introduce a power management policy for systems with limited power on hardware-level, which lowers the power consumption of the system according to the available power.

3.2.3 Summary of Proactive/Dynamic Power Management Techniques

Table 3.1 gives a summary to the discussed proactive/dynamic power management techniques. Here the original function of DVS and DVFS are shown.

Technique	**Function, Features**
Proactive/Dynamic Power Management (DPM)	Control of hardware components during run-time to minimize power consumption. DVS, DVFS, DPTC
Dynamic Voltage Scaling (DVS)	Adaption/Adjustment of the clock frequency during run-time.
Dynamic Voltage and Frequency Scaling (DVFS)	Adaption/Adjustment of the supply voltage and the clock frequency during run-time.
Dynamic Process and Temperature Compensation (DPTC)	Adjustment of the supply voltage and the clock frequency according to the temperature of a processor.
Task Rescheduling	Special scheduling algorithms to stay within a given power budget.

Table 3.1: Summary of the different proactive/dynamic power management techniques

Chapter 4

Design of the OS-Level Power Management

This chapter describes the concept of the chosen OS power management. The OS power management will handle the problem of limited power within an embedded system. This means that the OS power management will be responsible for staying within a given power budget. First an overview of the main parts is given, which is followed by a detailed description.

4.1 Overview

The concept is based on the general assumption that a given embedded system is supplied with limited power. A stable operation of the embedded system is essential, and therefore a given power budget must not be exceeded.
Figure 4.1 shows the main parts of the concept. The basic embedded system in this thesis is a LEON3 SoC, which is realized on an FPGA board. A power estimation unit is attached to the LEON3 SoC that estimates a real-time power profile. The required power management will be enabled through changes of the OS kernel and is therefore also called "OS-level" power management. Power information from the power estimation unit will be used to provide an OS-level power management, which is able to stay within a given power budget. The present power consumption as well as the given power budget are the key parameters.
Figure 4.2 illustrates the use-case diagram of the chosen power management technique. The operating system represents the system and the power management is the actor. The use-case of staying within power constraints is supported by the power budget and the power consumption of the SoC.

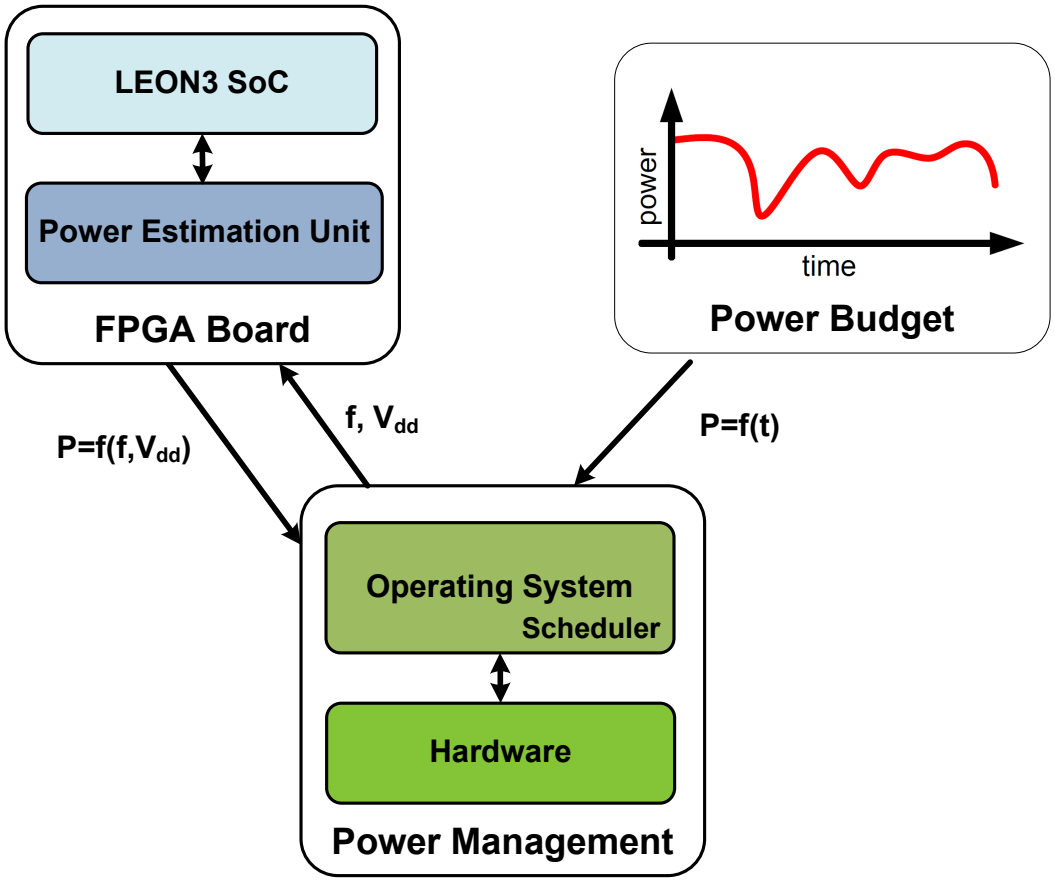

Figure 4.1: Concept overview

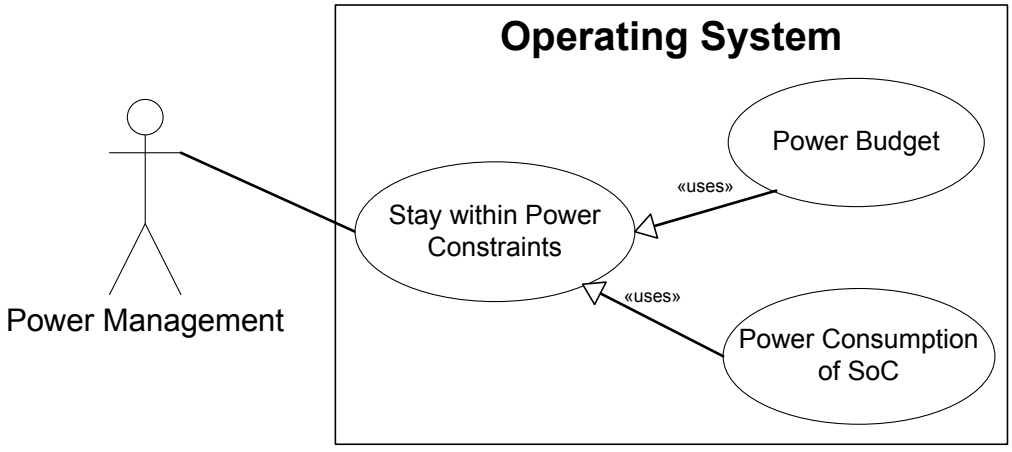

Figure 4.2: Use-case diagram of the chosen power management

A schematic overview of the used hardware and software parts is shown in Figure 4.3. The hardware part is based on a VHDL model, which contains the LEON3 SoC and the power estimation unit. This VHDL model is given as a list of vhd files. Based on the vhd files a net list is generated by the synthesis tool Xilinx ISE, which can be downloaded to the FPGA.
The basis for the software part is C-source code of a μC Linux OS. An image of the source code is generated by a μC Linux Cross-Compiler. After generation the tool GRMON is used to load the image to the LEON3 SoC platform. GRMON is also used for starting and for debugging the system.

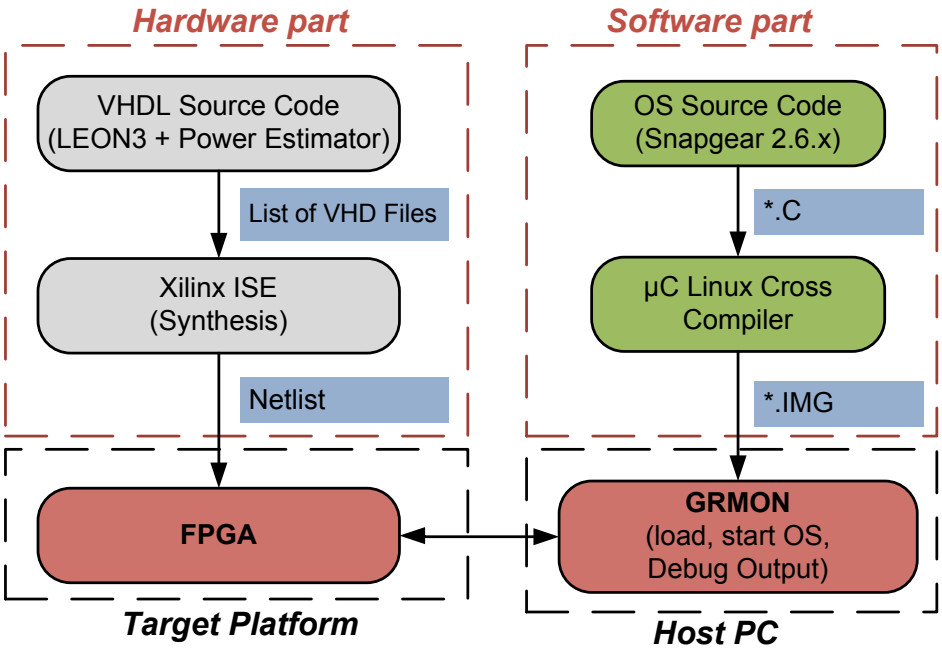

Figure 4.3: Schematic overview of the used hardware and software parts

4.2 SoC Platform

The basis of the used SoC platform is an FPGA board. The use of an FPGA development kit is very suitable during an implementation and evaluation cycle. This brings the following advantages:

- Development costs are decreased compared to an Application Specific Integrated Circuit (ASIC). No masks with very high fix costs are required.
- The time of implementation is very short.
- Less effort to realize corrections and extensions.

Disadvantages:

- A higher piece price if the amount of pieces is high.
- Maximum clock frequency of FPGAs is lower (typical 20 - 250MHz) than available clock frequency of ASICs (> 3GHz).
- The logical density is 10 times lower. This means that the required chip area is 10 times higher.
- A tendency to fewer functional tests in the forefield is typical, because of shorter design cycles and the possibility to correct errors very late.

All these disadvantages are maybe strong reasons in industry, but for prototype implementations a solution with an FPGA board is the most suitable.

4.2.1 LEON3

The LEON3 SoC is a 32-bit SPARC V8 architecture and supports multiprocessing with up to 16 CPU cores. The implementation of the cores can be done with the use of asymmetric multiprocessing (AMP) or synchronous multiprocessing (SMP). A full synthesis of the LEON3 SoC is also possible and the full source code is available under the GNU GPL license. The GNU GPL license gives the permission for evaluation, research and educational purposes. [AER05]
The features of the LEON3 SoC fulfill all requirements of this master thesis, and therefore a LEON3 SoC will be chosen as the target system. The LEON3 SoC will be used with one CPU core. It provides sufficient performance and is also adequate to determine the potential of the chosen power management. Furthermore, the complexity of the implementation is reasonable.

4.2.2 Power Estimation Unit

The power estimation unit provides the required power information to perform the chosen power management strategy. In our case this unit is a development of the Institute for Technical Informatics, Graz University of Technology [GBH+09]. It provides power values based on power profiling and allows real-time power analysis of the target system. The used estimation-based power profiling method generates its information by exploiting a power model of the target system. The complexity and the accuracy of the gained information depends on the abstraction layer of the used models. Low-level models are based on transistor- or gate-level. The used edition of the power estimation unit is situated on a higher layer with the advantages of a more compact model, real-time profiling and moderate area.

Power Model

Linear regression models are often used for power models on a high abstraction layer. The following equation shows a linear regression model.

$$y = \sum_{i=0}^{n-1} c_i \; x_i + \epsilon \tag{4.1}$$

x_i represent the system states (CPU - idle/run, memory access - read/write, etc.). The model coefficients c_i contain power information and a preliminary power characterization process is used for the initialization [BGS+10]. y represents the power estimate and ϵ shows the estimation error.

Power Emulation Architecture

Figure 4.4 shows the integration of the power model in hardware. The power emulation architecture includes power sensors, which observes state information of the system modules. Also the sta te information of a functional unit (CPU) can be considered for more accuracy. The power estimation unit accumulates the values of the power sensors and outputs a 32-bit power estimate $y(t)$ of the overall system. An averaging module is used for further post-processing and allows smoothing. The averaging is implemented as a configurable moving average filter according to Equation (4.2). The filter parameter N is used to configure the filter. The power information of the power estimation unit and of the averaging module are captured by the debug-trace generator and can be used for further processing.

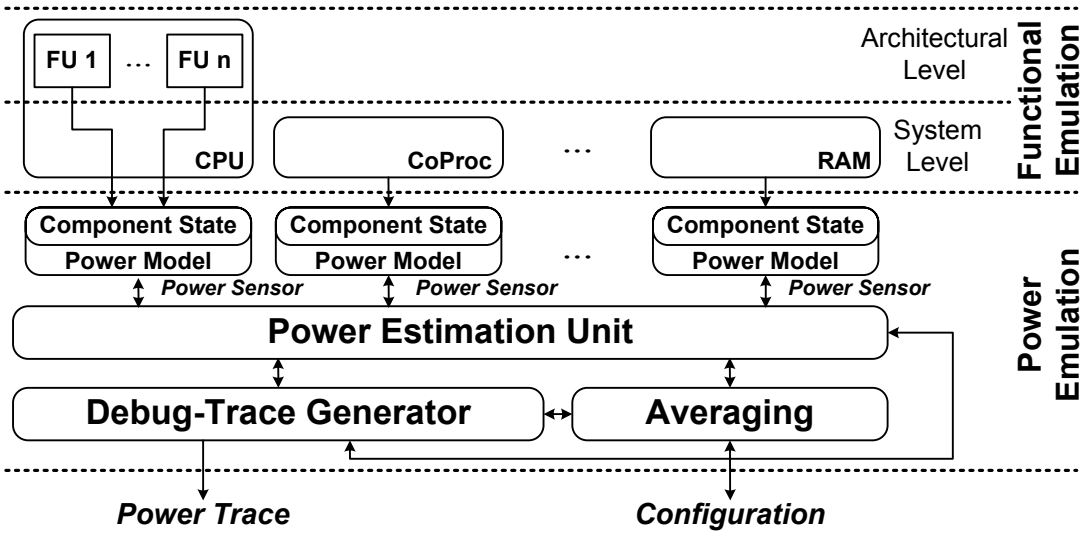

Figure 4.4: Power emulation architecture

$$y_{avg} = \frac{1}{N} \sum_{j=0}^{N-1} y(t-j) \tag{4.2}$$

4.3 Operating System

A big variety of OS for embedded systems exists due to many different application areas [Tur06]. On the one side there are commercial operating systems, and on the other side there are the open-source operating systems. Internally developed and commercial distributions of open-source operating systems offer an alternative. Commercial operating systems are often used because of their advantages: performance, real-time capability, compatibility with existing applications, support concerns, memory usage and legal ambiguity. The strengths of open-source operating systems are situated in the following areas: low costs, adaptability and extensibility. Also the evolution of the interest in open-source operating systems should be considered. The trend which OS are currently used for embedded systems is shown in Table 4.1.
In this master thesis an open-source solution is used. The chosen OS is a SnapGear μcLinux, which satisfies the low cost requirement, and compatibility to the chosen LEON3 SoC platform is also given.

Operating system currently use	**Current project**			
	2008	**2007**	**2006**	**2005**
Commercial OS	49%	47%	51%	55%
Open-source OS without commercial support	19%	22%	16%	25%
Internally developed or in-house OS	21%	21%	21%	20%
Commercial distribution of open-source OS	11%	10%	12%	-
Operating system plan to use next	**Next project**			
	2008	**2007**	**2006**	**2005**
Commercial OS	37%	41%	47%	50%
Open-source OS without commercial support	26%	27%	19%	34%
Internally developed or in-house OS	23%	15%	17%	16%
Commercial distribution of open-source OS	15%	16%	17%	-
	N=764	N=676	N=727	N=1303

Table 4.1: Trend of used OS for embedded systems

4.3.1 SnapGear Linux

The SnapGear μcLinux is a Linux distribution for embedded systems which supports the LEON3 SoC [AER05]. The purpose of the SneapGear μcLinux distribution is to provide fast developing of embedded Linux systems. The SneapGear μcLinux is available as a full source package. It contains kernel, libraries and application code.

Supported hardware:

- MMU
- LEON3 SMP
- GRETH 10/100/1000 Ethernet networking support
- SMC91x 10/100 Ethernet networking support
- OpenCores 10/100 Ethernet networking support
- PCI support
- GRETH over PCI
- ATA DMA and non-DMA
- Host USB 1.1 and/or 2.0
- I2C support
- SPI support

4.3.2 Linux 2.6 Scheduler

The scheduler is one of the most important parts within an OS. The goal of the scheduler is to guarantee a smooth run of all existing tasks [Aas05], [QW04]. This also includes a fair scheduling of the available computing time. The amount of the computation time depends on the priorities of the tasks. The Linux 2.6 scheduler is an O(1) process. This means that the run-time of the scheduler is constant and independent of the number of tasks. The core of the Linux 2.6 scheduling structure is based on an active and an expired run-queue per CPU (see Figure 4.5). Each run-queue contains 140 FIFO lists, which are used to handle the priorities of the existing tasks. At the beginning, all tasks are in the active run-queue and wait for their execution. If a task is scheduled next, it can be executed for a certain time period (timeslice). The task will be re-queued to the expired run-queue when its execution time is up. After some time all tasks in the active run-queue are executed and re-queued to the expired run-queue. Then the empty active run-queue will be swapped with the expired run-queue.

- **Timeslices**
  The timeslice of a task is a value which determines how long a task is allowed to run before interruption. After every interruption the timeslice of a task is calculated dynamically as a function of the associated priority. The minimum timeslice is 5ms, the default value is 100ms and the maximum value is 800ms.

- **Priorities**
  All tasks have a static priority. The priorities 0-99 are reserved for RT-tasks and 100-139 for normal tasks. The priority of normal tasks is also called “nice” value with the range -20 to 19. Each priority stands for an appropriate length of its timeslice.

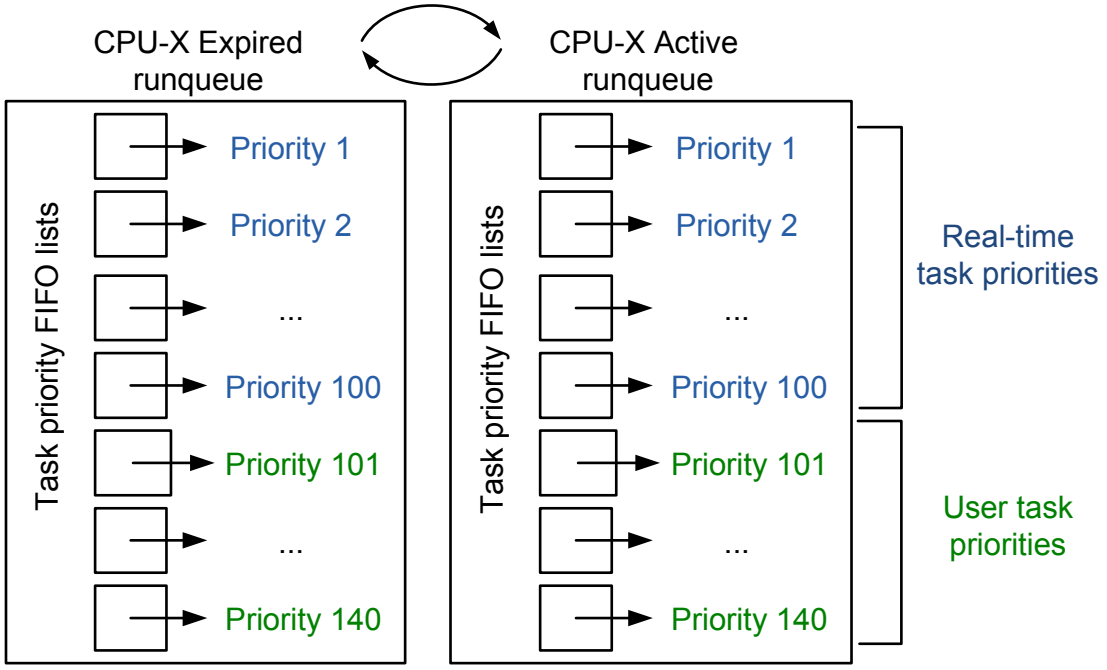

Figure 4.5: Scheduler core

Additionally, dynamic priority changing is allowed. This means that the scheduler is able to increase the priority of I/O-bound tasks and to decrease the priority of CPU-bound tasks. The maximum value of a dynamic increase or decrease is 5.

- **Scheduling policies**
  The Linux 2.6 scheduler provides four scheduling policies for a single task:
  - **SCHED_FIFO**: This scheduling policy is used for RT-tasks. Such a task is allowed to run as long as necessary. The execution will be finished only if an interrupt happens or the task is completed.
  - **SCHED_RR** : This scheduling policy is also used for RT-tasks. A task will be interrupted after its timeslice is up. Then it will be re-queued in the **active** run-queue (on tail of its priority list). The task will be executed next time after all tasks with the same priority get some execution time.
  - **SCHED_NORMAL**: Normal Round-Robin scheduling. A task will be interrupted after its timeslice is up. Then it will be re-queued in the **expired** run-queue. The task will be executed next time after all other tasks get some execution time.
  - **SCHED_BATCH**: Similar to the normal scheduling. It is used for CPU-bound tasks. Scheduler awards bigger penalties to these tasks.
- **Data structure of a task**
  All information for a task is defined in the structure "**task_struct**". Also the information for the scheduling process is defined in this structure. The following

information is important for the scheduling process:

Listing 4.1: Structure of a task

```
/*
 Structure of a task
 */
struct task_struct {
  ...
  volatile long state;
  ...
  int prio, static_prio, normal_prio;
  struct list_head run_list;
  struct prio_array *array;

  unsigned long sleep_avg;
  unsigned long long timestamp, last_ran;
  unsigned long long sched_time; /* sched_clock time spent running */
  enum sleep_type sleep_type;

  unsigned long policy;
  cpumask_t cpus_allowed;
  unsigned int time_slice;
  ...
  pid_t pid;
  ...
};
```

Listing 4.1 shows the structure of a task:

- **state** shows the present state of the task ()
- **prio** represents the dynamic priority, **static_prio** the static priority and **normal_prio** the priority after penalty calculation
- **run_list** and **array** are necessary for the management of the task
- **sleep_avg** shows the average duration of sleep time of a task
- **policy** gives information of the current scheduling policy of the task
- **time_slice** represents the rest of execution time
- **pid** shows the process ID of the task

This scheduler will be adapted to implement the chosen power management. The scheduling process is defined in the source files: */kernel/sched.c* and */includes/linux/sched.h*

4.4 Power Management

The practical work of this master thesis is to develop an OS-level power management, which exploits supporting power information from a power estimation unit. The purpose of the power management is to find a balance between an available power budget on the one side and the required power of an embedded system on the other side. Two countermeasures are available to ensure a reliable system operation.

- **Task rescheduling**
  If the required power consumption of the next task is higher than the available power, then the task will not run next. Instead, another task with a lower power consumption comes next.
- **DVFS**
  The clock frequency of the SoC will be decreased for a reduction of the overall power consumption.

The following steps will be the realization of an appropriate OS-level power management (Figure 4.6):

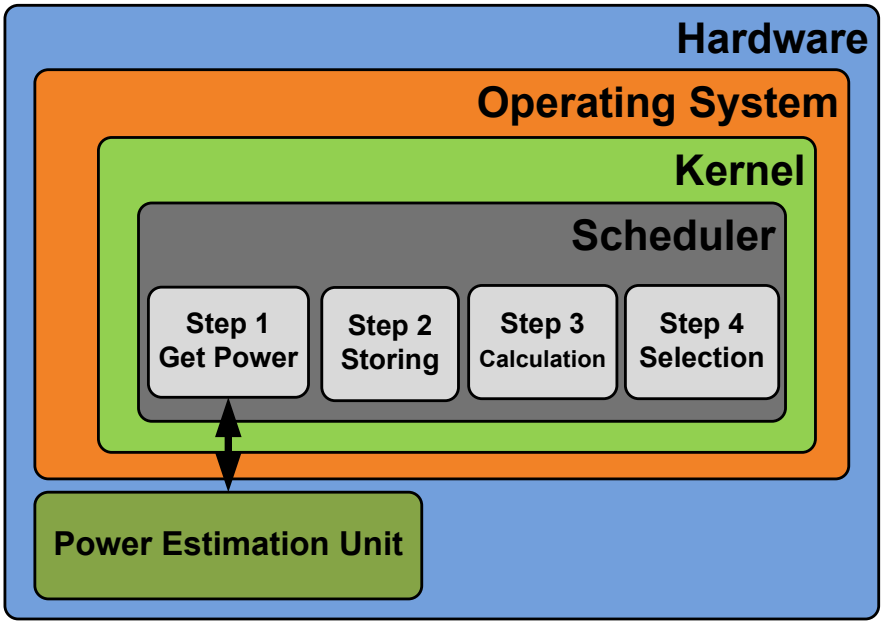

Figure 4.6: Overview of the four steps of the power management

- **Acquisition of the present power information from the power estimation unit**
  Power information of the target system is generated by the power estimation unit and will be provided to the OS. The power information contains the average power consumption of timeslice (Ts_{ji}) of a task (T_j). This works as follows: the OS triggers

a control signal (pe_ctrl) and the power estimation unit begins to calculate an average of the present power consumption over the period of one timeslice. The result is the average power consumption of the chosen time period (Figure 4.7). A schematic overview of the communication between OS and the power estimation unit is shown in Figure 4.8.

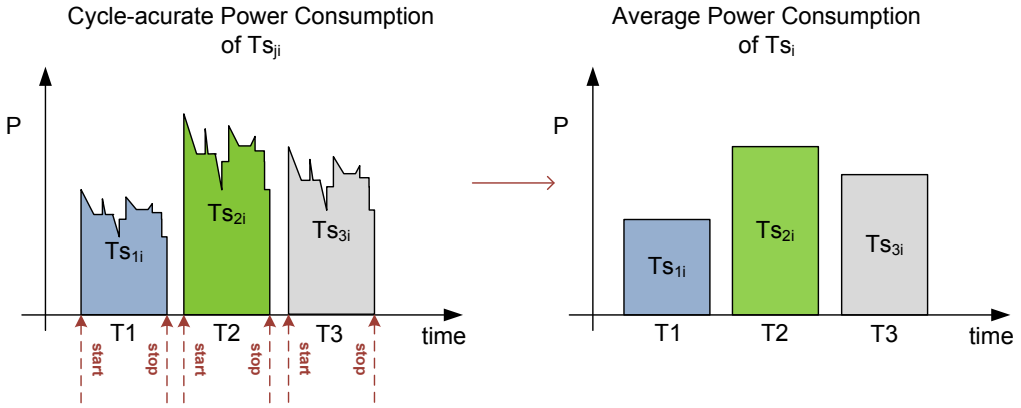

Figure 4.7: Power information generated from the power estimation unit

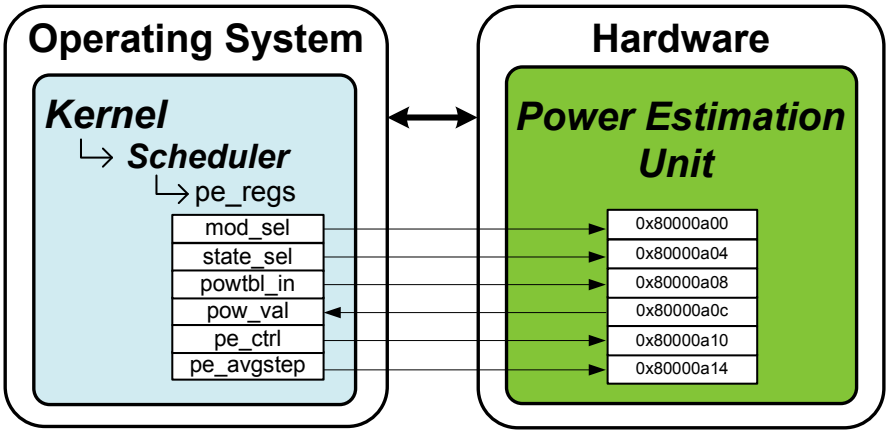

Figure 4.8: Overview of the communication between the OS and the power estimation unit with the following registers: **mod_sel** - register for the module selection, **state_sel** - register for the state selection, **powtbl_in** - register for power table configuration, **pow_val** - present power consumption, **pe_ctrl** - control register, **pe avgstep** - register to define the average step range

- **Storing of the power information**
  After the OS has applied the average power consumption of timeslice (Ts_{ji}) of a task (T_j), it is necessary to store this information. Therefore, the structure of a task will be extended with power information. Figure 4.9 shows the extension of the task structure to include power information.

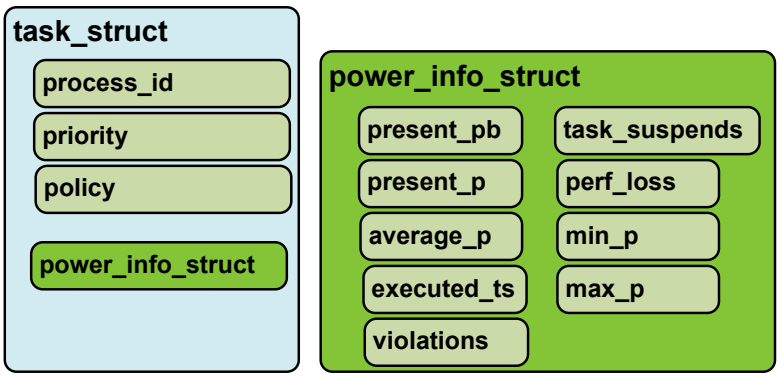

Figure 4.9: Extended task structure

  - Values for the present power state of a task
    - Present power budget (present_pb) → $PB(Ts_{ji})$
    - Present power consumption (present_p) → $P(Ts_{ji})$
    - Average power consumption (average_p) → $P_{AVG}(T_j)$
  - Values for the overall power state of a task
    - Presently executed timeslices (executed_ts)
    - # of occurred power budget violations (violations)
    - # of task suspends (task_suspends)
    - performance loss (perf_loss)
    - minimum/maximum power consumption (min_p/max_p)

- **Calculation of the present average power consumption**
  After storing the average power consumption of timeslice (Ts_{ji}) of a task (T_j), the overall average power consumption will be calculated and updated. For updating the average power consumption, three different averaging algorithms are explored (see 4.4.1). This means that several timeslices are used to calculate the average power consumption of the present task. Figure 4.10 gives an example.

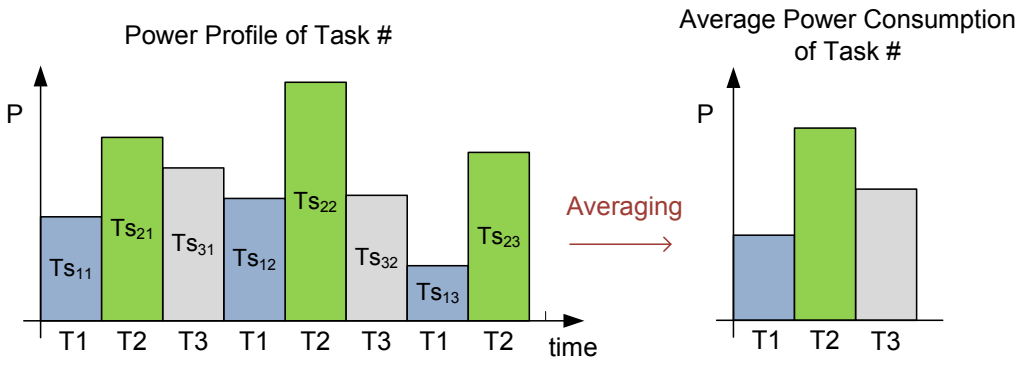

Figure 4.10: Update power information with further timeslices

- **Selection of the next running task**
  The previously calculated average power consumption is the basis for the scheduling decision. If a task is allowed to run next depends on its average power consumption in the past and on the present power budget. If the next task in the run queue exceeds the available power, it is not allowed to run next. Rescheduling will be done as long as a suitable task is found in the run queues. If no task is able to fulfill the present power constraints, the processor will be idle until the available power is high enough again. Figure 4.11 shows an example of the selection process:

  - t=1: Power consumption of Ts_{11} of T_1 is lower than the present power budget
    $\rightarrow T_1$ will be scheduled
  - t=2: Power consumption of Ts_{21} of T_2 is lower than the present power budget
    $\rightarrow T_2$ will be scheduled
  - t=3: Power consumption of Ts_{31} of T_3 and Ts_{12} of T_1 are higher than the present power budget
    $\rightarrow T_3$ and T_1 will be suspended; T_2 will be scheduled
  - t=4: Power consumption of Ts_{31} of T_3 is higher than present power budget
    $\rightarrow T_3$ will be suspend; T_1 will be scheduled
  - t=5: Power consumption of Ts_{31} of T_3 is lower than present power budget
    $\rightarrow T_3$ will be scheduled
  - t=6: No task is able to fulfill the present power budget - IDLE timeslice
  - t=7: Power consumption of Ts_{32} of T_3 is lower than present power budget
    $\rightarrow T_3$ will be scheduled

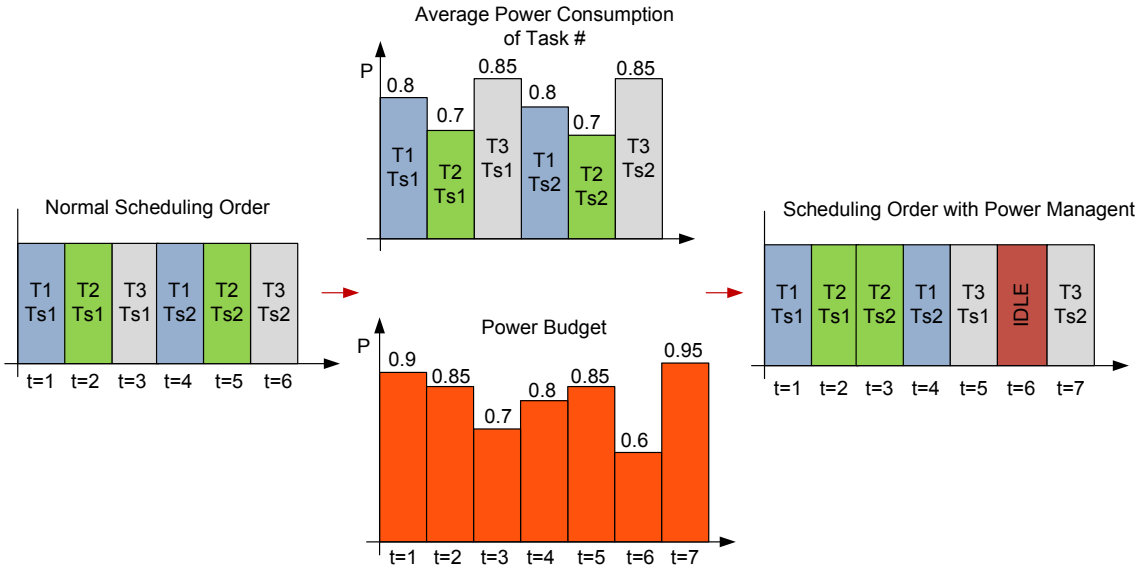

Figure 4.11: Selection process of the next running task

4.4.1 Averaging Algorithms

The used algorithms for the calculation of the average power consumption of a task are variations of moving average filters. They are used because of their potential to give information about the future power consumption of a task based on past power values. Moving average filters are often used in the field of time series data analysis to smoothen short-time variations or to emphasize long-time trends. Typical applications can be found in signal processing (low-pass-filter), financial analysis (stock market price) or in the field of statistics.[BV08]

Simple Moving Average (SMA)

A SMA filter calculates the mean value of a certain number of data points. The weight of every single data point is equal and defined with the factor 1 (Figure 4.12). The number of required data points is defined with N. For the calculation of the present average value (P_{SMA_n}), the present power value (P_n) and the N-1 values ($P_{n-1}, P_{n-2}, ...P_{n-(N-1)}$) are used. The following equation describes the SMA filter:

$$P_{SMA_n} = \frac{1 \cdot P_n + 1 \cdot P_{n-1} + 1 \cdot P_{n-2} + ... + 1 \cdot P_{n-(N-1)}}{N} = \frac{1}{N} \sum_{i=0}^{N-1} P_{n-i} \tag{4.3}$$

- **Pseudo code**

```
SMA_Algorithm (data_points[] , N)
begin
    sum = 0
    for i = 0 to N-1
    begin
        sum = sum + data_points[i]
    end
    average = sum / N
    return average
end
```

- **Optimization**
  A more efficient implementation is to drop out the oldest data point and to update the average with the new data point:

$$P_{SMA_n} = P_{SMA_{n-1}} - \frac{P_{n-N}}{N} + \frac{P_n}{N} \tag{4.4}$$

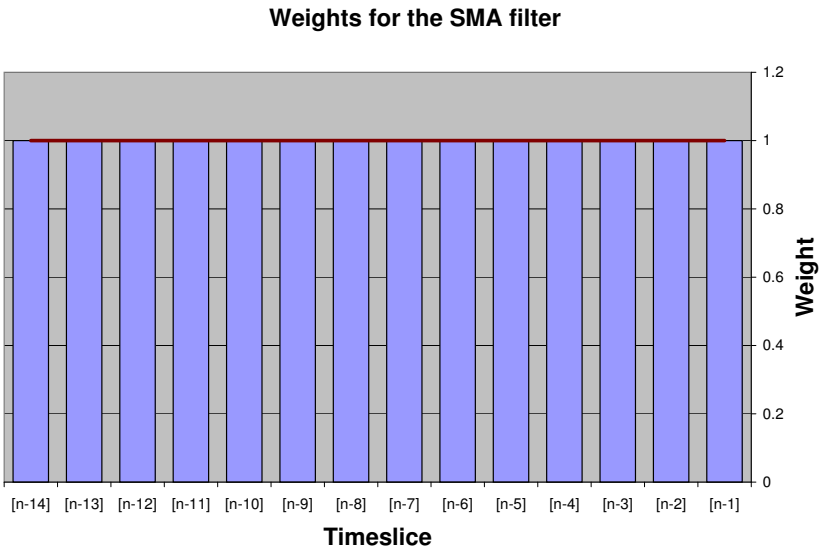

Figure 4.12: Weights for the SMA filter

Weighted Moving Average (WMA)

The WMA filter is a variation of the SMA filter. It uses different weights (Figure 4.13) for every single data point. The following equation describes the function of a WMA filter:

$$P_{WMA_n} = \frac{N \cdot P_n + (N-1)P_{n-1} + (N-2)P_{n-2} + ... + P_{n-(N-1)}}{N + (N-1) + (N-2) + ... + 2 + 1} = \frac{2}{N(N+1)} \sum_{i=0}^{N-1} (N-i)P_{n-i} \tag{4.5}$$

with

$$N + (N-1) + (N-2) + ... + 2 + 1 = \frac{N(N+1)}{2} \tag{4.6}$$

- **Pseudo code**

```
WMA_Algorithm (data_points[] , N)
begin
    numerator = 0
    for i = 0 to N-1
    begin
        numerator = numerator + (N-i) * data_points[i])
    end
    denominator = N * (N-1)

    average = numerator / denominator
    return average
end
```

- **Optimization**
  A more efficient implementation for the WMA filter exists. It works with a simple update of the last average. The following equations describe the optimization:

$$TotalP_n = P_n + P_{n-1} + ... + P_{n-(N-1)} \tag{4.7}$$

$$TotalP_{n+1} = TotalP_n - P_{n-N} + P_{n+1} \tag{4.8}$$

$$Numerator_1 = P_1 \qquad Noun = \frac{N(N+1)}{2} \tag{4.9}$$

$$Numerator_{n+1} = Numerator_n + N \cdot P_{n+1} - TotalP_n \tag{4.10}$$

$$P_{WMA_n} = \frac{Numerator_n}{Noun} \tag{4.11}$$

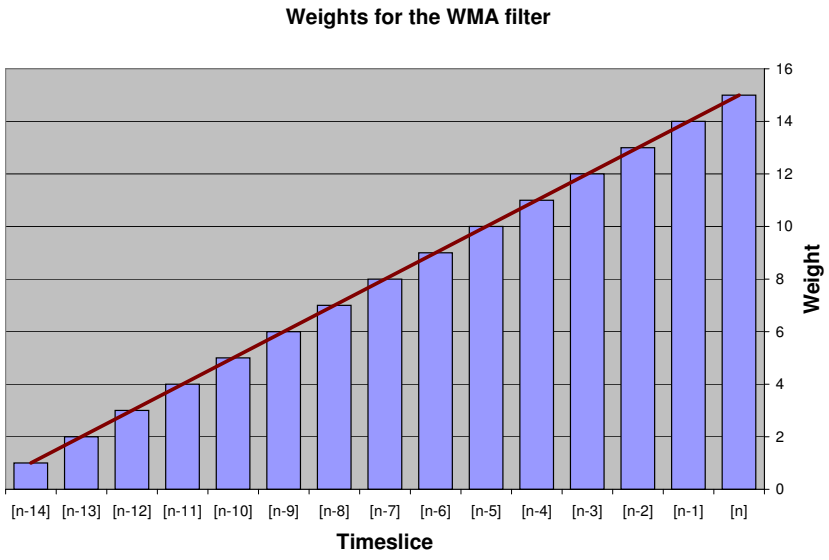

Figure 4.13: Weights for a WMA filter with N=15

Exponential Moving Average (EMA)

Similar to the WMA filter, the EMA filter also puts different weights on every single data point. The difference to the WMA filter is the exponential weighting of data points (Figure 4.14). To parameterize an EMA filter the value α is used. This parameter can take a value between 0 and 1. Typically, the EMA filter will be implemented in a recursive form. The following equations define the EMA filter:

$$P_{EMA_1} = P_1 \tag{4.12}$$

$$P_{EMA_{n+1}} = \alpha \cdot P_{n+1} + (1 - \alpha) P_{EMA_n} \tag{4.13}$$

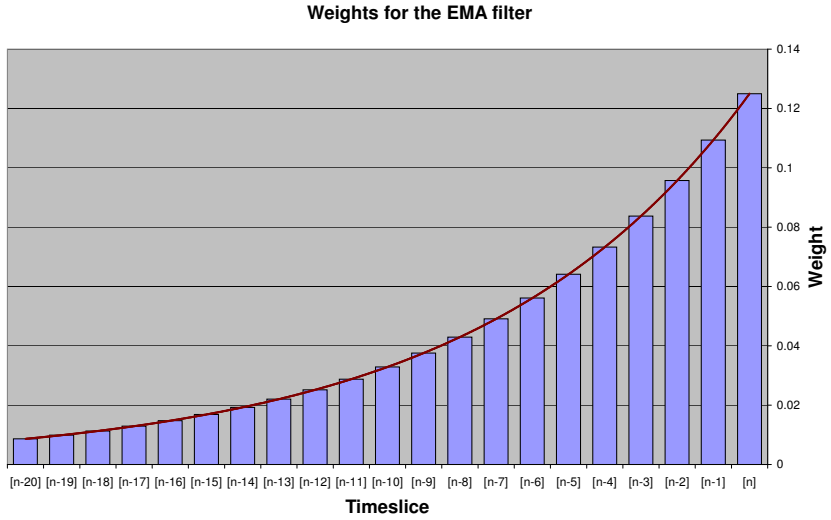

Figure 4.14: Weights for the first 20 values of the EMA filter with $\alpha = 1/8$

4.5 Power Budget

The power budget is chosen arbitrarily and exists as an array of data values. It is a guideline so that the power level will not change during a timeslice. It will be assumed that a minimum power level is guaranteed.
Normally, the value of the present available power would be provided by hardware sensors, which detect the available power from kinetic, electromagnetic or thermal sources. A mixture of these sources is possible. A practical application would also contain an energy buffer for energy storage. This could be realized in form of accumulators or super capacitors.

Chapter 5

Implementation of the OS-Level Power Management

This chapter shows the implementation of the concept. First the hardware and the software setup of the LEON3 SoC platform will be explained. The sections of hardware setup and software setup give an overview of the used tools and their configuration. For further information the relevant manuals and quick guides are recommended [AER10], [XIL10]. Then the necessary modifications for the implementation of the chosen power management will be shown. This addresses the OS, especially the scheduling mechanism.

5.1 Hardware Setup

The the LEON3 SoC is implemented on a FPGA board (Xilinx XC3S-2000) an its basis is the VHDL code of a LEON3 SoC. The given VHDL code also includes the used power estimation unit. The following tools are necessary for the hardware setup:

- **Xilinx ISE (10.1)** - FPGA-Synthesis Tool to yield a FPGA executable system from VHDL code.
- **Xilinx Impact** - OS Netlist Programming GUI for the FPGA

After installation of these tools the following steps are taken for the hardware setup (Figure 5.1).

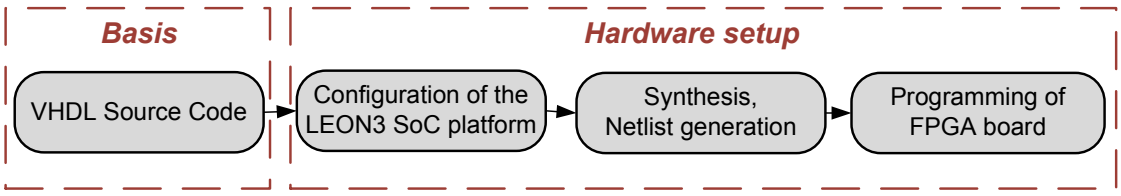

Figure 5.1: Overview of the steps for the hardware setup

5.1.1 Configuration of the LEON3 SoC Platform

The configuration of the LEON3 SoC platform is done with the command "**make xconfig**" in the appropriate "<grlib>/<design>" folder. This command starts a GUI for the configuration (Figure 5.2). The following settings are important to enable:

- Synthesis → FPGA-type: adapt to used board (e.g. xc3s-2000)
- Processor → DebugSupportUnit: enable LEON3 debug support unit; instruction trace buffer - Debug information is accessible on the host PC.
- Processor → Floating-point unit: disable - Floating point unit is licensed and not necessarry.
- Debug link: enable serial debug link, JTAG debug link - Enables the communication between the LEON3 SoC and the host PC.

After finishing the configuration, the command "**make scripts**" prepares the necessary scripts for the subsequent synthesis.

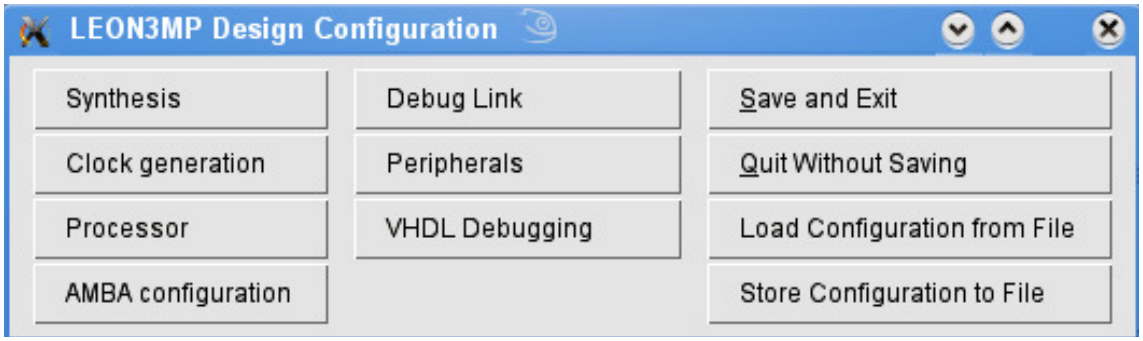

Figure 5.2: Snapshot of LEON3 configuration GUI

The LEON3 SPARC V8 processor includes the following units:

- Integer unit
- Cache system
- Memory management unit
- Debug support unit

5.1.2 Synthesis, Netlist Generation

After a successful configuration the synthesis will be started with the command "**make ise**". The result of the synthesis is a corresponding netlist, which is used in the next step for programming the FPGA board. See [AER10] and [XIL10] for further information to the used commands.

5.1.3 Programming of the FPGA Board

The last step of the hardware setup is the programming of the FPGA board. The command "**impact**" starts a GUI, which is used for the programming process. After loading the appropriate mcs/bit-files to the three units (mcs to xcf04s units, bit to xc3s2000 unit), the programming can be started. If the LEON3 design is loaded from the host PC, xcf04s units can be bypassed, only the FPGA (xc3s2000) bit-file is to be loaded. The xcf04s units are flash memory, which are used to store configuration data for a stand-alone usage of the system. Figure 5.3 gives a snapshot of the programming GUI.

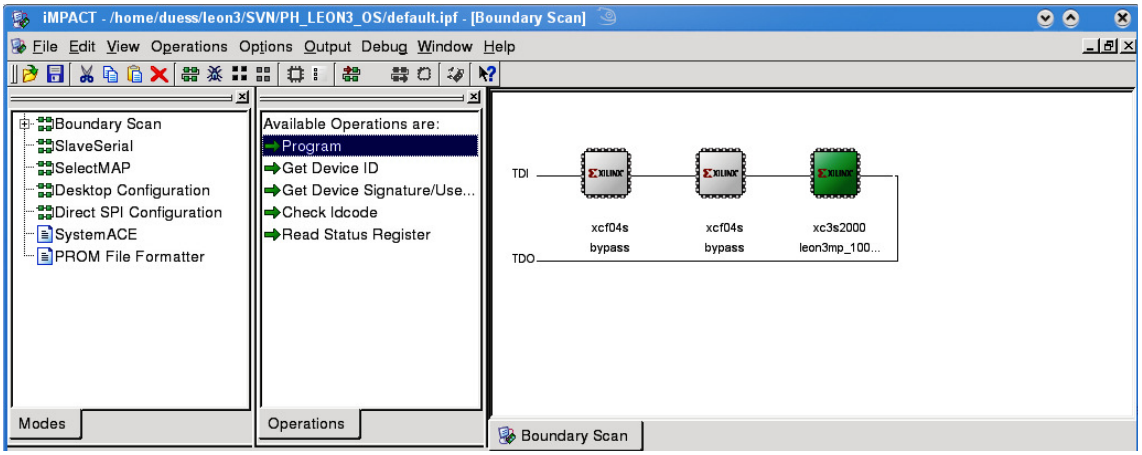

Figure 5.3: Snapshot of LEON3 programming GUI

5.2 Software Setup

It is also necessary to install an OS on the LEON3 SoC platform. The basis is C source code. The following tools are needed for a successful software setup:

- **sparc-uclinux-gcc compiler** - Tool for snap-gear OS compilation
- **GRMON** - Tool for the communication between the host PC and the LEON3 SoC platform

The following steps show the process from C source code to a running operating system (5.4):

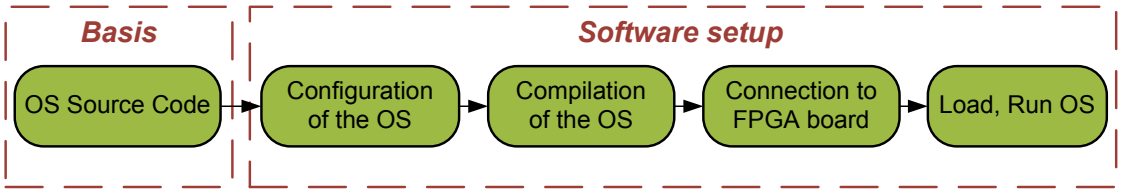

Figure 5.4: Overview of the steps for the software setup

5.2.1 Configuration of the OS

The configuration of the OS is done in two parts. First the general configuration GUI will be started with the command "**make xconfig**" in the "<snapgear>" folder and the option of "Customize Kernel Settings" will be enabled (Figure 5.5). After "Save and Exit" of the current GUI, the kernel configuration GUI will be started automatically. A snapshot of the kernel configuration GUI is shown in Figure 5.6.

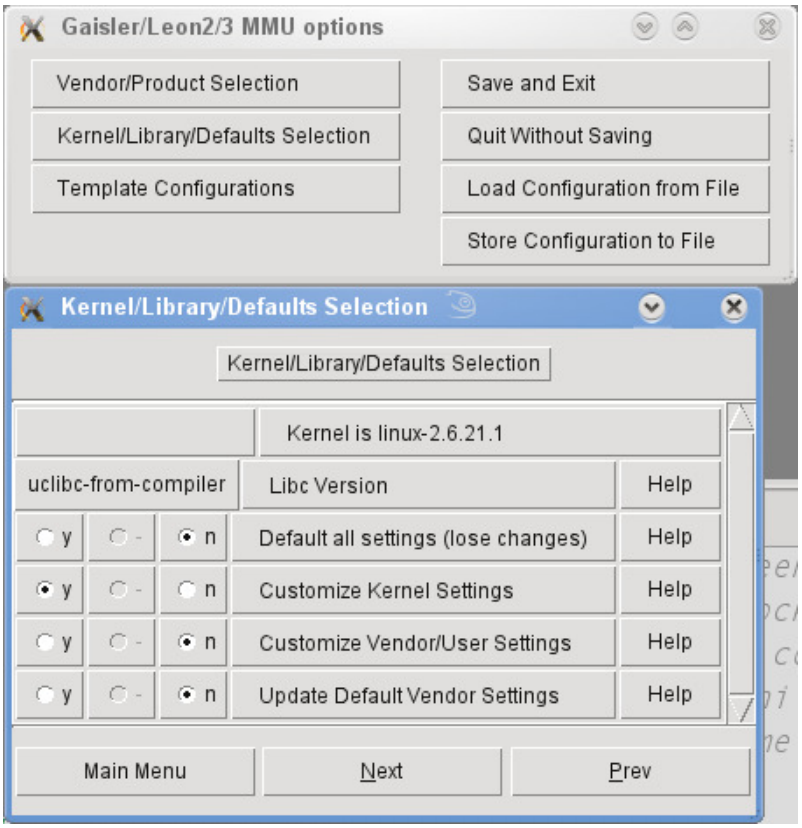

Figure 5.5: Snapshot of the OS configuration GUI

5.2.2 Compilation of the OS

After successful configuration of the OS, the compilation will be started with the command "**make**". The result of the compilation is a runnable image of the OS.

5.2.3 Connection to the LEON3 SoC Platform

For the connection between the host PC and the LEON3 SoC platform a "XILINX Platform Cable USB II" and the tool "GRMON" are used. The connection will be established with the command "**grmon-eval -xilusb -nb -u**".

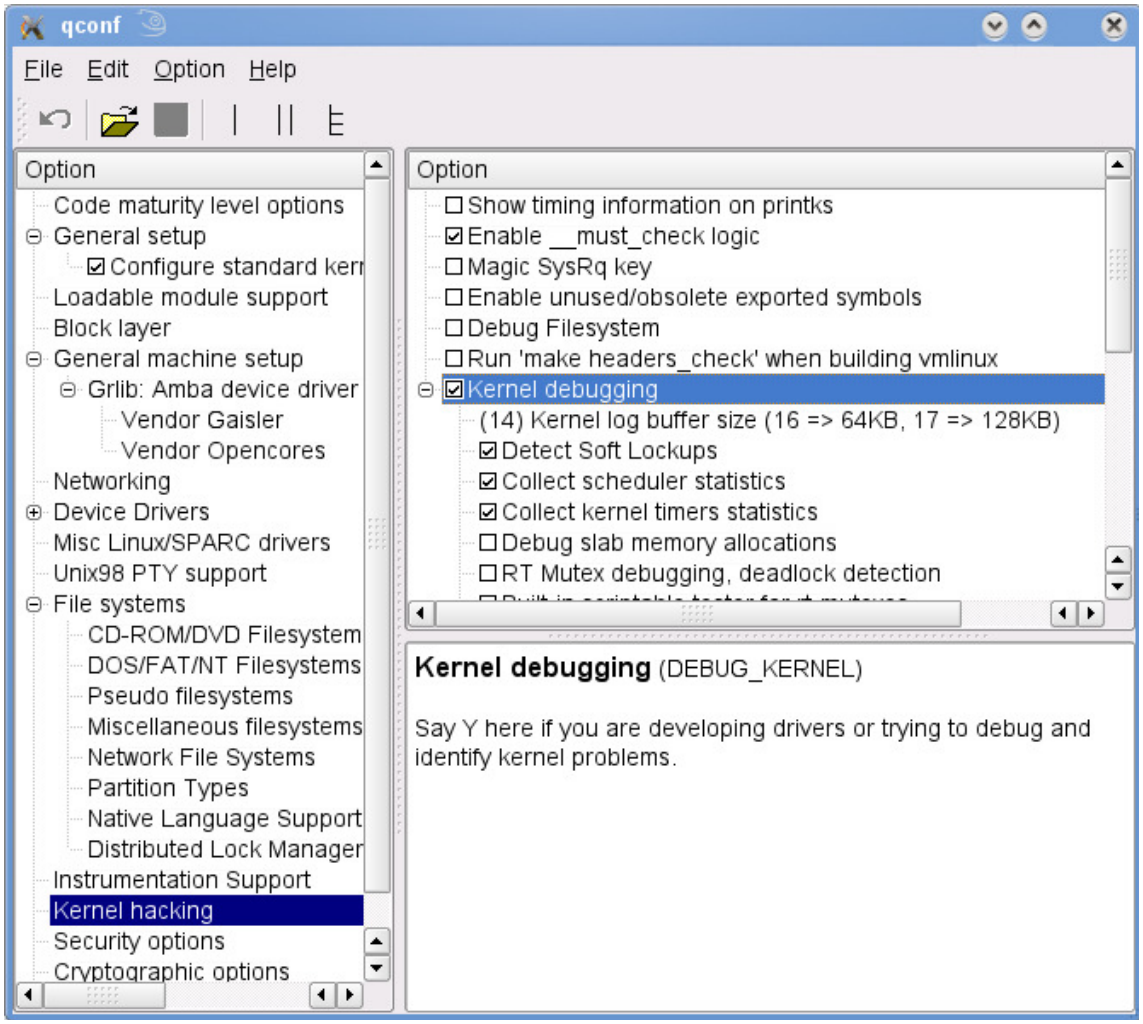

Figure 5.6: Snapshot of the OS kernel configuration GUI

5.2.4 Load, Run OS

After a connection between the LEON3 SoC platform and the host PC has been established, the OS images will be loaded with the command "**load images.dsu**". Then the OS will be booted with the command "**run**".

5.3 Linux Scheduler

All modifications for establishing of the power management are made in the files "/kernel/sched.c" and "/includes/linux/sched.h". The include file "sched.h" is extended with the structure for the power information and also the parameters for the configuration of

the power management are situated there. The "sched.c" file is normally responsible for the scheduling procedure. Now this file is extended with the implementation of the power management.
In the concept the power management was divided into four subtasks:

- Acquisition of the present power consumption from the power estimation unit
- Storing of the power information
- Calculation of the present average power consumption
- Selection of the next running task

Also functionalities for the simulation of the power budget and for writing the power information to a file have been implemented.

5.3.1 Acquisition the Present Power Information from the Power Estimation Unit

The basic information for the power management is the present power consumption of the used LEON3 SoC platform. This information is provided by the power estimation unit, which is also placed on the SoC platform. Before the first read of power consumption can happen, an initialization of the power estimation unit is necessary. As mentioned in the concept, the power estimation unit is able to provide power information, which is the average power consumption of a given time range. The time range will be chosen to be as long as one single timeslice of a task. The following code snippets describe the procedure of initialization and reading.

Listing 5.1: Structure for the communication with the power estimation unit

```
1  /*
2   * Structure for power estimation unit
3   */
4  struct pe_regs_t {
5    volatile int mod_sel;
6    volatile int state_sel;
7    volatile int powtbl_in;
8    volatile int pow_val;
9    volatile int pe_ctrl;
10   volatile int pe_avgstep;
11 };
```

Listing 5.1 shows the structure for the communication with the power estimation unit:

- **mod_sel**: register (32-bit) for the module selection of the power estimation unit.
- **state_sel**: register (32-bit) for the state selection of the power estimation unit.
- **powtbl_in**: register (32-bit) for power table configuration of the power estimation unit.

- **pow_val** is the value of the present power consumption.
- **pe_ctrl** is the control register (32-bit) of the power estimation unit, Bit 0 → global enable of power estimation unit, Bit 1 → enable/disable averaging, Bit 2 → standard averaging/coarse-grained averaging.
- **pe_avgstep**: register (32-bit) to define the average step range if coarse-grained averaging is enabled. Then the power estimation unit provides the average power of a timeslice. The duration of a timeslice is defined with 100ms. pe_avgstep is calculated as followed:

$$pe\_avgstep = \frac{Timeslice}{Tclk * Avg\_Array\_Size} = \frac{100 \cdot 10^{-3}s}{25 \cdot 10^{-6}s * 128} = 31250 \tag{5.1}$$

$$Timeslice: \ 100ms, \ size\ of\ averaging\ array: \ 128$$
$$clock\ frequency = \ 40MHz \rightarrow Tclk = \ 25 \cdot 10^{-6}s$$

Listing 5.2: Acquisition of the present power information from the power estimation unit

```
1  /*
2   * Acquisition of the present power information from the power estimation
        unit
3   */
4  static int read_power_info()
5  {
6    static struct pe_regs_t *pe_regs;
7    static int first=0;
8    if(!first)
9    {
10     pe_regs = ioremap(0x80000a00, 24);
11     pe_regs->pe_avgstep=32150;
12     pe_regs->pe_ctrl=0x00000007;
13     first=1;
14   }
15   return (pe_regs->pow_val);
16 }
```

Listing 5.2 shows the acquisition of the present power information from the power estimation unit. If the function "read_power_info" is called the first time, the initialization of the power estimation unit takes place (**line 8**). **Line 10** maps the data structure of pe_regs to the memory-mapped area of the LEON3 SoC where the power estimation unit is located. **Line 11** sets the average step (one timeslice of a task equals 200ms). **Line 12** sets Bit 0,1,2 of pe_ctrl, which globally enables power estimation, enables averaging and enables coarse-grained averaging. **Line 15** returns the average power consumption of the chosen time range.

Listing 5.3: Call of the "power information reading" function

```
1  /*
2   * This function gets called by the timer code, with HZ frequency.
3   * Interrupts are disabled.
4   */
5  void scheduler_tick(void)
6  {
7    ...
8    p->power_info.power= read_power_info()*1000;
9    ...
10 }
```

Listing 5.3 shows the call of the "power information reading" function. **Line 8** reads the present power information and stores it to the power variable of the power_info structure of a task. This reading of power information is done as early as possible, in order to influence the power consumption of a task as little as possible with the scheduling procedure.

5.3.2 Storing of the Power Information

After reading the power information from the power estimation unit, the information is stored in the "power_info" structure. Each "task" is extended with such a "power_info" structure. This structure also includes variables for later evaluation and for the different calculation possibilities of the average power consumption (Section 4.4).

Listing 5.4: Extension of the task structure with power information structure

```
1  /*
2   Structure of a task
3   */
4  struct task_struct {
5  ...
6    struct power_info_struct power_info;
7  ...
8  };
9
10 /*
11  Structure for storing power information of a task
12  */
13 struct power_info_struct{
14
15   //Values for the present power state
16   int current_budget;
17   int power;
18   int moving_average;
19
20   //Values for calculation of moving average
21   int ringbuffer[SIZE_OF_RB];
22   int timeslice_cnt;
23   int totalP;
24   int numerator;
25   int noun;
```

```
26
27    //Values for the overall power state of a task
28    int violations;
29    int not_scheduled;
30    int min_power;
31    int max_power;
32    int performance;
33
34    //Values for exact evaluation
35    int butget_values[MAX_TIMESLICES];
36    int power_values[MAX_TIMESLICES];
37    int moving_average_values[MAX_TIMESLICES];
38    int vdd[MAX_TIMESLICES];
39    int f[MAX_TIMESLICES];
40    int timestamp_start[MAX_TIMESLICES];
41    int write_offset;
42  };
```

Listing 5.4 shows the extension of the task structure with the power information structure. **Lines 16-18** store the values for the present power state (power budget, power consumption and average power consumption) of a task. The variables from **line 21-25** are used for the calculation of the next average power consumption of a task. **Lines 28-32** represent the overall power state of a task. It shows how many power violations occurred, how often the power management has intervened (not_scheduled), the performance loss and the minimum/maximum power consumption of a task. **Lines 35-41** are responsible for an exact evaluation of the task. The power information of each single timeslice is stored.

5.3.3 Calculation of the Present Average Power Consumption

The updating process of the average power consumption of a task is one of the main parts of the power management. Three different moving average filters are implemented: SMA, WMA, EMA. Each of these filter algorithms have some configuration parameters. Before starting, one of the algorithms must be chosen and the according parameters must be set up. The following code snippets describe the initialization parameters and the different moving average filter algorithms.

Listing 5.5: Initialization of the averaging parameters

```
1  #define SIMPLE_MA           0
2  #define WEIGHTED_MA         1
3  #define EXPONENTIAL_MA      2
4  #define MOV_AV_FILTER       EXPONENTIAL_MA
5  #define SIZE_OF_RB          5
6  #define EMA_ALPHA           100
7  #define MAX_TIMESLICES      100
8  #define DEFAULT_AVERAGE     150000
```

Listing 5.5 shows the initialization parameters of the averaging filters. **MOV_AV_FILTER** represents the filter algorithm, which is used for the calculation of the moving average. **SIZE_OF_RB**: the set-up parameter for the simple and the weighted moving average. It

also defines how many past power values are used for the average calculation.
EMA_ALPHA: the set-up parameter for the exponential moving average algorithm.
MAX_TIMESLICES defines how many timeslices will be logged.
DEFAULT_AVERAGE defines the default average power consumption of a task.

Listing 5.6: Call of the "calculating moving average" function

```
1 static void task_running_tick(struct rq *rq, struct task_struct *p)
2 {
3   ...
4   if (!--p->time_slice) {
5       p->power_info.moving_average=calculate_moving_average(p);
6       ...
7   }
8   ...
9 }
```

Listing 5.6 shows the call of the "calculating moving average" function. The condition of **line 4** is fulfilled if the timeslice of a task is consumed. Then the average power consumption will be updated and stored to the power information structure of the task (**line 5**).

Listing 5.7: Calculation of the moving average

```
1  static int calculate_moving_average(struct task_struct *p)
2  {
3    int moving_average=0;
4    int moving_average_old;
5    int power=0;
6    int idx=0;
7    int timeslice=0;
8    int *rb;
9
10   if(p->power_info.power>p->power_info.current_budget)
11     p->power_info.violations++;    //Violation occurs
12
13   power=p->power_info.power;
14   if (p->power_info.timeslice_cnt==1)
15     p->power_info.min_power=power;
16   else if(power<p->power_info.min_power)
17     p->power_info.min_power=power;
18
19   if(power>p->power_info.max_power)
20     p->power_info.max_power=power;
21
22   p->power_info.timeslice_cnt++;
23   timeslice=p->power_info.timeslice_cnt;
24   rb=&(p->power_info.ringbuffer);
25   idx= (timeslice-1)%SIZE_OF_RB;
26
27   switch(MOV_AV_FILTER)
28   {
29     case SIMPLE_MA: //Simple moving average
30       moving_average_old=p->power_info.moving_average;
```

```
31        if(timeslice<=SIZE_OF_RB)
32        {
33          moving_average_old=moving_average_old*(timeslice-1);
34          moving_average=(moving_average_old+power)/timeslice;
35        }
36        else
37          moving_average=moving_average_old+((power-rb[idx])/SIZE_OF_RB);
38
39        rb[idx]=power;
40      break;
41
42      case WEIGHTED_MA: //Weighted moving average
43        if(timeslice<=SIZE_OF_RB)
44        {
45          p->power_info.noun=p->power_info.noun+timeslice;
46          p->power_info.numerator=p->power_info.numerator+(timeslice*power);
47          p->power_info.totalP=p->power_info.totalP+power;
48          moving_average= (p->power_info.numerator)/(p->power_info.noun);
49        }
50        else
51        {
52          p->power_info.numerator=p->power_info.numerator+(SIZE_OF_RB*power)-
                p->power_info.totalP;
53          p->power_info.totalP=p->power_info.totalP+power-rb[idx];
54          moving_average= (p->power_info.numerator)/(p->power_info.noun);
55        }
56        rb[idx]=power;
57      break;
58
59      case EXPONENTIAL_MA:    //Exponential moving average
60        if(timeslice==1)
61          moving_average=power;
62        else
63          moving_average= (power*EMA_ALPHA)/1000+(((1000-EMA_ALPHA)*p->
                power_info.moving_average)/1000);
64      break;
65    }
66    if( timeslice<MAX_TIMESLICES)
67    {
68      p->power_info.butget_values[timeslice]=p->power_info.current_budget;
69      p->power_info.power_values[timeslice]=power;
70      p->power_info.moving_average_values[timeslice]=moving_average;
71    }
72  return moving_average;
73  }
```

Listing 5.7 shows the implementation of the different averaging algorithms. **Lines 10,11** detect violations. **Lines 13-20** are responsible for keeping the minimum and maximum power consumption up to date. **Line 25** calculates the storing index for the new power value. **Lines 29-40** represents the SMA algorithm. The WMA algorithm is shown from **line 42-57**. **Lines 59-64** give the implementation of the EMA algorithm. **Lines 66-71** are used to store additional power information of the consumed timeslice.

5.3.4 Selection of the Next Running Task

The last step of the implemented power management is the choice of the next running task. The main scheduling function "schedule()" is responsible for this choice. The scheduler works normally as long as the next chosen task does not exceed the given power budget. If a violation of the present power budget occurs, the scheduler tries to find another task which is able to fulfill the power constraints. If no task can be found, the scheduler waits until a higher present power budget is available. The following code lines describe the changes in the main scheduling function to achieve these goals.

Listing 5.8: Changes in the main scheduling function

```
1  /*
2   * schedule() is the main scheduler function.
3   */
4  asmlinkage void __sched schedule(void)
5  {
6  ...
7    idx = sched_find_first_bit(array->bitmap);
8    queue = array->queue + idx;
9    next = list_entry(queue->next, struct task_struct, run_list);
10
11   current_budget=budget[budget_cnt]*1000;
12   budget_cnt++;
13   if(budget_cnt==1000)
14     budget_cnt=0;
15
16   if(next->power_info.timeslice_cnt==0)
17   {
18     next->power_info.moving_average_values[0]= DEFAULT_AVERAGE;
19     next->power_info.moving_average=DEFAULT_AVERAGE;
20   }
21
22   while (next->power_info.moving_average>current_budget)
23   {
24     if(switch_cnt==2)
25     {
26       current_budget=budget[budget_cnt]*1000;
27       budget_cnt++;
28
29       performance++;
30       next->power_info.performance=performance;
31
32       if(budget_cnt==1000)
33         budget_cnt=0;
34       switch_cnt=0;
35     }
36     next->power_info.not_scheduled++;
37     dequeue_task(next, rq->active);
38     enqueue_task(next, rq->expired);
39
40     if (unlikely(!array->nr_active)) //change runqueues
41     {
42       rq->active = rq->expired;
43       rq->expired = array;
```

```
44          array = rq->active;
45          rq->expired_timestamp = 0;
46          rq->best_expired_prio = MAX_PRIO;
47          switch_cnt++;
48        }
49        idx = sched_find_first_bit(array->bitmap);
50        queue = array->queue + idx;
51        next = list_entry(queue->next, struct task_struct, run_list);
52      }
53      timeslice_start++;
54      next->power_info.current_budget=current_budget;
55      if(next->power_info.timeslice_cnt<MAX_TIMESLICES)
56      {
57        next->power_info.vdd[next->power_info.timeslice_cnt]=VDD;
58        next->power_info.f[next->power_info.timeslice_cnt]=F;
59        next->power_info.timestamp_start[next->power_info.timeslice_cnt]
60                                          = timeslice_start;
61      }
62  ...
63  }
```

Listing 5.8 shows the changes in the main scheduling function. **Lines 7-9** show the normal process to find the next running task. Then the present available power budget is calculated (**lines 11-14**). **Lines 16-20** set the default average value of the present power consumption. **Line 22** decides if the chosen task is allowed to run next. If the task is not allowed, then it will be dequeued from the active run queue and will be put to the back of the expired run queue (**lines 37,38**). Also the variable that shows power management activity will be updated (**line 36**). If the active run queue is empty after dequeuing the task, then the two run queues will be switched (**lines 40-48**). **Lines 49-51** shows the process to find the next suitable task. If no task in both run queues fulfills the power budget, the scheduler must wait until a higher power budget is available (**lines 24-35**). This case causes a performance loss (**lines 29,30**). If the next running task is found, the variables for the present power state of task and also for later evaluation will be set (**lines 53-61**).

5.3.5 Power Budget

As mentioned in the concept, the power budget in this work is modeled by an array. It includes 1000 entries, which have been chosen arbitrarily. The power budget array is situated in the "/includes/linux/budget.h" file. The following codes snippet is an extraction of this file (Listing 5.9).

Listing 5.9: Extract of the power budget array

```
1  //Array for the Power Budget
2  int budget[1000] = {
3                        159,
4                        125,
5                        145,
6                        169,
7                        146,
```

```
8                           151,
9                           132,
10                          143,
11                          144,
12                          ...
13                        };
```

5.3.6 Writing Power Information to the File System

For later evaluation purposes, the power information of a task is collected. The power information is stored in the power information structure. After a task has finished, its power information will be written to the file system of the OS with the "write_power_sum()" function.

Listing 5.10: Writing power information to the file system

```
1  static int write_power_sum(struct task_struct *p)
2  {
3    int i,lenght,cnt, offset;
4    char buf[200];
5    struct file* power_file;
6
7    sprintf(buf,"/home/nfs/power_info/%d",p->pid);
8    power_file=file_open(buf, O_RDWR | O_CREAT , 0);
9
10   sprintf(buf,"ProcessID: %d\nTimeslices: %d\nNot_Scheduled: %d\nViolations:
         %d\nMin. Power: %d\nMax. Power: %d Performance: %d\n\n\0",p->pid,p->
         power_info.timeslice_cnt,p->power_info.not_scheduled,p->power_info.
         violations,p->power_info.min_power,p->power_info.max_power,p->
         power_info.performance);
11
12   for(i=0;i<200;i++)
13   {
14     if(buf[i]=='\0')
15     {
16       lenght=i;
17       break;
18     }
19   }
20   p->power_info.write_offset=p->power_info.write_offset+file_write(
         power_file, p->power_info.write_offset,buf,lenght);
21
22   p->power_info.write_offset=p->power_info.write_offset+file_write(
         power_file,p->power_info.write_offset, "Budget - Power - Average - F - -
         VDD - Timestamp \n\n",52);
23   for(cnt=1;cnt<p->power_info.timeslice_cnt;cnt++)
24   {
25     if(cnt==(MAX_TIMESLICES-1))
26       break;
27
28     sprintf(buf,"%d | %d | %d | %d | %d | %d \n\0",
29     p->power_info.butget_values[cnt], p->power_info.power_values[cnt],
30     p->power_info.moving_average_values[cnt],p->power_info.f[cnt],p->
           power_info.vdd[cnt],
```

```
31      p->power_info.timestamp_start[cnt]);
32
33      for(i=0;i<200;i++)
34      {
35        if(buf[i]=='\0')
36        {
37          lenght=i;
38          break;
39        }
40      }
41      p->power_info.write_offset=p->power_info.write_offset+file_write(
            power_file, p->power_info.write_offset,buf,lenght);
42    }
43    file_close(power_file);
44  return 0;
45  }
```

Listing 5.10 shows the writing of power information to the file system. **Line 7,8** open a file with the name of the process ID of the task. First the characteristic power information (process ID, suspends, violations, minimum power, maximum power) is written to file (**lines 10-20**). **Lines 22-42** show the procedure of writing the exact power information (power budget, power consumption, average power consumption, frequency, supply voltage, starting time stamp) of each timeslice of a task. **Line 43** closes the file.

Listing 5.11: Call of the"write power" function

```
1  static inline void finish_task_switch(struct rq *rq, struct task_struct *
       prev)
2  {
3    ...
4    if (unlikely(prev_state == TASK_DEAD)) {
5      ...
6      write_power_sum(prev);
7      ...
8    }
9    ...
10 }
```

Listing 5.11 shows the call of the "write power" function. The power information will be written to the file system after the last task switch of a task (**line 6**).

Chapter 6

Evaluation and Results

This chapter illustrates the evaluation and the results of the implemented power management. First, the evaluation conditions are introduced. They include the evaluation measurement setup, a description of the evaluation task-set, the power profiling of the evaluation task-set and the variations of the averaging algorithms. Second, the measurement results are presented. The results are divided into: (i) the outcome of the OS-level power management and (ii) its impact on the LEON3 SoC's performance.

6.1 Evaluation

The evaluation of the implemented power management is important to gain results for the adapted system. The following terms are important for the evaluation:

- **Evaluation task-set**
  Is the set of the implemented tasks to test the target system.
- **Timeslice**
  The timeslice Ts_{ij} of a task T_j is a value which determines how long a task is allowed to run before interruption. A timeslice has been chosen to 200ms.
- **Idle time**
  If no task is allowed to run, the system stays in idle state for a period of time ($\rightarrow$ performance loss).
- **Task suspend**
  A suspend occurs if the present power budget of the next running task is lower than the average power consumption of the next running task ($P_B(Ts_{ij}) < P_{AVG}(T_j)$).
- **Violation**
  A violation occurs if the present power budget is lower than the power consumption of the present timeslice of a task ($P_B(Ts_{ij}) < P(Ts_j)$).

6.1.1 Evaluation Measurement Setup

Figure 6.1 shows the evaluation setup. The evaluation data will be generated locally on the target system by the evaluation task-set. The GRMON tool is used to transfer the

data from the target system to the host PC.

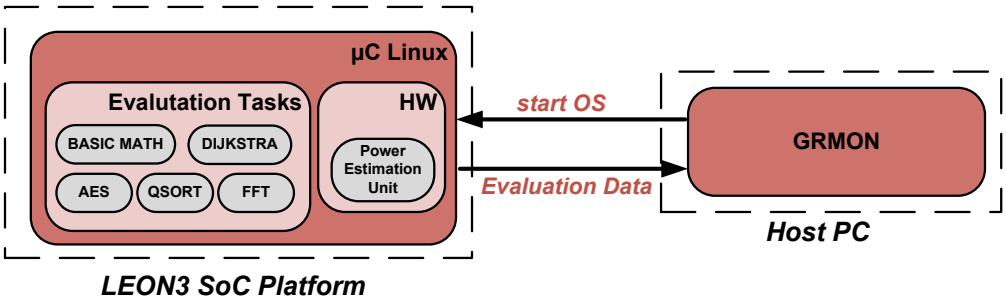

Figure 6.1: Evaluation setup

Evaluation Data
The following information is used for the evaluation:

- Number of timeslices consumed by the entire evaluation task-set
- Number of occurred power budget violations of the entire evaluation task-set
- Number of task suspends of the entire evaluation task-set
- Performance loss caused by the power budget violations of the entire evaluation task-set

Configuration of the Power Estimation Unit
The power estimation unit is configured as follows:

- Global enable of power estimation unit → On
- Averaging → On;
- Coarse-grained averaging → On;
- Averaging step range= 31250 (Equation (5.1))

For further information see Section 5.3.1.

6.1.2 Evaluation Task-Set

Five different tasks are running on the target system to evaluate the power management. These tasks will be started after the operating system is booted. The tasks represent typical applications for embedded systems. The following tasks compose the evaluation task-set:

- **AES (Advanced Encryption Standard)**
  The AES is a standard for a symmetric encryption system. It provides a block cypher with three key lengths: 128-bit, 192-bit, 256-bit. For the evaluation the AES with a key length of 128-bit is used.

- **BASIC MATH**
  This task includes some basic mathematical functions: solving of cubic equations, calculation of square roots, angle conversations.

- **QSORT**
  This task is an implementation of the "Quick Sort" sorting algorithm. The basis is the "Divide and Conquer" principle.

- **FFT (Fast Fourier Transformation)**
  The FFT is a basic algorithm in the area of signal processing . It is used to transform signals from the time domain to the frequency domain.

- **DIJKSTRA**
  DIJKSTRA is a network algorithm which is used to compute the shortest path between a start node and a target node. The basic is a graph with non-negative edge path costs.

6.1.3 Power Profiling of the Evaluation Task-Set

Figure 6.2 shows the power profiling of the implemented evaluation task-set. The power profiling is acquired by the power estimation unit and the average power consumption of the five evaluation tasks differ from 0.69 to 0.86. The BASIC MATH task and the FFT task are the tasks with the highest average power consumption. They have an average of approximately 0.85. The AES task has the lowest average power consumption of approximately 0.7. The DIJKSTRA task has the third highest average power consumption and the QSORT task the fourth highest. These two tasks show a significant increase in their power consumption after a certain period of time. This step is the result of the initial phase of these two tasks.

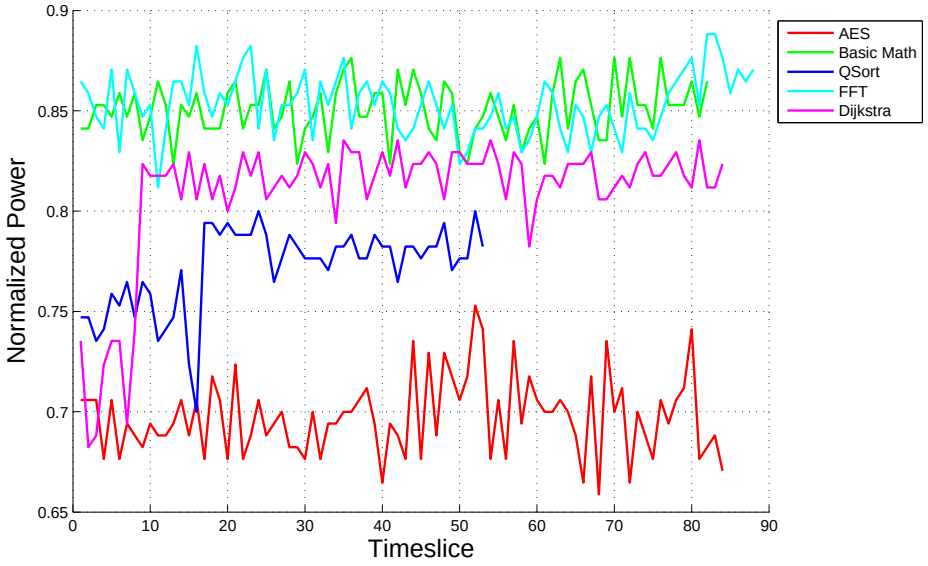

Figure 6.2: Power profiles of the evaluation task-set

6.1.4 Variations of the Averaging Algorithms

As mentioned in Chapter 4, three different moving average filter algorithms are implemented to support the chosen power management. These averaging algorithms are used to provide an adaptable power management. For the evaluation, the power management is tested with different parameterization of the averaging algorithms. Additionally, two further cases are tested. First, power management is disabled. This case is used as a reference. Second, only the power consumption of the last consumed timeslice of a task is used for the scheduling decision (LastPowerVal). Table 6.1 shows the used algorithms for the evaluation of the power management.

Algorithm	**Variation**	**Descriptions**
Without Power Management	-	Reference implementation - Power management is disabled
Simple Moving Average	Buffer = 5	SMA Algorithm - The last 5 power values are used for the average calculation
	Buffer = 20	SMA Algorithm - The last 20 power values are used for the average calculation
	Buffer = 50	SMA Algorithm - The last 50 power values are used for the average calculation
Weighted Moving Average	Buffer = 5	WMA Algorithm - The last 5 power values are used for the average calculation
	Buffer = 20	WMA Algorithm - The last 20 power values are used for the average calculation
	Buffer = 50	WMA Algorithm - The last 50 power values are used for the average calculation
Exponential Moving Average	Alpha = 0.9	EMA Algorithm - Present power consumption is weighted with 90%, last average power consumption with 10%
	Alpha = 0.5	EMA Algorithm - Present power consumption is weighted with 50%, last average power consumption with 50%
	Alpha = 0.1	EMA Algorithm - Present power consumption is weighted with 10%, last average power consumption with 90%
Last Power Val	-	Only the power consumption of the last consumed timeslice of task is used for the scheduling decision

Table 6.1: Variations of the evaluation algorithms

6.2 Results

The results can be divided in two sections:

- **Outcome of the OS-level power management**
  Shows the changes in the scheduling order and the changed fragmentation of the computation time. This outcome of the implemented OS-level power management is mainly caused by the present power budget.
- **Impact on the LEON3 SoC's performance**
  Gives information of the performance changes of the LEON3 SoC, caused by the implemented power management.

6.2.1 Outcome of the OS-Level Power Management

These results give information about the scheduling order and the fragmentation of the computation time of the executed evaluation task-set. The timeslices, the task suspends and the violations will be shown. To illustrate the results, the cases when power management is disabled (Without-PM) and the best averaging algorithm (EMA-Alpha09) will be compared. The results of the other averaging algorithms are shown in Appendix A.

Without Power Management

Figure 6.3 shows the scheduling order when power management is disabled. The evaluation task-set will be scheduled in a normal round robin order. The snapshot of the scheduling order also shows the occurred violations. The pie chart gives information about the computation time. Because of the normal round robin scheduling order, every task gets the same amount of computation time.

Figure 6.4 shows the detailed results of the evaluation task-set when power management is disabled. The timeslices-subplot shows the consumed timeslices to finish to execution of the evaluation task-set. The number of consumed timeslices differ from 53 to 88. The suspends-subplot shows that no suspend occurs when power management is disabled. This is clear because the power management is responsible for suspends. The violations-subplot shows that the number of violations differ from 1 to 48. A task with a low power consumption (AES) causes fewer violations than tasks with a high power consumption (BASIC MATH, FFT).

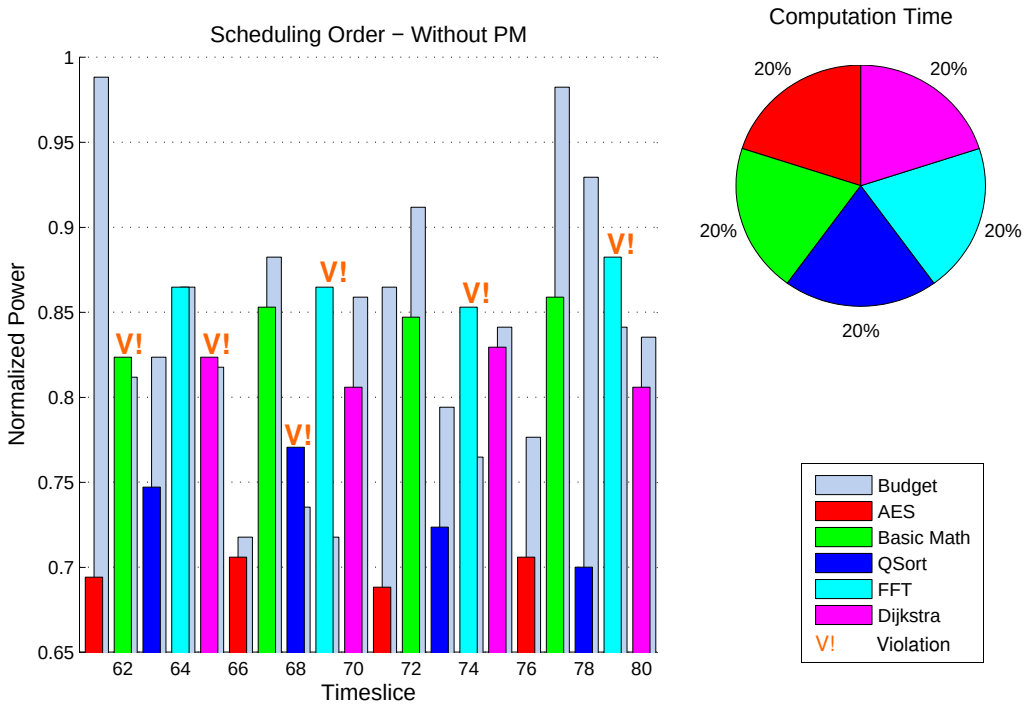

Figure 6.3: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: Without-PM

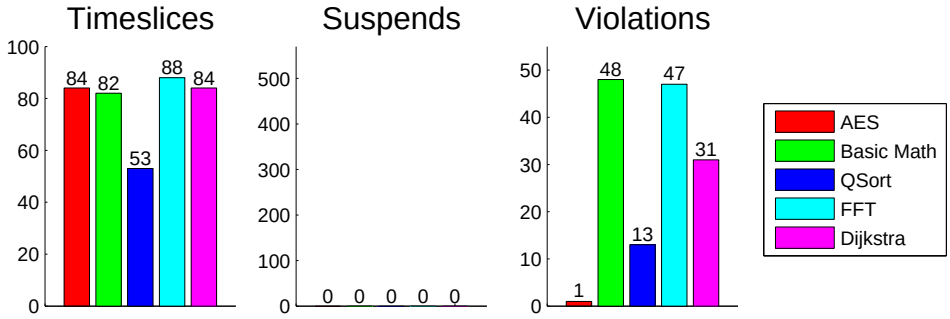

Figure 6.4: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: Without-PM

Exponential Moving Average - Alpha=0.9

Figure 6.5 illustrates the scheduling order when the EMA-Alpha09 algorithm is used to support the power management. The snapshot of the scheduling order also shows the power consumption of the consumed timeslices and the corresponding power budget. The order differs strongly from a normal round robin scheduling, because the power management suspends tasks, which would cause violations. The fragmentation of computation time is shown in the pie chart. Tasks with low power consumption (AES) will be scheduled more often than tasks with a high power consumption (BASIC MATH, FFT).

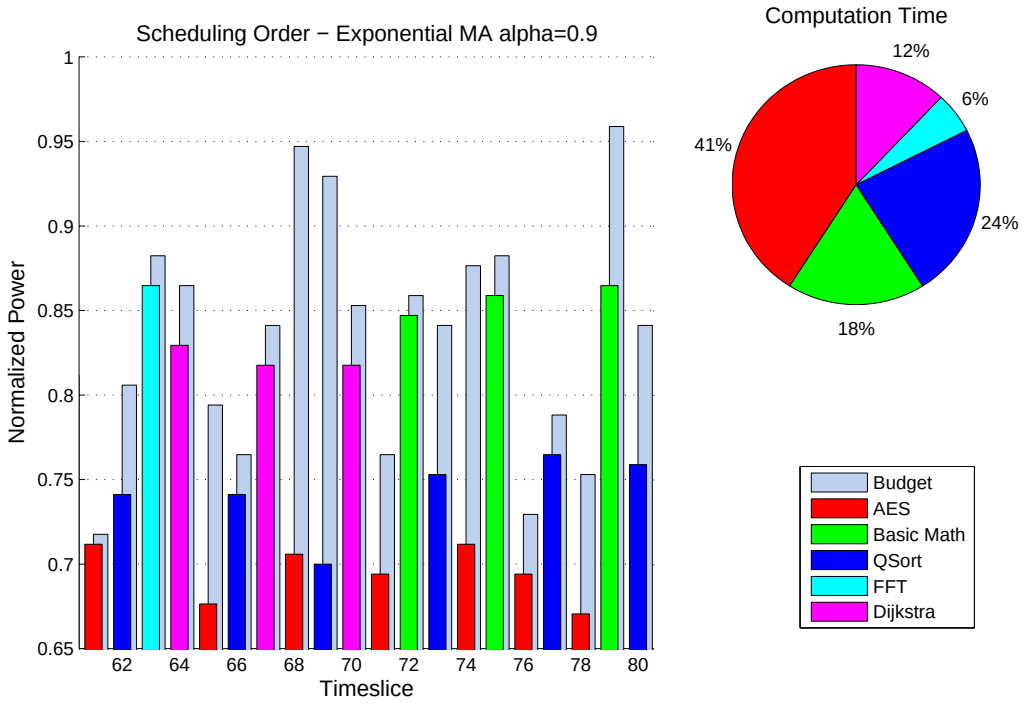

Figure 6.5: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: EMA-Alpha09

Figure 6.6 shows the detailed results when the EMA-Alpha09 algorithm is used to support the power management. The number of timeslices are nearly similar to the consumed timeslices in the case without power management. The number of suspends differ from 34 to 493. This number is highly dependent on the average power consumption of the corresponding task. Suspends occur if the present power budget for the next running task is lower than its average power consumption ($P_B(Ts_{ij}) < P_{AVG}(T_j)$). A task with a low power consumption (AES) causes significant fewer suspends than a task with a high power consumption (FFT). The suspends of tasks with a high power consumption amounts to a multiple of their consumed timeslices because the execution of the tasks with a low power consumption is finished first and only tasks with a high power consumption are waiting for their execution. Then the present power budget is often to low and the tasks will be suspended again and again. The violations-subplot shows the improvement of the implemented power management in contrast to the case when power management is disabled. Only 0 to 7 violations occur during the execution of the evaluation task-set. Again tasks with a high power consumption perform worse.

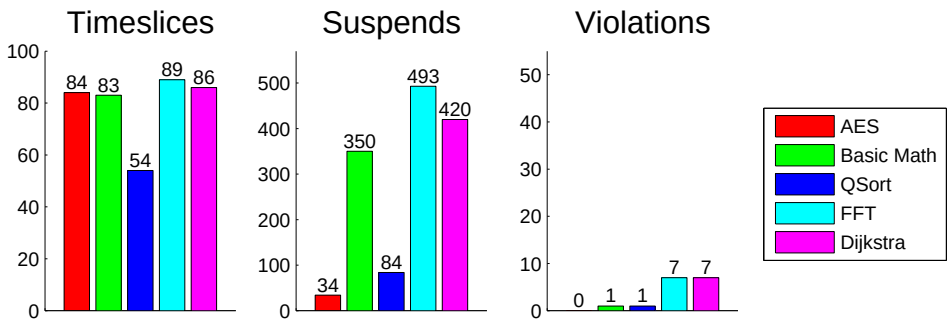

Figure 6.6: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: EMA-Alpha09

6.2.2 Impact on the LEON3 SoC's Performance

The performance of the LEON3 SoC is important for the evaluation of the power management, because with these results it is possible to show the potential of the implemented power management. Also the disadvantages of the implemented power management are shown. The important key figures are the number of caused violations and the performance loss.

Consumed Timeslices

The number of consumed timeslices does not depend on the different averaging algorithms. To compute the evaluation task-set, a mean of 393 timeslices were consumed. Small differences in the number of consumed timeslices are common and caused by the operating system.

Number of Task Suspends

The number of task suspends gives information about the scheduler activity. A suspend occurs if the present power budget for the next running task is lower than its average power consumption ($P_B(Ts_{ij}) < P_{AVG}(T_j)$). Figure 6.7 shows the number of task suspends which occur while the evaluation task-set is computed. The number of suspends depends on the used averaging algorithm and has a variation of 8%. If power management is disabled, no task suspend occurs. The number of occured suspends is a multiple of the cosumed timeslices. This is caused by the tasks with a high power consumption. For more details to task suspends see 6.2.1.

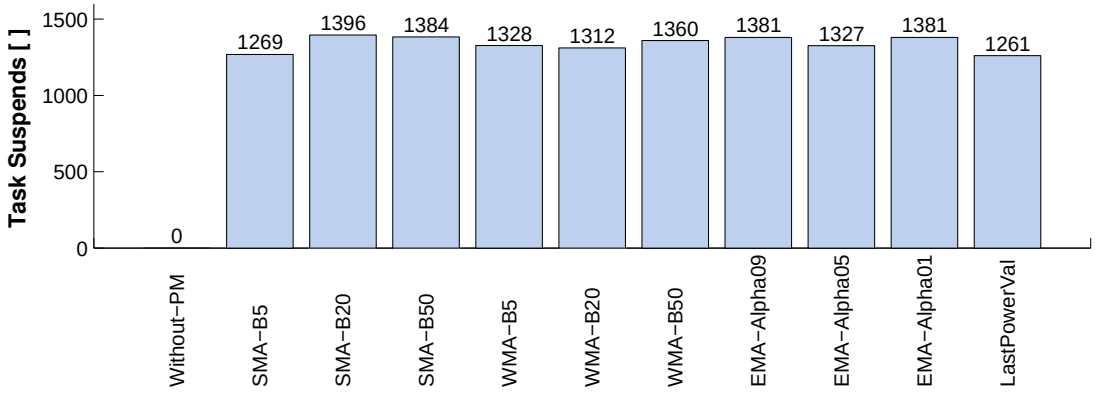

Figure 6.7: Number of task suspends of the evaluation task-set dependent on the different averaging algorithms

Percentage of Violations

The percentage of violations is the most important key figure. It gives information about the occurred violations of the given power budget. A violation occurs if the present power budget is lower than the power consumption of the present timeslice of a task ($P_B(Ts_{ij}) < P(Ts_j)$). Figure 6.8 shows the occurred violations of the evaluation task-set dependent on the different averaging algorithms. When power management is disabled, approximately 35% of the scheduled timeslices cause a violation. The improvement with the use of the power management is significant, the percantage of violations decreases to less than 10%. The best averaging algorithm was the EMA with α=0.9. Power management which is supported with this algorithm causes only 4% violations and reaches an improvement of 8x.

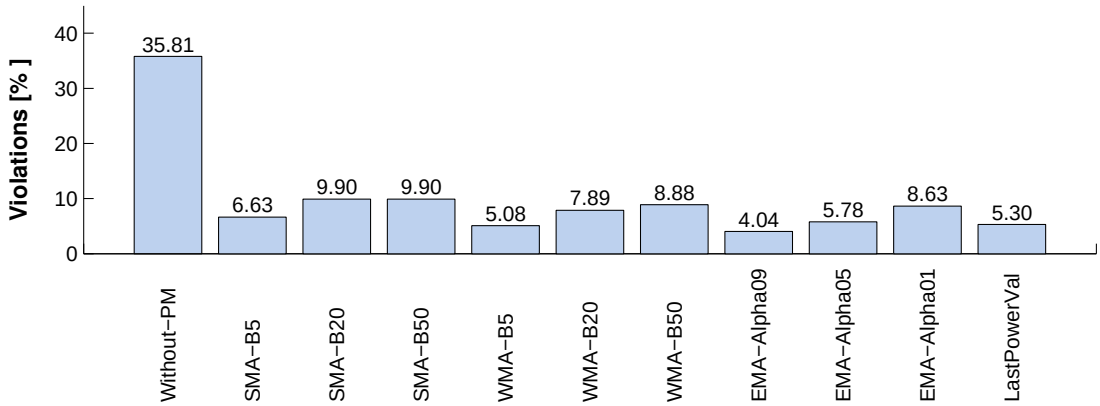

Figure 6.8: Percentage of occurred violations of the of the evaluation task-set dependent on the different averaging algorithms

Performance Loss

Figures 6.9 and 6.10 show the performance loss of the implemented power management. Performance loss must be accepted if no task in the run queues is below the present power budget. Then the system stays in idle state until a higher power budget is available. Figure 6.9 shows the performance loss with an idle time of 200ms. Figure 6.9 illustrates a performance loss of approximately 30% when power management is used. If power management is disabled, the performance is maximum (100%). An improvment of the performance can be reached through a reduction of the idle time. If the duration of one idle timeslice is reduced to 100ms, the performance loss is 15-20% (Figure 6.10). This reduction is only possible if the given power budget also changes its value every 100ms.

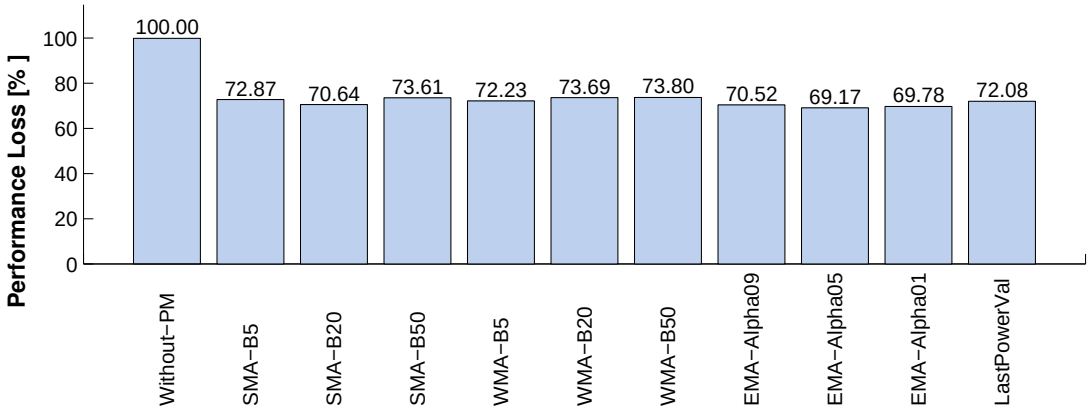

Figure 6.9: Performance loss - Idle time = 200ms

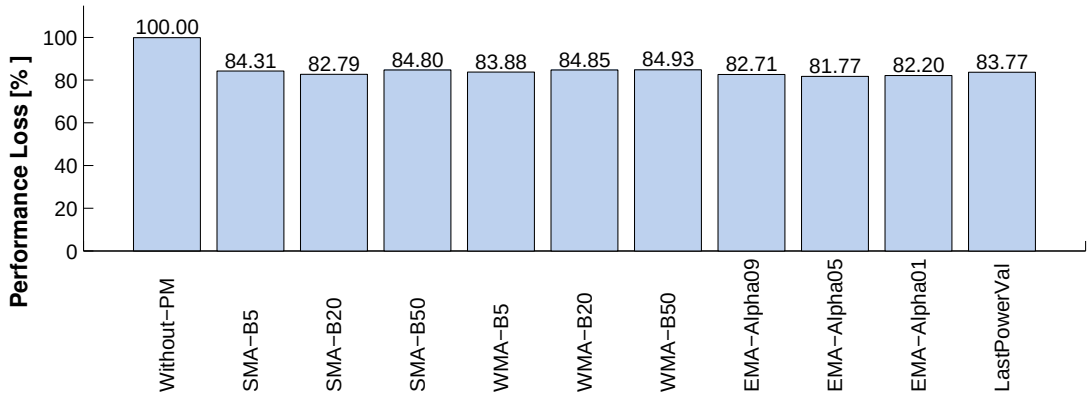

Figure 6.10: Performance loss - Idle time = 100ms

The performance results show that the implemented power management is a promising alternative for systems with limited power. With the use of the power management, a reduction of the occurred violation by 8x is possible and the performance loss is only approximately 30%.

Chapter 7

Conclusion and Future Work

To fulfill today's power requirements for designing embedded systems, power management is essential [She08]. Power management is used to deal with two existing issues: (i) limited energy with the goal to minimize the energy/power consumption of the system and (ii) limited power with the main goal to ensure a stable operation of the system.

7.1 Conclusion

This thesis focuses on the design and implementation of an OS-level power management which is able to handle the problem of limited power. Basis for the implementation of the power management is a Linux operating system. To realize the power management, the present power consumption of a LEON3 SoC is used. This present power consumption is provided by an additional power estimation unit. Task rescheduling as a power management technique is used to minimize the violations of a given power budget. The implemented power management is set up in four major steps: (i) acquisition of the present power information for an executed task by the help of the power estimation unit, (ii) storing of the power information, (iii) calculation of the present average power consumption of the executed tasks and (iv) selection of the next running task under consideration of the given power budget. Adaptations of the scheduler function of the Linux operating system are realized to implement the chosen power management. Different algorithms for the calculation of the average power consumption of the tasks have been implemented to provide variations of the power management. A set of tasks was used for the evaluation of the power management. The tasks are typical applications (cryptography, mathematics, network, signal processing) for embedded systems.

A significant improvement of the occurred violations of the given power budget could be gained by the use of the implemented power management. Without power management approximately 35% of the scheduled timeslices cause a violation for a predetermined power budget. If the implemented power management is used, the number of occurred violations can be minimized to only 4% and results in an improvement of approximately 8x. The performance loss is kept within reasonable limits and amounts to approximately 30%. To sum up it can be said that the designed and implemented power management is an adequate approach for systems with limited power.

7.2 Future Work

The implemented power management shows potential for systems with limited power. Despite the shown potential, some open issues should be explored and solved. It has to be kept in mind that the current edition of the implemented power management was tested on a real hardware system, but the given power budget has been chosen arbitrarily. It would be also possible to gain some improvements if the implemented power management worked in combination with a hardware power management mechanism. Furthermore, additional metrics for the scheduling decision could be explored.

7.2.1 Exploration of the Implemented Power Management with a Real Power Budget

The implemented power management was tested with an arbitrarily chosen power budget. For the operation in real systems it is necessary to test and evaluate the power management with a real power budget. A solar panel would be a common choice for a power supply source to test the power management under real conditions. The results should be satisfactory, because the averaging algorithms of the power management match well with the quadratic trend of the solar activity (Figure 2.2). Also other power supply sources (kinetic energy, radio frequency radiation) should be tested.

7.2.2 Hardware Power Management Mechanism

An additional use of a hardware power management mechanism would be promising for the implemented OS-level power management. The hardware power management mechanism is able to reduce weaknesses of the implemented OS-level power management. Therefore, the use of hardware based dynamic voltage and frequency scaling (DVFS) is recommended to gain improvements.

Weaknesses of the Implemented OS-Level Power Management

- Granularity - The operating system is not able to handle power peaks, which occur during a timeslice.
- Highest priority task - A task with a very high power consumption could starve.
- Power consumption of the LEON3 SoC - The minimum power consumption of the LEON3 SoC is fixed and cannot be lowered.

DVFS for Dealing with Power Peaks During a Timeslice

It is necessary to deal with power peaks to guarantee a stable operation of the target system. These power peaks can occur during a timeslice and the operating system is not able to handle it. Hardware based DVFS can reduce the violations of the power budget, because it is fast enough to avoid the power peaks.

DVFS for Highest Priority Tasks

Hardware based DVFS can also be used to guarantee execution time for highest priority tasks. DVFS can lower the power consumption of the whole system and highest priority tasks with a high power consumption are able to run without violations of the given power budget.

DVFS to Minimize Idle Time

If no task is able to fulfill the present power budget, the system runs in idle mode. Instead of idle time, DVFS can lower the power consumption of the whole system and tasks are able to run within the given power constraints.

7.2.3 Introduction of Additional Metrics for the Scheduling Decision

The scheduling decision is based on the average power consumption of a task and the present algorithms for the calculation of the power consumption of a task rely on averaging metrics. Dependent on the power consumption trend of a task, other metrics could achieve better results. The existing averaging algorithms could be modified with a threshold or a variance.

Appendix A

Detailed Results

The following sections show detailed results of the different averaging algorithms. Each section consists of two figures. The first figure shows the scheduling order. When power management is used, the scheduling order differs strongly from a normal round robin scheduling. The changes are caused by the implemented power management, based on task suspends.
The second figure shows the results of the evaluation task-set in terms of number of consumed timeslices, number of task suspends and number of violations. Each evaluation task is examined separately.

A.1 Simple Moving Average - Bufer=5

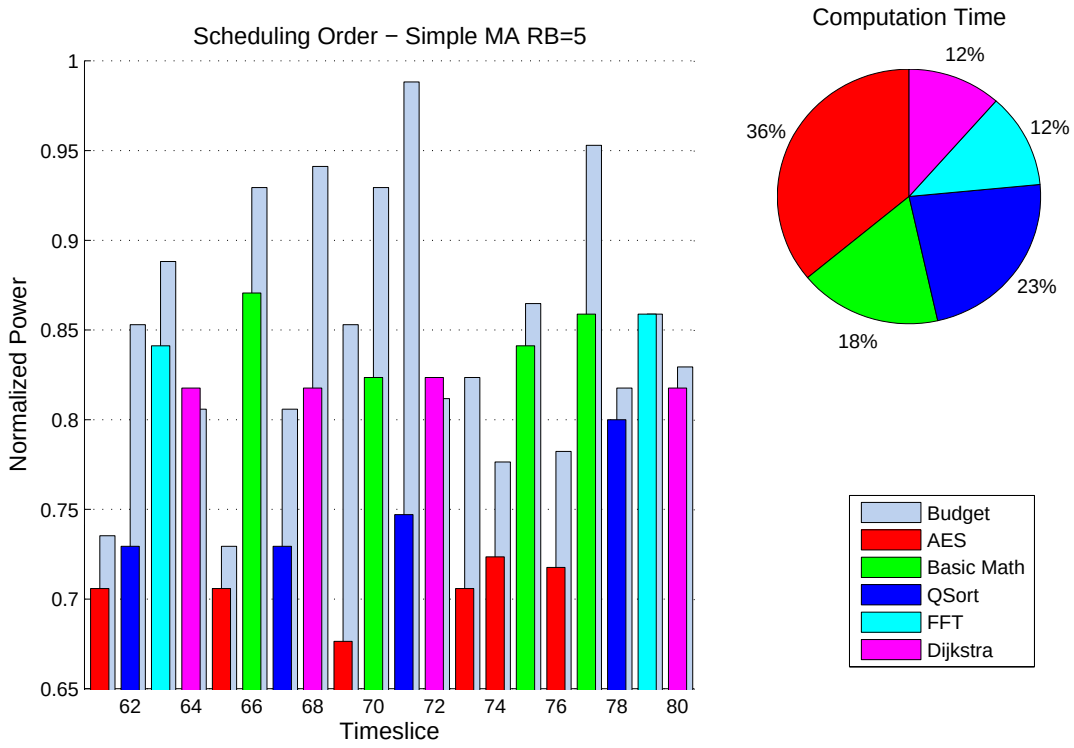

Figure A.1: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: SMA-B5

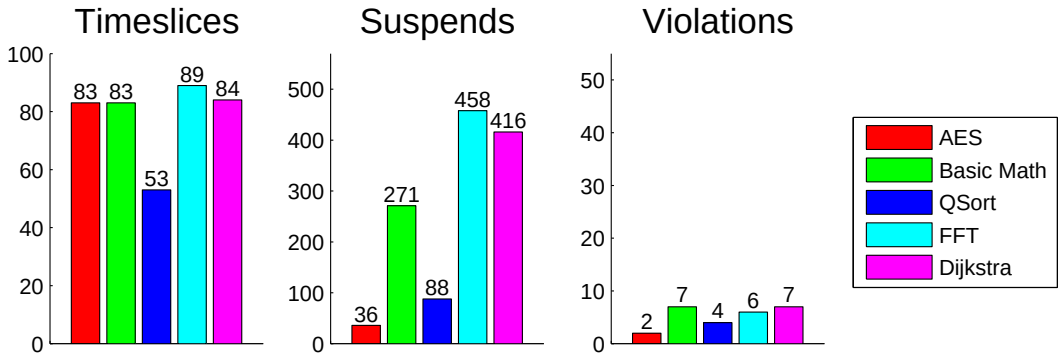

Figure A.2: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: SMA-B5

A.2 Simple Moving Average - Bufer=20

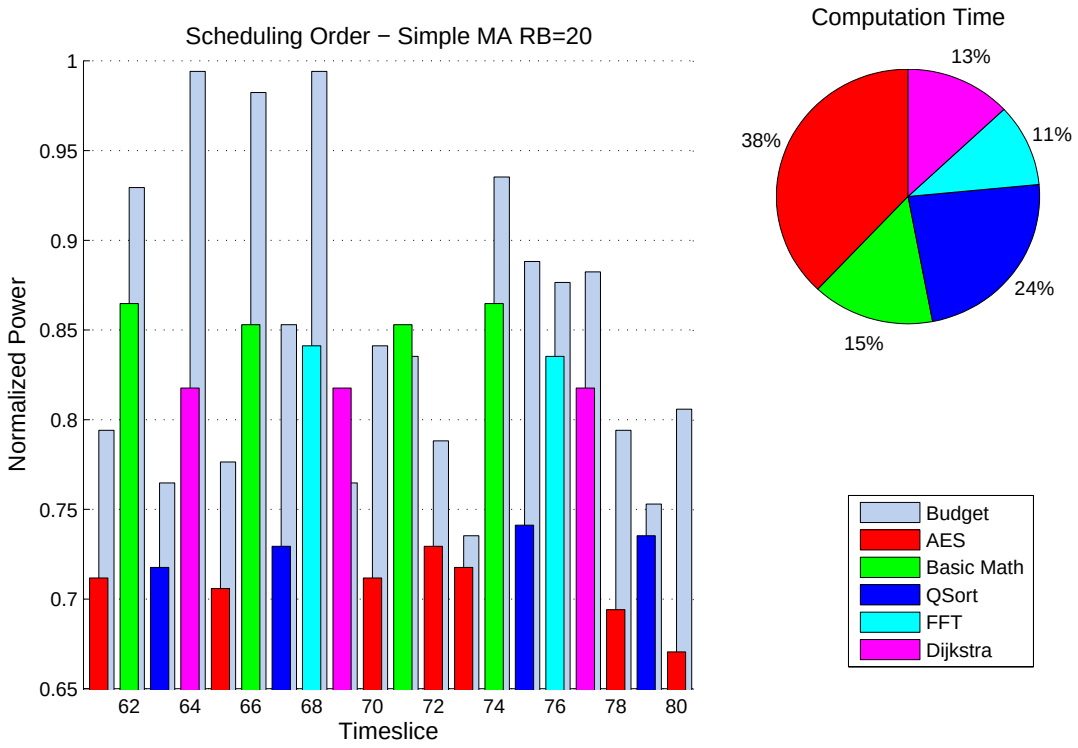

Figure A.3: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: SMA-B20

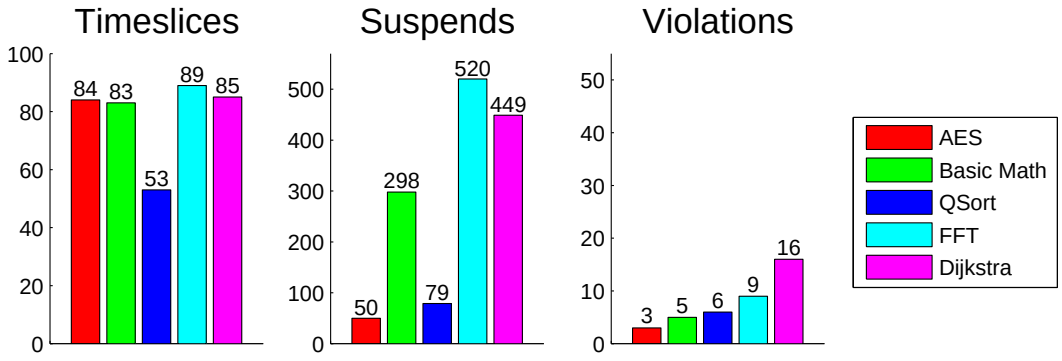

Figure A.4: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: SMA-B20

A.3 Simple Moving Average - Bufer=50

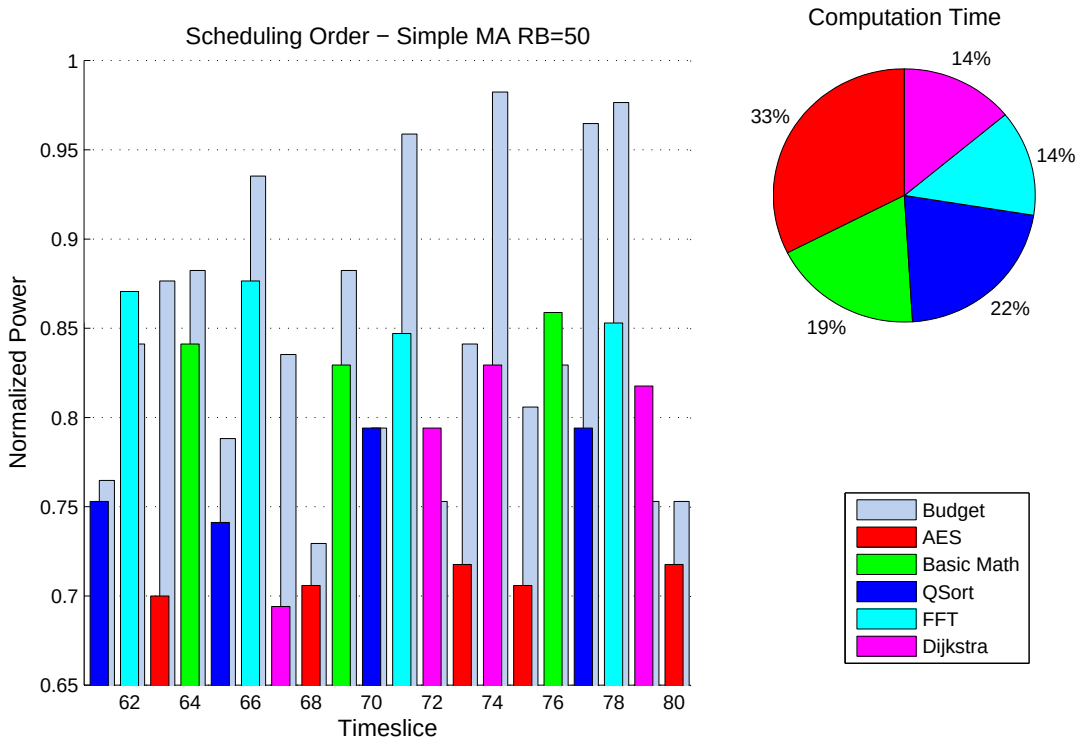

Figure A.5: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: SMA-B50

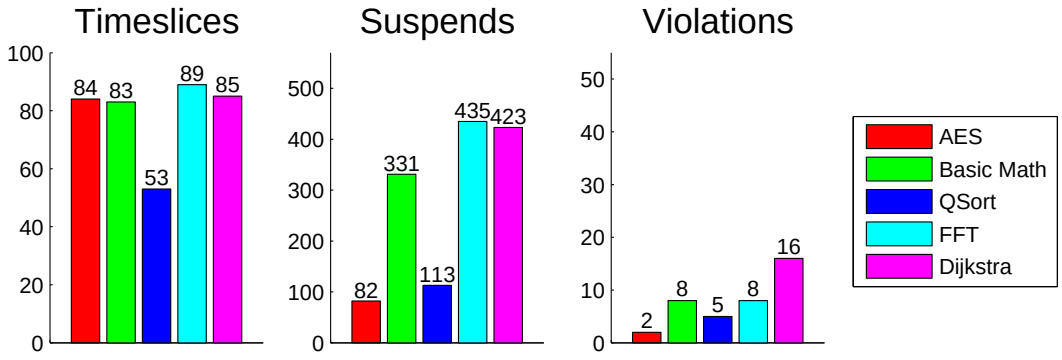

Figure A.6: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: SMA-B50

A.4 Weighted Moving Average - Bufer=5

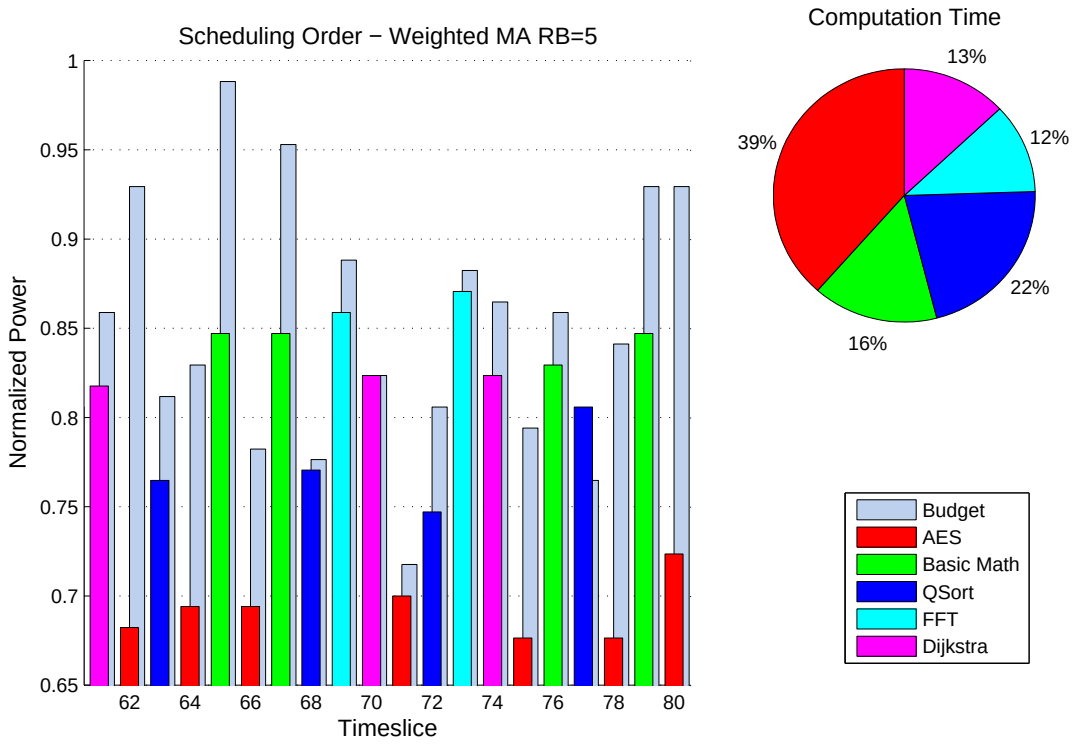

Figure A.7: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: WMA-B5

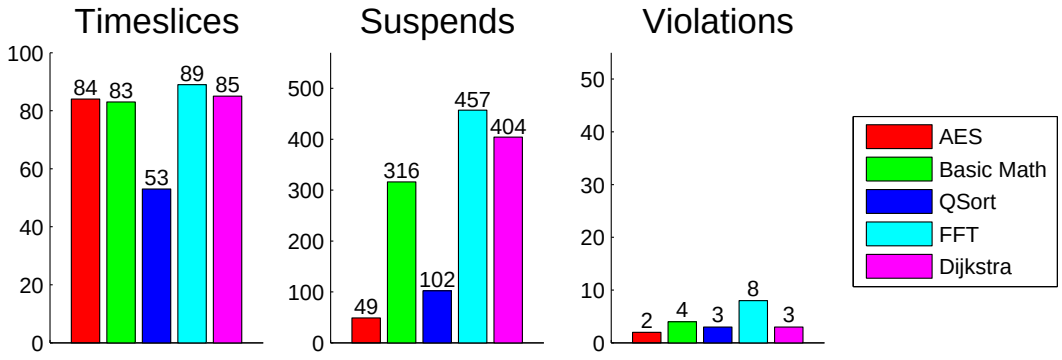

Figure A.8: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: WMA-B5

A.5 Weighted Moving Average - Bufer=20

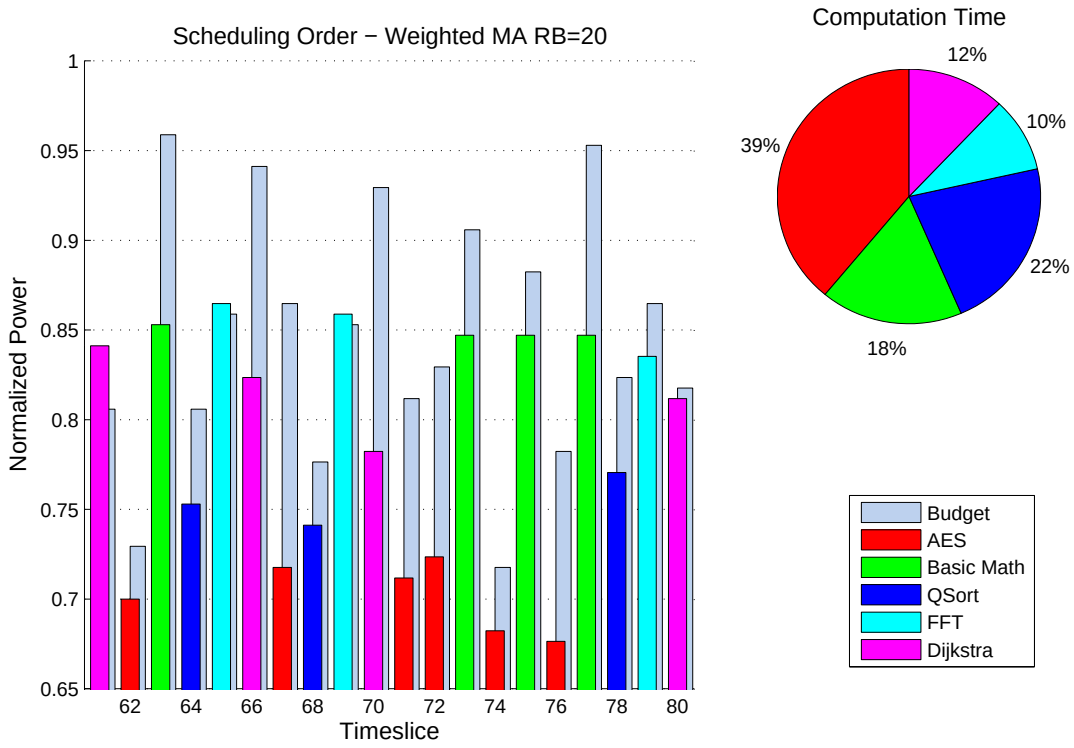

Figure A.9: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: WMA-B20

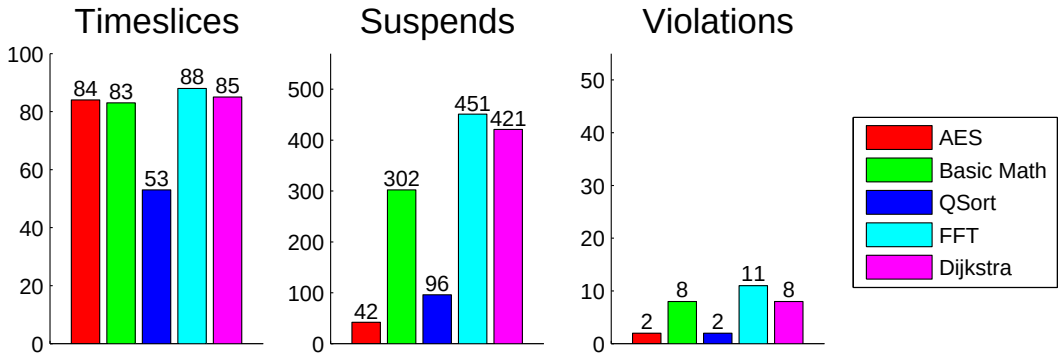

Figure A.10: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: WMA-B20

A.6 Weighted Moving Average - Bufer=50

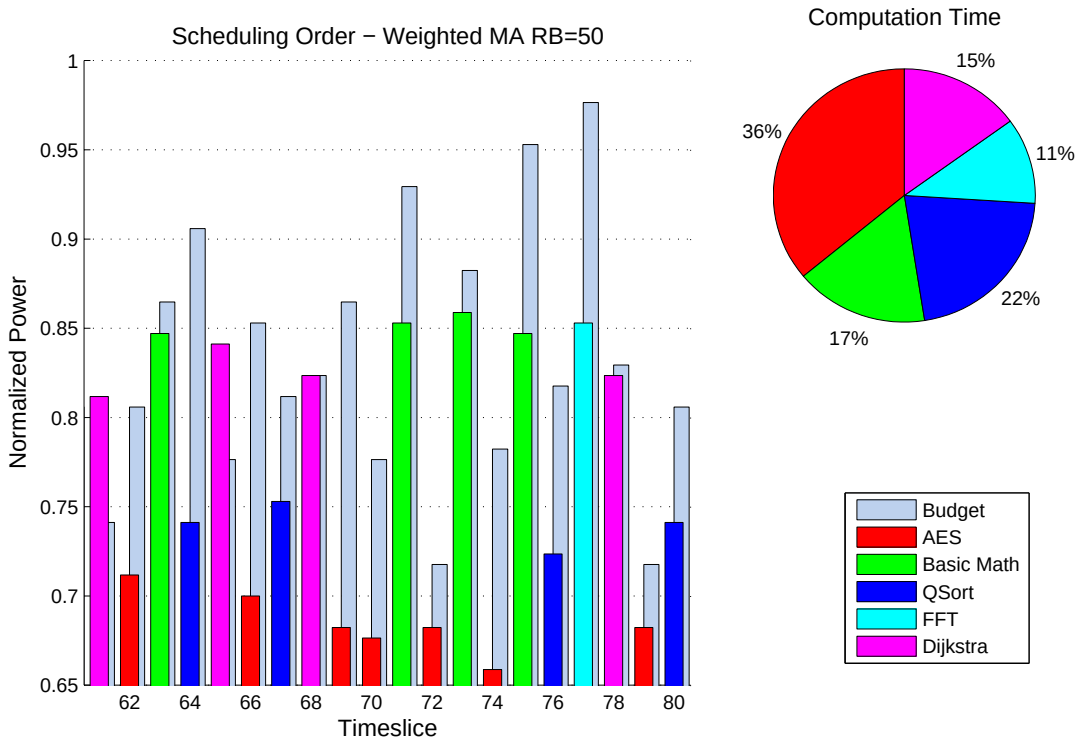

Figure A.11: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: WMA-B50

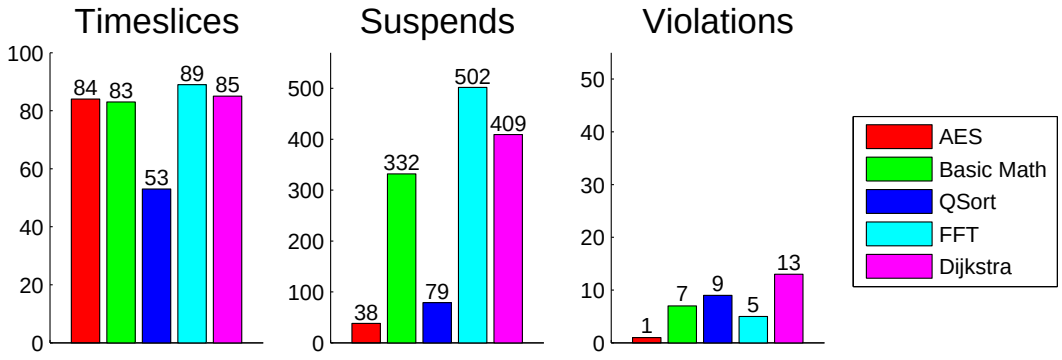

Figure A.12: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: WMA-B50

A.7 Exponential Moving Average - Alpha=0.5

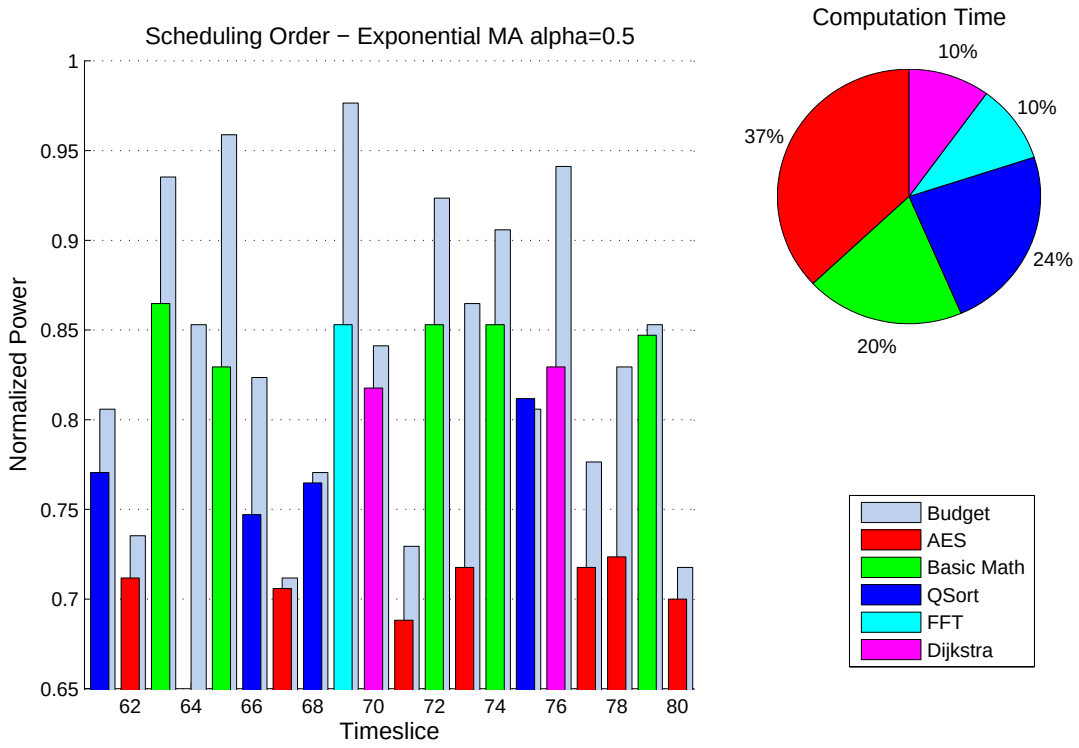

Figure A.13: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: EMA-Alpha05

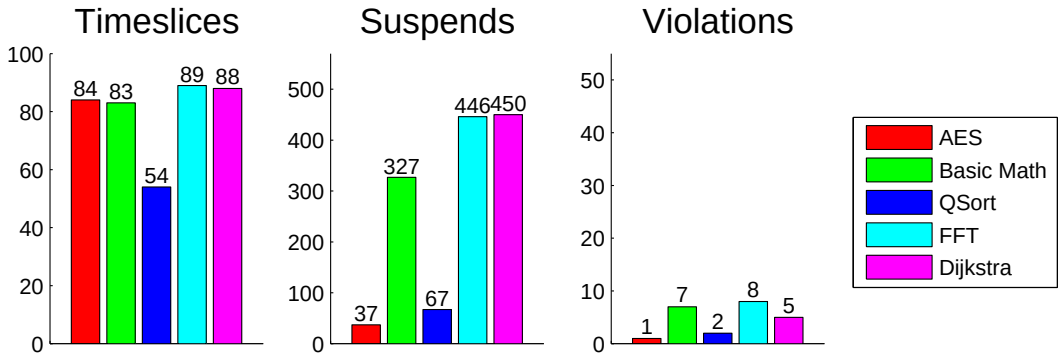

Figure A.14: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: EMA-Alpha05

A.8 Exponential Moving Average - Alpha=0.1

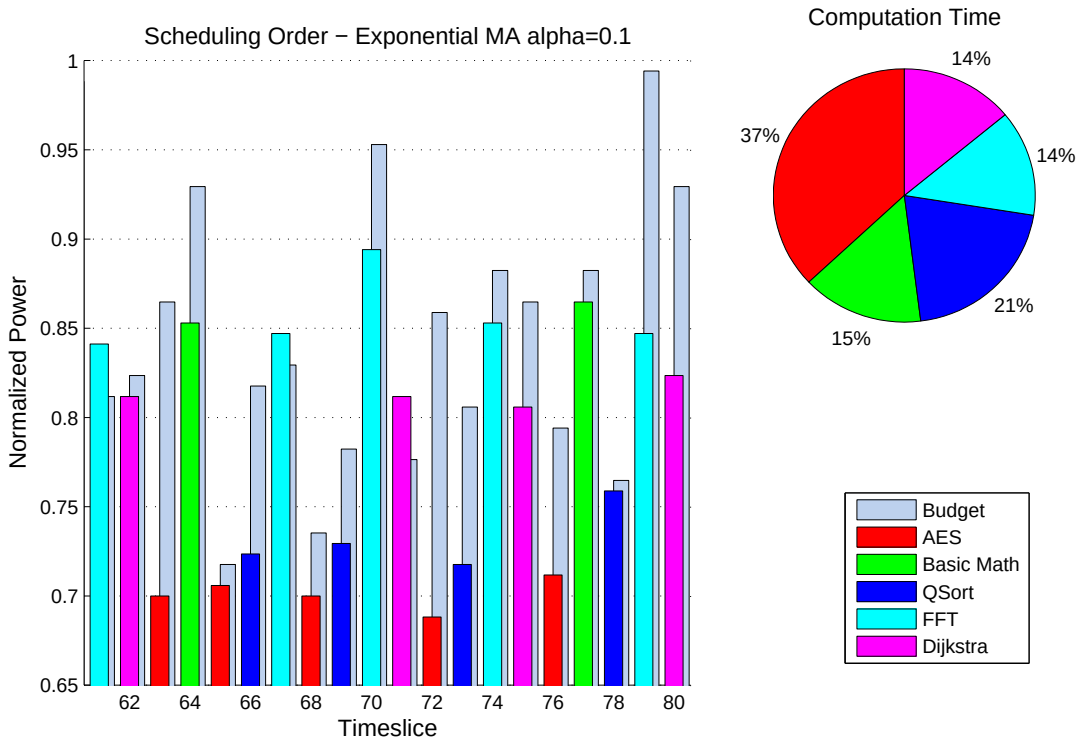

Figure A.15: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: EMA-Alpha01

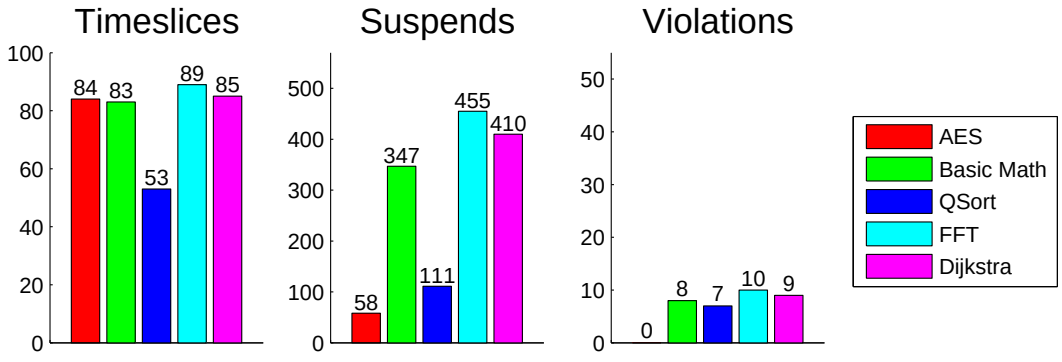

Figure A.16: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: EMA-Alpha01

A.9 Last Power Value

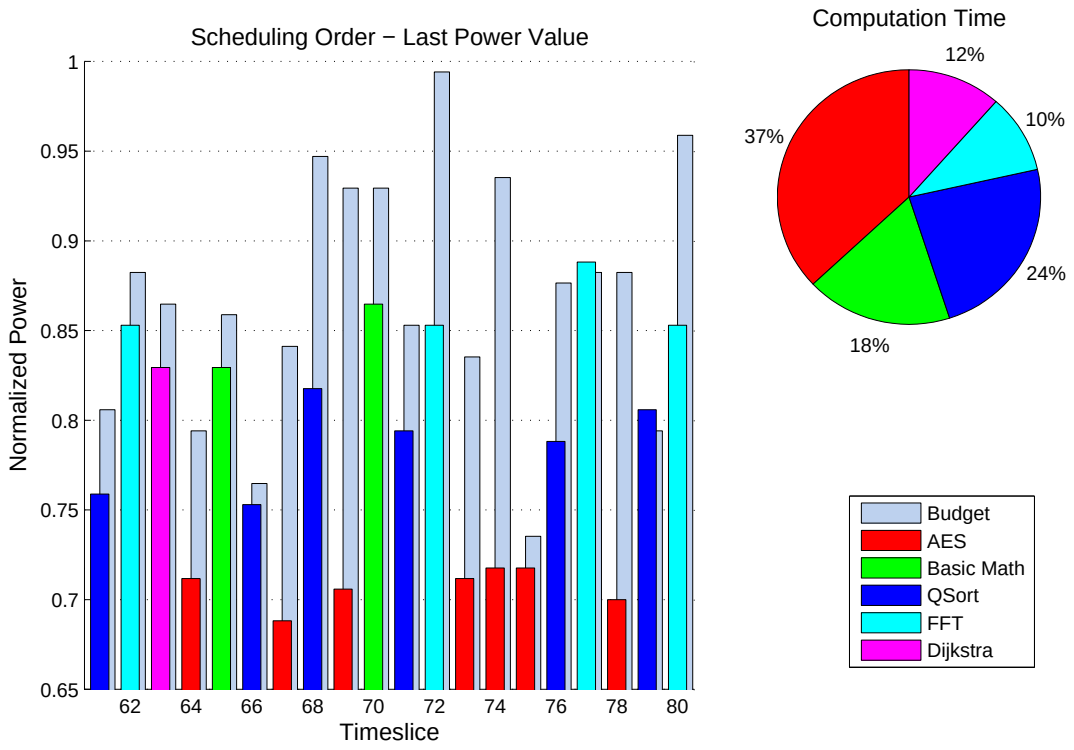

Figure A.17: Representative extract of the scheduling order and the computation time fragmentation. Averaging algorithm: LastPowerVal

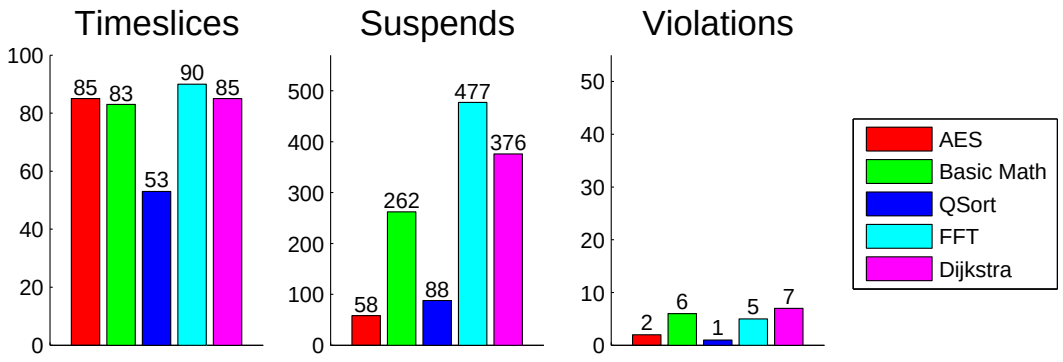

Figure A.18: Number of timeslices, suspends and violations of the evaluation task-set. Averaging algorithm: LastPowerVal

List of Abbreviations

A/D	Analog to Digital
ACPI	Advanced Configuration and Power Interface
APM	Advanced Power Management
ASIC	Application Specific Integrated Circuit
AMP	Asymmetric Multiprocessing
CPU	Central Processing Unit
D/A	Digital to Analog
DPM	Dynamic Power Management
DPTC	Dynamic Process and Temperature Compensation
DVS	Dynamic Voltage Scaling
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
EMA	Exponential Moving Average
FCFS	First Come First Served
FIFO	First In First Out
FPGA	Field Programmable Gate Array
I/O	Input/Output
MIPS	Million Instructions Per Second
OS	Operating System
PC	Personal Computer
PEU	Power Estimation Unit
PMU	Power Management Unit
RF	Radio Frequency
RM	Rate Monotonic
RTOS	Real-Time Operating System
R&D	Research and Development
SMA	Simple Moving Average
SJF	Shortest Job First
SMP	Synchronous Multiprocessing
SoC	System on Chip
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WCET	Worst Case Execution Time
WMA	Weighted Moving Average

Bibliography

[Aas05] Josh Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. *Silicon Graphics, Inc.*, 2005.

[Abb06] Doug Abbott. *Linux for Embedded and Real-Time Applications*. Elsevier Inc., 2006.

[ACP09] Advanced Configuration and Power Interface Specification, June 2009.

[AER05] AEROFLEX Gaisler. *LEON3 - Multiprocessing CPU Core*, 2005.

[AER10] AEROFLEX Gaisler. *http://www.gaisler.com*, 2010.

[BCMS01] L. Benini, G. Castelli, A. Macii, and R. Scarsi. Battery-Driven Dynamic Power Management. *Design & Test of Computers, IEEE*, 18 Issue: 2:53 – 60, 2001.

[BGS+10] C. Bachmann, A. Genser, C. Steger, R. Weiss, and J. Haid. Automated Power Characterization for Run-Time Power Emulation of SoC Designs. 2010.

[BV08] Udo Bankhofer and Jürgen Vogel. *Datenanalyse und Statistik*. Gabler, 2008.

[Cam09] K.W. Cameron. The Road to Greener IT Pastures. *Computer*, 42:87–89, 2009.

[CSB92] A. Chandrakasan, S. Sheng, and R. W. Brodersen. Lowp-Power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27:473–484, 1992.

[GACB+04] Bita Gorji-Ara, Pai Chou, Nader Bagherzadeh, Mehrdad Reshadi, and David Jensen. Fast and Efficient Voltage Scheduling by Evolutionary Slack Distribution. *Asia-South Pacific Design Automation Conference*, 2004.

[GBH+09] Andreas Genser, Christian Bachmann, Josef Haid, Christian Steger, and Reinhold Weiss. An Emulation-Based Real-Time Power Profiling Unit for Embedded Software. *Systems, Architecture, Modeling and Simulation*, pages 67 – 73, 2009.

[GCO06] Yan Gu, Samarjit Chakraborty, and Wei Tsang Ooi. Games are up for DVFS. *ACM IEEE Design Automation Conference*, Session 35:598–603, 2006.

[Hea03] Steve Heath. *Embedded Systems Design*. Newnes, 2003.

[HK03] Chung-Hsing Hsu and Ulrich Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. *Conference on Programming Language Design and Implementation*, pages 38–48, 2003.

[KHZS07] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. Power Management in Energy Harvesting Sensor Networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 6, 2007.

[Kim06] Taewhan Kim. Application-Driven Low-Power Techniques Using Dynamic Voltage Scaling. In *Embedded and Real-Time Computing Systems and Applications*, 12th, pages 199–206, 2006.

[Kit09] Troy Kitch. Green Up: Strategies for Dynamic Power Management using embedded Linux. *Embedded Computing Design*, 2009.

[LDM01] Yung-Hsiang Lu and G. De Micheli. Comparing System-Level Power Management Policies. *Design & Test of Computers, IEEE*, 18 , Issue:2:10 – 19, 2001.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *ACM*, 20:46 – 61, 1973.

[LLTW06] Huaxiang Lu, Yan Lu, Zhifang Tang, and Shoujue Wang;. Soc Dynamic Power Management Using Artificial Neural Network. *Intelligent Systems Design and Applications*, pages 133 – 137, 2006.

[LNS+07] Jeabin Lee, Byeong-Gyu Nam, Seong-Jun Song, Namjun Cho, and Hoi-Jun Yoo. A Power Management Unit with Continuous Co-Locking of Clock Frequency and Supply Voltage for Dynamic Voltage and Frequency Scaling. *Circuits and Systems*, pages 2112 – 2115, 2007.

[MM05] Loreto Mateu and Francesc Moll. Review of Energy Harvesting Techniques and Applications for Microelectronics. *The International Society for Optical Engineering*, 5837:359–373, 2005.

[Mur08] S. Murugesan. Harnessing Green IT: Principles and Practices. *IT Professional*, 10:24–33, 2008.

[OKKK08] Seungyong Oh, Jungsoo Kim, Seonpil Kim, and Chong-Min Kyung. Task Partitioning Algorithm for Intra-Task Dynamic Voltage Scaling. *Circuits and Systems*, pages 1228 – 1231, 2008.

[Ols08] Stephen Olsen. Power Management for Portable Consumer Electronic Devices. *Mentor Graphics Corporation*, 2008.

[QW04] Markus Quaritsch and Thomas Winkler. Linux - Ein Einblick in den Kernel. 2004.

[Rut09] S. Ruth. Green IT More Than a Three Percent Solution? *Internet Computing, IEEE*, 13:74–78, 2009.

[SAHE02] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. Energy-Efficient Mapping and Scheduling for DVS Enabled Distributed Embedded Systems. *Design Automation and Test in Europe*, 2002.

[SGD09] Joachim Schröder, Tilo Gockel, and Rüdiger Dillmann. *Embedded Linux - Das Praxisbuch.* Springer, 2009.

[She08] Findlay Shearer. *Power Management for Portable Devices.* Newnes, 2008.

[SJK05] Dongkun Shin and Member Jihong Kim. Intra-Task Voltage Scheduling on DVS-Enabled Hard Real-Time Systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 24:1530 – 1549, 2005.

[SKC04] J. Seo, T. Kim, and K. Chung. Profile-Based Optimal Intra-Task Voltage Scheduling for Hard Real-Time Applications. *Design Automation Conference*, 2004.

[SKL01] D. Shin, J. Kim, and S. Lee. Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications. *IEEE Design and Test of Computers,*, 18:20 – 30, 2001.

[SRH05] David Snowdon, Sergio Ruocco, and Gernot Heiser. Power Management and Dynamic Voltage Scaling: Myths and Facts. 2005.

[Tan03] Andrew S. Tannenbaum. *Moderne Betriebssysteme.* Pearson Studium, 2003.

[Tex97] Texas Instruments. *CMOS Power Consumption and Cpd Calculation*, 1997.

[TTD03] David Tam, Winnie Tsang, and Catalin Drula. Dynamic Voltage Scaling in Mobile Devices. 2003.

[Tur06] Jim Turley. Operating Systems on the Rise. *Embedded Systems Design*, 2006.

[VM03] G. Varatkar and R. Marculescu. Communicationaware Task Scheduling and Voltage Selection for Total Systems Energy Minimization. *International Conference on Computer-Aided Design*, 2003.

[VvSGH09] Ruud Vullers, Rob van Schaijk, Bert Gyselinckx, and Chris Van Hoof. Is There a Sweet Spot for Energy Harvesting? *Device Research Conference*, pages 7–8, 2009.

[XIL10] XILINX. *http://www.xilinx.com/*, 2010.

[YDS95] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. *IEEE Symposium on Foundations of Computer Science*, 1995.

[ZHC02] Y. Zhang, X. Hu, and D. Z. Chen. Task Scheduling and Voltage Selection for Energy Minimization. *Design Automation Conference*, 2002.