Mario Bambagini

Energy Saving in Real-Time Embedded Systems

Scuola Superiore Sant'Anna
di Studi Universitari e di Perfezionamento

Anno Accademico 2013-2014
Corso di perfezionamento
Innovative Technologies

Energy Saving in Real-Time Embedded Systems

Autore
Mario Bambagini
Tutor
Giorgio Buttazzo

ISBN: XXXXXXXXXXXX

Abstract

This thesis addresses the energy-aware scheduling issue in real-time embedded systems, while considering single-core, multi-core and distributed architectures.

In actual computational platforms, the energy consumption is rapidly growing due to the continuous technological improvements which let designers deploy far more transistors per inch, increasing the provided facilities. Besides the higher cost to power the electronic systems, the higher energy dissipation represents a serious design issue as such energy is transformed in heat which, if not effectively dissipated, may increase the probability of faults and shorten the overall lifetime.

In real-time systems, the data correctness does not depend only on the their values, but also on when they are produced. In other words, real-time schedulers are in charge of guaranteeing that jobs are successfully executed by their deadlines. Such time-critical systems are widely used in industry to increase the predictability and reliability, spreading from soft real-time systems (e.g.: video-conference software) where a deadline miss affects only the quality of service, to hard real-time systems (e.g.: flight controllers) in which consequences may be harmful.

Such a scenario introduces an interesting trade-off between time requirements associated to the workload execution and energy dissipation. On one hand, real-time applications require high performance to meet timing constraints. On the other hand, the energy is minimized by either lowering the performance level or switching the system off, which both lead to a delay in the workload execution. According to such assumptions, schedulers must judiciously grant the appropriate amount of computational resources to the pending workload in such a way their deadlines are guaranteed and the overall dissipation is reduced, at the same time.

Concerning single-core platforms, we proposed an innovative approach which, exploiting a different task model, significantly beats the state of the art. Then, the energy-saving problem is enhanced by considering bandwidth requirements which features sensor nodes and a scheduling algorithm is provided to handle it. In addition, the actual beliefs about the energy issue are pragmatically put in discussion and analyzed from a practical point of view.

Multi-core platforms have also been taken into account, providing an analysis of several partitioning heuristics.

Finally, the problem of partitioning a real-time workload on distributed systems was taken into account, dealing also with fault tolerance issues rather than only with real-time and energy constraints.

The analysis starts motivating the problem in Chapter 1, detailing the reasons behind the energy dissipation and introducing the actual solutions which aim at keeping it under control. Then, Chapter 2 considers the power and workload models which have been considered, whereas Chapter 3 reports the state-of-the-art algorithms which addressed the same problem. The analysis proceeds with the description of the proposed solutions for single-core, multi-core and distributed systems which are reported in Chapter 4, Chapter 5 and Chapter 6, respectively. In addition, Chapter 7 addresses the real-time scheduling issue for systems with renewable energy. Finally, Chapter 8 concludes the thesis remarking the main results.

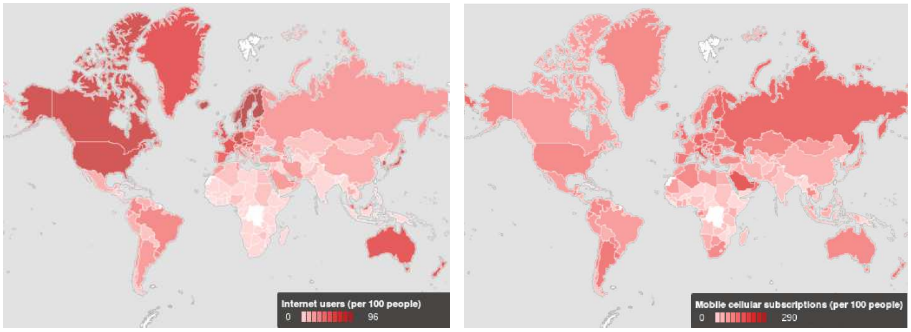
Contents

Chapter 1

Introduction

In the last decades, the *Information Technology* has grown exponentially and enhanced many fields, spreading from consumer electronics (e.g., desktops and notebooks) to specific sectors, such as embedded and high performance domains.

According to the data published by the World Bank [WBL] referred to the period 2009-2013, the numbers of internet users and mobile devices (represented in Figure 1.1(a) and Figure 1.1(b), respectively) have reached astonishing peaks in developed and BRIC countries. More precisely, in north America, the 96% of the population own an internet subscription and, in the eastern part of the world, almost three mobile devices belong to each person. Moreover, an interesting grown features also the developing countries, as such technologies represent a profitable market and an extraordinary tool for improving the quality of life.



(a) Internet users per capita (b) Mobile phones per capita

Figure 1.1: World Bank statistics for the period 2009-2013.

However, the previous numbers state only the actual facts. Scientists [Gat07,Gor13] agree unanimously on the fact that such trends are steadily growing and can not be stopped or even slowed down. Hence, we are part of a world whose energy demand will keep rising constantly regardless the development level of each country. In order to make such an aspect more clear, let us consider how the energy consumption has varied in the U.S. houses in almost 15 years, as reported by the Residential Energy Consumption Survey [REC] and depicted in Figure 1.2. The average consumption due

to electronics has rose up of an additional 10%, passing from 24% to 34.6%.

For such reasons and others, which are unfortunately out of the scope of this thesis, the energy consumption is a crucial point which must be effectively addressed in current systems. More precisely, this work focuses its attention on the real-time scheduling issue of systems in which the software execution is characterized from time constraints.

This chapter proceeds analyzing the problem from an engineering point of view, detailing the technical aspects which feature the energy dissipation (Section 1.1) and giving a glance at the actual solutions (section 1.2).

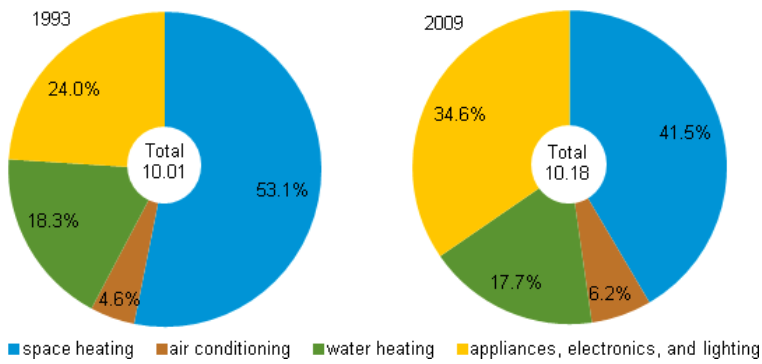


Figure 1.2: Results of the Residential Energy Consumption Survey.

1.1 Energy issue

The most widespread electronic technology in actual digital circuits is CMOS (*Complementary Metal-Oxide Semiconductor*), whose peak power dissipation happens during the state transitions of the transistors.

Intuitively, the higher the number of implemented features, the higher the number of transistors in the system and, consequently, the higher the power dissipation. In other words, a higher request of energy is the direct consequence of the increased performance. This is the side effect of the Moore's empiric law which states the growth of the number of transistors in the processors, as showed in Figure 1.3: *the number of transistors in a chip doubles every eight months*.

For such a reason, the frequency scaling feature has been introduced for enabling the application level to scale the performance down in order to reduce the transition rate and then, the consequent energy dissipation. Other techniques which are widely exploited at design time consists of implementing different physical parameters (such as supply and threshold voltage) to characterize particular circuits with less strict performance requirements. However, such techniques mainly concern design-time strategies which are out of the scope of this work.

Another side effect of the increased power consumption is related to the fact that most power is converted in heat which must be effectively dissipated. The trend of heat density is depicted in Figure 1.4, representing the not easy task which designers have to deal with. Besides the intuitive drawback of physically dissipating such heat by appropriate cooling systems, high temperatures drastically reduces performance (scattering effect), leading to the infamous *power wall*. In other words, it represented a stall

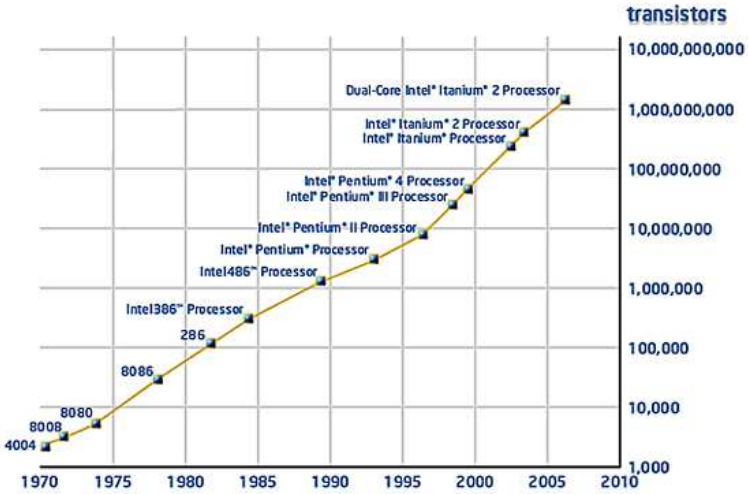


Figure 1.3: Moore's law about the transistor number in processors.

point in which any attempt to improve performance by either adding more transistors or increasing the frequency, caused an overall powerfulness reduction.

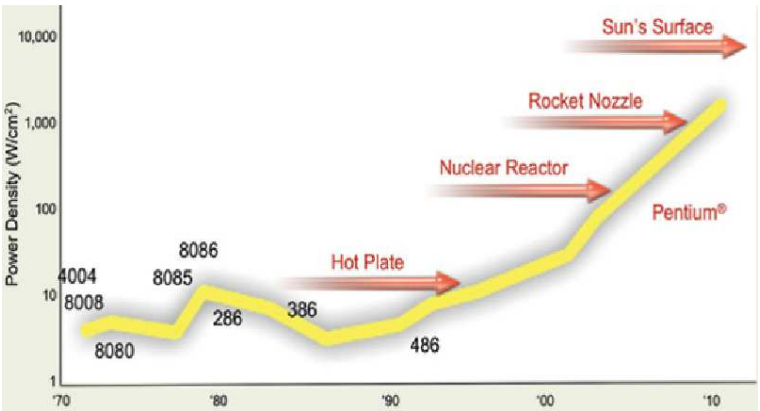


Figure 1.4: Heat density in Intel single-core processors.

Along with the memory and ILP walls, such limitations led to the adoption of multi-core systems. In such architectures, although cores run at lower frequencies, the overall performance is boosted up by the parallel execution of threads.

However, single-core systems are still widely used in the embedded system domain which generally requires lower performance but higher predictability and reliability.

As a consequence of progresses in the VLSI manufacturing, miniaturization has considerably shrunk the transistor size, lowering the supply voltage, thereby reducing the dynamic power consumption. Although the threshold voltage has also been lowered, the gap between supply and threshold voltages was reduced. This led to a significant increase in the leakage consumption, because the smaller the gap, the higher

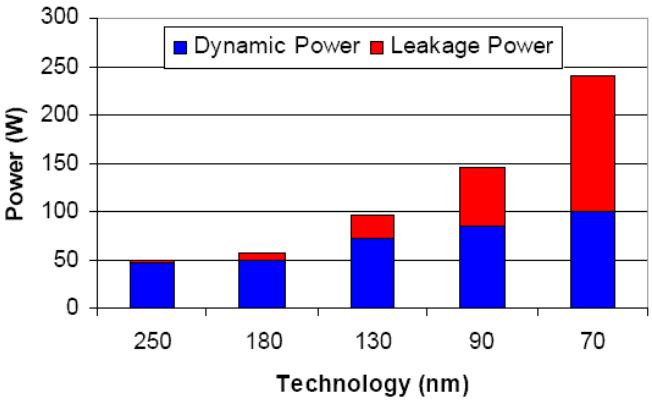


Figure 1.5: Dynamic and leakage dissipation for different technology processes.

the sub-threshold dissipation [SPG02, NC10, KSH02]. Thus, the static power consumption, whose contribution was considered negligible, has become as important as the dynamic power. More precisely, the leakage consumption is ascribable to quantum phenomenons and it is always present, not depending on the system activity. Figure 1.5 reports the contribution of the leakage and dynamic dissipation for different technology processes in Intel [INT] CPUs, remarking the rising impact of the static consumption.

To address such an issue, several low-power states have been introduced to let the system reduce the power dissipation when there is no workload to execute. More precisely, such states suspend the code execution and switch off several components of the system: the more components are disabled, the lower the power consumption. However, the required time to enter and exit the low-power state depends on the number and kind of asleep components, which may introduce conflicts with the time requirements of the applications. For such a reason, actual processors provide a wide range of low-power states, characterized from different consumption and overhead.

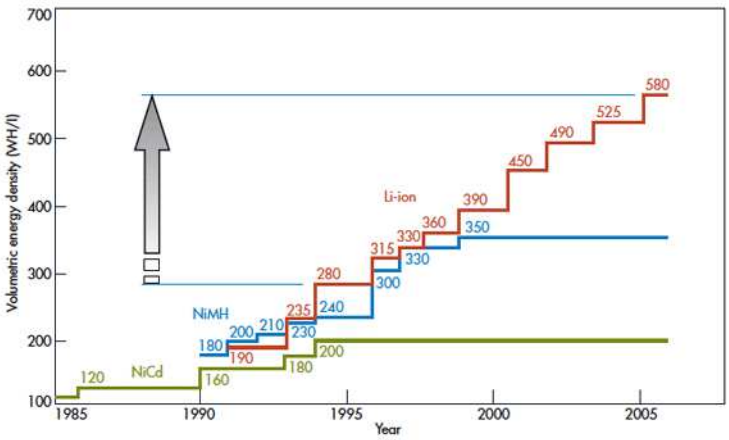


Figure 1.6: Growth of battery capacity in 20 years.

The last note, but not least, is related to the gap between required and supplied power for battery-operated platforms. Within a certain margin of approximation, we can assert that the power consumption broadly doubles every eighteen months as a consequence of the Moore's law, however, the battery capacity doubles every ten years, as highlighted in Figure 7.11. In other words, two different improving paces feature power suppliers and consumers, leading to a gap which should be addressed by software or by other mechanisms.

1.2 Energy management systems

Nowadays, many end-user and specific-purpose processors provide hardware facilities which attempt to automatically reduce the power dissipation through frequency throttling and low-power states, such as Intel's *SpeedStep* and AMD's *PowerNow!* [AMD].

However, much effort has been profused to address the problem at a more abstract layer in order to implement strategies which are hardware independent.

The first attempt to keep energy consumption under control was Advanced Power Management (APM), introduced in the 1992 by Intel and Microsoft [MIC]. The main purpose consisted of increasing the battery duration in notebooks which started to become popular in that period. The component allows APM-aware applications to communicate their energy requirements to the APM-driver provided by the operating system. The requests are subsequently supplied to the APM-aware BIOS, in charge of managing the APM-compliant hardware. The main drawback of this approach is that figuring out and requesting a specific power management policy from the operating system becomes essentially a responsibility of the application – ideally power management should be transparent to the individual applications. However, one of its most important contribution was the introduction of APM-compliant hardware to support a set of different states with specific power consumption and functionality features.

In 1996, Intel, Microsoft, and Toshiba [TOS] released an enhanced framework, called *Advanced Configuration and Power Interface* (ACPI), which has became the standard de facto for device configuration and monitoring. In particular, ACPI offers the operating system (in charge of handling the settings and power state transitions) an easy and flexible interface to discover and configure the compliant devices. The main contribution consists of having moved the policies from the application level to the operating systems (OSPM - Operating System Power Management), implementing a better abstraction. For instance, unused system components, as well as the entire system, can be switched to a low-power state, according to the current state and user preferences transparently to the running processes.

Despite the ACPI approach is very flexible and effective for many general purpose systems, it is considerably expensive in terms of computation and memory requirements for small footprint systems. For such a reason, in the embedded systems domain, the problem is mostly addressed in a tailored way in order to deal with the limited resources. However, Brock and Rajamani [BR03] proposed a solution with a set of pre-defined policies. The tasks are divided into groups according to either their energy requirements or importance classes. The current policy is chosen by the *policy manager*, a component provided by the system designer. The system behavior is encoded as a grid, where each cell represents the configuration to be adopted when a task of a specific group runs on the processor and a policy is active.

Chapter 2

System model

This section presents the most relevant models used in the literature for the design and analysis of energy-aware scheduling algorithms. Specifically, Section 2.1 overviews various power models, and Section 2.2 presents the computational workload models.

2.1 Power model

The power consumption of a single gate in CMOS technology has been modeled accurately in the literature [CSB95]. Specifically, the power consumption P_{gate} of a gate is expressed as a function of the supply voltage V and clock frequency f:

$$P_{gate} = p_s C_L V^2 f + p_s V I_{short} + V I_{leak} \tag{2.1}$$

where C_L is the total capacitance driven by the gate, p_s is the gate activity factor (i.e., the probability of gate switching), I_{short} is the current between the supply voltage and ground during gate switching, and I_{leak} is the leakage current, which is independent of the actual frequency and system activity. The three components of the sum in Equation (2.1) correspond to *dynamic, short circuit* and *static* power components, respectively.

In essence, the dynamic power is the power required to load and unload the output capacitors of the gates. Unlike the dynamic component, the short circuit current I_{short} depends on the temperature, size, and process technology. The leakage current is a quantum phenomenon where mobile charge carriers (electrons or holes) pass by tunnel effect through an insulating region, leading to a current that is independent from switching activity and frequency. Such a dissipation is due to three causes: gate leakage (from gate to source loses), drain junction leakage (losses in the junctions) and subthreshold current (from drain to source losses).

In Equation (2.1), the two variables that do not depend on the physical parameters are the supply voltage V and the clock frequency f. However, they are not completely independent, because the voltage level limits the highest frequency that can be used: the lower the voltage, the higher the circuit delay. Specifically, the circuit delay is related to the supply voltage V by the following equation:

$$circuit\ delay = \frac{V}{(V - V_T)^2} \tag{2.2}$$

where V_T denotes the *threshold voltage*, which is defined as the minimum voltage needed to create a channel from drain to source in a MOSFET transistor.

In the literature, the processor is assumed to be able to dynamically scale the clock frequency f in a given range $[f_{min}, f_{max}]$. Often, the analysis is performed by replacing the clock frequency by the processor *speed* s, defined as the normalized frequency $s = f/f_{max}$, so that the maximum processor speed is considered as $s_{max} = 1.0$. In the existing work, we observe two main approaches: in one research line, the frequency/speed range is assumed to be of fine granularity; i.e., the system's frequency is considered *continuous*. As opposed to this *ideal* model, the second research line considers m *discrete* speed/frequency levels. This line is based on the observation that the current processors offer typically a small number of discrete speed levels.

To characterize the power consumption $P(s)$ of the single-core system as a function of the processor speed, one of the most general formulations has been proposed by Martin and Siewiorek [MS01]:

$$P(s) = K_3 s^3 + K_2 s^2 + K_1 s + K_0. \tag{2.3}$$

The K_3 coefficient expresses the weight of the power consumption components that vary with both voltage and frequency. The second order term (K_2) captures the non-linearity of DC-DC regulators in the range of the output voltage. The K_1 coefficient is related to the hardware components that can only vary the clock frequency (but not the voltage). Finally, K_0 represents the power consumed by the components that are not affected by the processor speed.

Another variant of Equation (2.3) used in literature (e.g., [ZA09b]), is

$$P(s) = P_{ind} + P_{dyn}(s) \tag{2.4}$$

where the power dissipation is explicitly divided into static (P_{ind}) and dynamic ($P_{dyn}(s)$) power components. P_{ind} is assumed to be independent of the system speed, and P_{dyn} is assumed to be a polynomial function of the speed s.

A more specific power model (e.g., [BBL09]) considers the set of operating modes supported by the processor. Each mode is described by three parameters: the frequency f, the lowest voltage V that supports that frequency level, and the corresponding power consumption. To some extent, Martin's equation can be considered a generalization of this model, as it provides an interpolation of the various operating points on an ideal processor where the speed/voltage can be adjusted in a continuous manner.

Switching from one speed level to another involves both a *time* and *energy* overhead. These overheads depend both on the original and final speed levels (e.g., [XMM07, MHQ07]). When scaling the speed, the execution is suspended and the overhead is mostly due to the time required to switch the crystal on and/or adjusting the Phase-Locked Loop (PLL). Generally, the wider the difference between the two frequencies, the higher the introduced overhead. In this work, the notation $\mu_{s_1 \rightarrow s_2}$ denotes the time overhead when transitioning from the speed level s_1 to the speed level s_2.

When the leakage power dissipation is not negligible (i.e., $K_0 \neq 0$ and $P_{ind} \neq 0$ in Equation (2.3) and Equation (2.4), respectively), scaling the system speed down also increases the computation times and leakage energy consumption, which in turn may *increase* the total energy consumption. To address this issue, the concept of *critical speed* (also known as the *energy-efficient speed*), denoted by s^*, was introduced to denote the lowest available speed that minimizes the total energy consumption, which consists of dynamic and static power figures (e.g., [ADZ06, CK06]). Specifically, if we assume $P(s)$ as in Equation (2.3), it becomes strictly convex, and s^* is defined as the

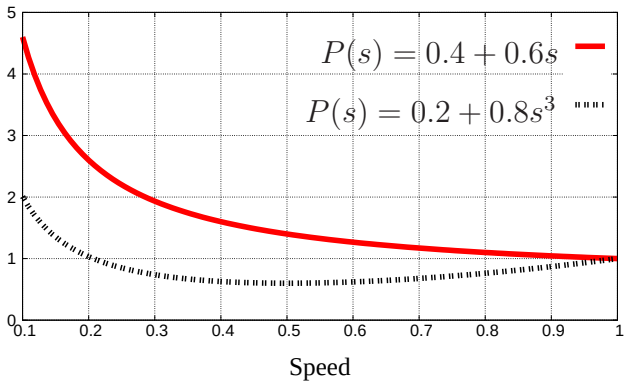


Figure 2.1: $\frac{\delta P(s)/s}{\delta s}$ function of DPM- and DVFS-sensitive models.

lowest speed that minimizes the energy consumption per cycle, which is equivalent to the speed value that makes the derivative of $P(s)/s$ null.

For instance, let us consider the power function $P(s) = 0.2 + 0.8s^3$. The derivative of $P(s)/s$ is $\frac{\delta P(s)/s}{\delta s} = 1.6s - 0.2/s^2$ which is null for $s = s^* = 0.5$, implying that, scaling the speed below 0.5 is not energy-efficient. This can be easily shown by considering a task with WCET = 10 time units while assuming that it can be executed at any speed $\in \{0.2, 0.5, 0.7, 1.0\}$ without missing its deadline. The relative energy consumptions for executing the task at the different speed assignments are: $E(0.2) = P(0.2) * 10/0.2 = 10.32$, $E(0.5) = P(0.5) * 10/0.5 = 6$, $E(0.7) = P(0.7) * 10/0.7 = 6.8$ and $E(1.0) = P(1.0) * 10 = 10$. The minimum energy consumption is indeed obtained for s^*, while it increases at both lower and higher speeds. One can see that the energy consumption of a task is a quadratic function with global minimum at s^*. A model whose critical speed is lower than the maximum one is identified as DVFS-sensitive, whereas a DPM-sensitive architecture is characterized by $s^* = 1.0$. Figure 2.1 shows the $\frac{\delta P(s)/s}{\delta s}$ function of the previously considered model $\left(P(s) = 0.2 + 0.8s^3\right)$ in contrast with a DPM-sensitive model $(P(s) = 0.4 + 0.6s)$. It is worth noting that such an analysis minimizes only the energy consumption during the time intervals when tasks are executed, because it implicitly assumes a negligible power consumption during the CPU idle intervals.

An additional feature provided by almost all the current processors is the ability to switch to *low-power* states when the task execution is suspended. Each low-power state σ_x is characterized by its power consumption (P_{σ_x}) and the time and energy overheads involved in entering and exiting that state, denoted as $\delta_{s\to x}$, $\delta_{s\to x}$, $E_{s\to x}$ and $E_{x\to s}$, respectively. For the sake of simplicity, we use the overall time and energy overheads associated with the low-power state σ_x, namely δ_{σ_x} and E_{σ_x}, as the sum of the initial and final transition overheads. In general, the "deeper" a low-power state, the lower the power consumption, but also the higher time and energy overheads involved in the transition. An exhaustive analysis of the low-power states in actual architectures has been undertaken by Benini et al. [BBDM00].

Considering the time and energy overheads involved in transitions to low-power states, there is, in general, a minimum time interval that justifies switching to a specific low-power state – this is because, if the system returns to active state too quickly, the energy overhead of the transition would offset the power savings of the low-power state. Consequently, the parameter B_{σ_x}, referred to as the *break-even time*, corresponds to the

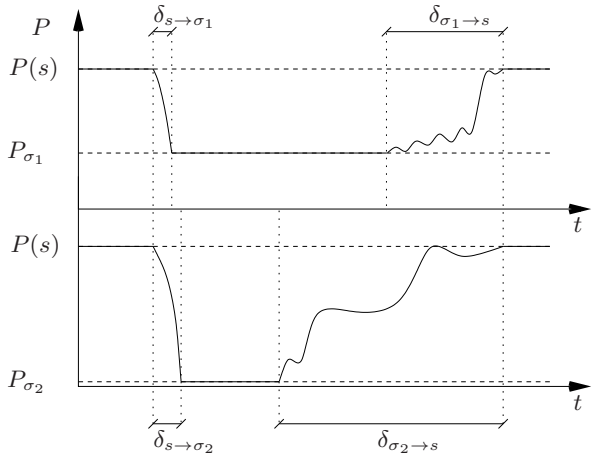


Figure 2.2: An example with two low-power states.

length of the shortest idle interval that must be available in the schedule to effectively exploit the sleep state σ_x. Specifically, B_{σ_x} is the maximum of the time required to perform a complete transition and the minimum idle time length that can amortize the switching energy (e.g., [QNHM04, ZA09a]):

$$B_{\sigma_x} = \max\left(\delta_{\sigma_x}, \frac{E_{\sigma_x} - \delta_{\sigma_x} \cdot P_{\sigma_x}}{P_{ref} - P_{\sigma_x}}\right) \tag{2.5}$$

where P_{ref} is the power consumption of the processor in a default state when tasks do not execute. For instance, P_{ref} can be the power consumption of a particular inactive state which requires a negligible transition overhead, or, in case the processor is kept active during idle intervals, it may correspond to the power consumption at the minimum speed level.

Different low-power states are characterized by different parameters. Figure 2.2 illustrates two different state transitions. The first case illustrates a low-power state σ_1 with a medium power consumption and a relatively short break-even time. On the other hand, the second low-power state σ_2 guarantees the lowest power consumption but introduces a significant temporal overhead from active to sleep and back to active. Finding the most suitable low-power state depends on the length of the available idle interval which, in turn, is determined by the timing constraints.

When multi-core processors are considered, the overall power consumption is function of the particular state and speed of each core. More precisely, the energy consumption of the entire system in the interval $[t_1, t_2]$ can be expressed as:

$$E(t_2, t_1) = E_{CPU}(t_2, t_1) + E_{NO\_CPU}(t_2, t_1). \tag{2.6}$$

where E_{CPU} denotes the energy dissipated by the processors and E_{NO_CPU} the energy dissipated by the remaining components, including the main memory, disks, network interfaces and other peripherals whose behaviors can be considered not directly affected by the running frequency of the processors. Although running tasks impact on such devices, for the sake of simplicity, we assume a constant average device dissipation, that is:

$$E_{NO\_CPU}(t_2, t_1) = (t_2 - t_1) \cdot P_{NO\_CPU},$$

where P_{NO_CPU} is the power consumed by the devices. In other papers, P_{NO_CPU} is also referred to as P_{uncore}. Since each processor can have a different state/speed at any time, the processor energy dissipation is the integral of power consumption in the interval $[t_1, t_2]$:

$$E_{CPU}(t_2, t_1) = \sum_j \int_{t_1}^{t_2} P_{CPU_j}(t)dt.$$

Basically, the processor power consumption at time t is a function of the actual state of each processor: the low-power state in use if it is asleep, or the running speed if active.

To characterize the power model of the processors, the equations concerning the active power consumption (Equation (2.3) and Equation (2.4)) can be extended to handle m different variables for each core (e.g., [CH14]). However, considering the scenario in which several cores are put in one of the low-power state while the remaining units are active is not representable with the previous approach. In order to provide a representation for such configurations, an effective solution, in spite of its simplicity, consists of tabling the power dissipation for a subset of all the possible scenarios (e.g., [BBB13]).

2.2 Workload model

In hard real-time systems, the computational workload is typically characterized by a set Γ of n *periodic* or *sporadic* tasks $\{\tau_1, \tau_2, ..., \tau_n\}$. Each task τ_i is cyclically activated on different input data and therefore generates a potentially infinite sequence of instances $\tau_{i,1}$, $\tau_{i,2}$, ..., referred to as *jobs*. The jobs of a *periodic* task τ_i are regularly separated by a *period* T_i, so the release time of a generic job $\tau_{i,k}$ can be computed as

$$r_{i,k} = \Phi_i + (k-1)T_i$$

where Φ_i denotes the activation time of the first job, also referred to as the *task offset*. On the other hand, in the case of *sporadic* task τ_i, the period T_i indicates the *minimum inter-arrival time* of its jobs: $r_{i,k+1} \geq r_{i,k} + T_i \ \forall k$. A real-time task τ_i is also characterized by a *relative deadline* D_i, which specifies the maximum time interval (relative to its release time) within which the job should complete. Depending on the specific assumptions, relative deadlines can be less than, equal to, or greater than periods. In the most common case, the relative deadlines are equal to periods, which is commonly called as *implicit-deadline task sets*. Once a job $\tau_{i,k}$ is activated, the time at which it should finish its execution is called the *absolute deadline* and is given by $d_{i,k} = r_{i,k} + D_i$.

Each task τ_i is also characterized by a *Worst-Case Execution Time (WCET)* $C_i(s)$, which is a function of the processor speed. In a large body of works, WCET is considered to be fully scalable with the speed, i.e., $C_i(s) = C_i/s$. However, a number of research works (e.g., [SAMR03, ADZ06]) noted that this is only an upper bound, because several I/O and memory operations are performed on devices and memory units that do not share the clock frequency with the CPU. For instance, if a task moves data to/from hard disk drive, the operation depends mostly on the bus clock frequency, the hard disk reading/writing speed, and the interference caused by other tasks accessing to the bus. To take the speed-independent operations into account, the task's WCET can be split into a fixed portion C_i^{fix} not affected by speed changes and a variable portion C_i^{var} which is fully scalable with the speed. Hence,

$$C_i(s) = C_i^{fix} + C_i^{var}/s.$$

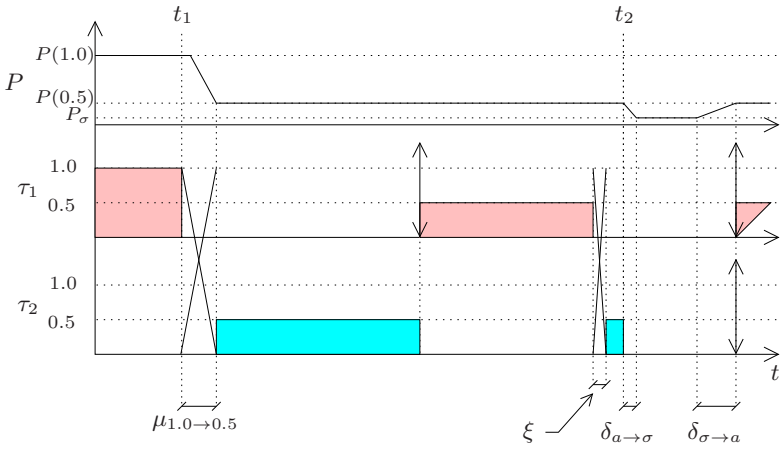


Figure 2.3: An example task schedule and corresponding power consumption.

An equivalent representation (e.g., [BBMB13a]) which is often used to model such feature is

$$C_i(s) = \alpha C_i^{max} + (1-\alpha)C_i^{max}/s,$$

where C_i^{max} is the overall execution time at the maximum speed ($C_i^{max} = C_i(1.0) = C_i^{fix} + C_i^{var}$) and $\alpha = C_i^{fix}/C_i^{max}$ is the fraction of speed-independent code.

In terms of CPU scheduling, tasks may be assigned a *fixed-priority* level, representing the relative importance or urgency of the task with respect to the others. In systems with dynamic priorities, the priority levels of jobs of a given task may vary over time: for instance, with the *Earliest-Deadline-First* (EDF) policy [LL73], the priorities are determined according to the absolute deadlines of the current active jobs of the periodic tasks, and hence, naturally vary over time.

In most algorithms, tasks are assumed to be fully *preemptive*, meaning that they can be suspended at arbitrary points in favor of higher-priority tasks. Preemption simplifies the schedulability analysis, but introduces a runtime overhead ξ (preemption cost) during task execution, which includes several penalties such as the context switch cost, the pipeline invalidation delay, and the cache-related preemption delay. The preemption cost is often assumed to be constant and speed independent. On the other hand, *non-preemptive scheduling*, while characterized by negligible runtime overhead, introduces significant blocking delays on high priority tasks that heavily penalize schedulability.

To visualize the power consumed during task execution, the scheduling diagram is typically extended by representing the power consumed by a task on the vertical axis. Consequently, the total energy $E(t_a, t_b)$ consumed by a task in an interval [t_a, t_b], which is the integral of the power function during the interval, is given by the corresponding execution area. Figure 2.3 illustrates the schedule of two tasks where, at time t_1, the speed is changed from 1 to 0.5, and at time t_2 the processor enters a sleep state. The speed scaling overhead $\mu_{1.0\to0.5}$ and the preemption cost ξ are also shown in the diagram.

Chapter 3

Related work

This section introduces the most relevant algorithms which have addressed the energy-aware scheduling issue in real-time systems.

Several surveys on energy-aware scheduling have been published recently, their primary focus was on DVFS algorithms. For instance, Chen and Kuo [CK07] addressed single and multi-core systems, by classifying algorithms according to the task periodicity. Similarly, Kim [Kim06] surveyed the intra- and inter-task DVFS algorithms, by considering only single-core systems. Saha and Ravindran [SR12] reported a performance comparison of a number of single-core DVFS algorithms through their implementation in the GNU/Linux kernel [LIN]. More recently, [Mit14] has presented a general survey of energy management techniques for embedded systems, including also micro-architectural techniques. As an effort to provide a greater in-depth overview of the existing DVFS- and DPM-based algorithms, we surveyed in [BAB14] the most relevant energy-aware scheduling algorithms, focusing our research only on single-core systems.

This section broadly divides the algorithms in a first group which considers single-core platforms and a second one which deals with multi-core architectures. These two classes are presented in Section 3.1 and Section 3.2, respectively. A taxonomy is proposed for each group to further catalog algorithms and make the analysis easier. In addition, Section 3.3 reports several problems which are related to the energy awareness in real-time systems.

3.1 Energy-aware scheduling on single-core systems

This section considers energy-aware real-time scheduling algorithms for single-core systems which are organized according to the taxonomy illustrated in Figure 3.1. The algorithms are first classified along the DVFS and DPM dimensions, based on the primary power management technique that they use. The DVFS algorithms (Section 3.1.1) are then divided according to the type of *slack* (the unused CPU time) that they reclaim for scaling speed to save energy: *static, dynamic,* or *both*. Specifically, the algorithms that exploit only the *static slack* consider the residual processor utilization in the worst-case execution, whereas those that reclaim the *dynamic slack* take advantage of the difference between the worst-case and the actual execution time of the jobs. In other words, the DVFS algorithms that exploit the dynamic slack take advantage of the runtime variability of the workload, since in practice many real-time jobs complete

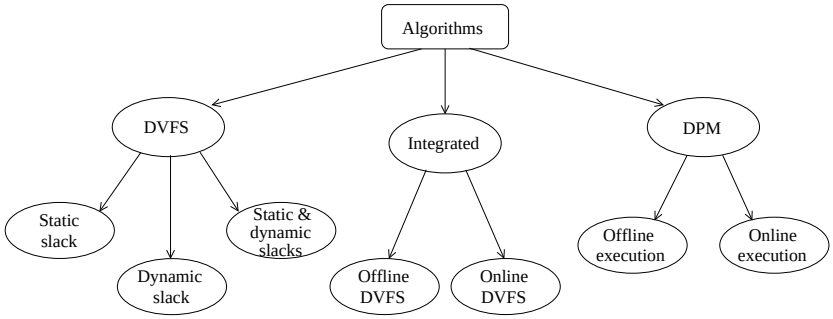


Figure 3.1: Algorithm taxonomy for single-core systems.

early (i.e., without presenting their worst-case workload).

Such a classification does not immediately apply to DPM algorithms (Section 3.1.2), since, due to their work-conservative nature, the dynamic slack is automatically accounted in almost all the cases. Thus, they are classified as *offline* and *online* approaches.

Finally, the algorithms that use both DVFS and DPM techniques are designated as *integrated* algorithms (Section 3.1.3). These algorithms are further divided according to when the speed assignment decisions are made, either *offline* or *online*.

3.1.1 DVFS algorithms

The DVFS-based algorithms rely on the system's capability of adjusting the processor supply voltage and frequency (hence, the speed) to reduce the power consumption, while still meeting the real-time constraints. Historically, such speed scaling techniques have been the first approach to face the energy management challenge, as in CMOS circuits the dynamic power consumption was recognized to be much more important than the leakage (static) consumption.

Most of the early DVFS algorithms have assumed a power function equal to $P(s) = s^{\alpha}$ ($2 \leq \alpha \leq 3$), implicitly ignoring the leakage power. Using such a power function, the lower the speed, the lower the consumed energy; hence, this power model favors the algorithms that use the lowest speed that can still meet the deadlines, leaving no idle intervals in the schedule.

The slack of a job refers to the CPU time that it does not use before its deadline. Hence, the *static slack* available to any job of a task τ_i can be computed offline as $slack_i = D_i - R_i$, where R_i is the worst-case response time of τ_i. At runtime, extra slack (referred to as *dynamic slack*) may become available when the job completes early, without consuming its WCET.

The DVFS solutions can be also classified as *inter-task* and *intra-task* algorithms. In inter-task algorithms, when a job is dispatched, it is guaranteed to execute at the same speed level until it completes or is preempted by another (high-priority) job. When it resumes execution (after preemption), the scheduler may re-adjust its speed by considering the available slack at that time. The *inter-task* algorithms form the majority of the current DVFS solutions, as it requires only the information about the WCET of the jobs, and involves low run-time overhead. On the other hand, if the information about the execution time of the job is available, in particular its *probability distribution*, then there may be benefits in adjusting the job's speed *while* it is in

progress, at well-determined points. This is the main idea behind the *intra-task* algorithms [XXMM04, XMM05, SKL01], in which the job starts to execute at a low speed level (relying on the fact that its early completion is more likely than the worst-case scenario), and then its speed is increased *gradually* at well-determined *power management points (PMPs)* as it continues to execute. Thus, for each task, a *speed schedule* is computed offline, showing what speed level will be assigned to its jobs during their execution, and at what point. The intra-task algorithms aim at minimizing the *expected dynamic energy consumption*, however they also require that the compiler generates code to enable the application to make system calls to the operating system at the well-determined PMPs during job execution, and they involve more overhead due to more frequent speed changes.

The rest of this section provides an overview of the most relevant DVFS algorithms, divided according to the type slack they exploit for scaling speed: static slack, dynamic slack or both.

Static slack reclaiming

One of the first papers on DVFS based energy-aware scheduling is proposed by Yao et al. [YDS95]. The paper presented three algorithms by considering aperiodic tasks, continuous CPU speed, no speed scaling overhead, negligible power consumption during idle intervals, and task computation times inversely proportional to CPU speed ($C(s) = C/s$). The first algorithm consists of recursive identification of time intervals with maximum *computational density* (defined as the sum of CPU cycles of the tasks with arrival and deadline within the interval, divided by the length of the interval length). Specifically, the algorithm identifies the interval with the maximum intensity, sets the CPU speed to the intensity value for that interval, and it is recursively re-invoked for the remaining execution intervals in the schedule. The offline algorithm is proved to be optimal and has an $O(n \log^2 n)$ complexity for n aperiodic jobs. A second algorithm, executed online, considers jobs that may arrive dynamically. The algorithm recomputes the optimal schedule at each arrival time considering only the new and pending jobs. The third algorithm (AVR) sets the speed, for each instant, equal to the sum of density of those jobs whose arrival and deadline range contains the time instant under consideration. Although the complexity of AVR is lower than the previous optimal approaches, deadline misses may occur. In fact, since the speed is set equal to the sum of the worst-case utilization of the active jobs, the processor can be significantly slowed down when there are few active tasks, so the system may not be able finish the remaining work if additional tasks arrive, leading to deadline misses.

Ishihara and Yasuura [IY98] provided an analysis for synchronous frame-based real-time tasks (with identical release time and period), proving that under their assumed system model (no overhead and all tasks consume the same amount of energy), the energy is minimized when each job completes just at its respective deadline. That result implies that on a system with continuous speed/voltage, the total energy is minimized at the speed/voltage that reduces the idle time to zero. While that result is also implicit in the optimal [YDS95] algorithm mentioned above, the main contribution of [IY98] is the derivation of an important property of the systems with discrete speed levels: when the system is constrained to use a finite set of speed/voltage, the energy is minimized by using the two speed/voltage values adjacent to the speed value that is optimal assuming a continuous range. When systems where tasks may have different power consumption characteristics are considered, an immediate result is that using a uniform speed across all tasks is longer optimal, and there is a need to adjust the CPU

speed at context switch times. In two of the earliest efforts,

The problem of finding an optimal solution on a system with discrete speed levels was discussed in [BBL09], for a set of periodic or sporadic tasks under both EDF and Fixed-Priority (FP) scheduling policies. The authors provided a method to compute the optimal speed offline (first assuming a continuous speed spectrum) and then introduced a speed modulation technique to achieve the target speed using two discrete values. The analysis selects the pair of available frequencies that minimize the energy consumption by also incorporating time and energy switching overheads. The execution time consists of a part that is speed-dependent and another one which is not.

Dynamic slack reclaiming

All the algorithms considered here are based on EDF and assume that the computational times scale linearly with the speed ($C(s) = C/s$).

Lee and Shin [LS04] proposed OLDVS, an algorithm which accumulates the dynamic slack due to early completions and exploits it to decrease the CPU speed so that the current task is completed at the same time that it would complete in the schedule with the worst-case workload. The idea was improved in [GSL07] through the intra-task algorithm OLDVS*, which divides each job execution in two parts: the first part is executed at a low speed level and the speed is increased if does not complete by the end of the first part. This approach relies on the observation that the probability of completing the job in the first part is significantly higher than finishing in the second half. Both algorithms assume a discrete set of speeds, negligible power consumption during the idle intervals, and zero switching overhead.

Zhu et al. [ZM05] combined the DVFS mechanism with feedback control theory to save energy for periodic real-time task sets with uncertain execution times. Their approach uses a PID controller to compute the estimated execution time of the next job as a function of the difference between the actual and the expected execution time of the previous job of the same task. The plant in the closed control loop is represented by the EDF scheduler. The frequency/voltage selection is greedy, as it considers the estimated execution time for the running task and WCET for the others. Moreover, the frequency spectrum is assumed to be continuous and the speed scaling overhead is considered negligible. It is also assumed that the CPU uses the lowest speed level during the idle intervals.

Lawitzky et al. [LSP08] implemented an energy saving algorithm based on the Rate-Based Earliest Deadline (RBED) framework [BBLB03], which supports CPU time budget allocation and dispatching. The paper took speed scaling overhead into account and offers a system-wide view by considering not only the CPU, but also bus and memory. The speed scaling overhead is automatically accounted within the CPU budget assigned to each task. In addition, the authors proposed to manage also the static slack which, otherwise, would be entirely allocated to non real-time tasks. Their proposal consists of increasing the utilization values of real-time tasks to exploit the entire remaining static slack, even though, the actual execution times are not changed. In such a way, at runtime, the overestimated utilization is automatically transformed into dynamic slack which is, in turn, easily handled within the presented framework.

Dynamic and static slack reclaiming

All the algorithms reported here consider periodic tasks whose computational times scale linearly with the speed ($C(s) = C/s$). Moreover, the speed scaling overhead is

considered negligible and the power consumption is modeled as $P(s) = \beta \cdot s^3$.

Pillai and Shin [PS01] proposed three algorithms considering both EDF and RM scheduling policies. The first approach, referred to as *Static Voltage Scaling (SVS)*, runs offline and exploits only the static slack: when the system starts, the running speed is set equal to the lowest available speed level which guarantees the task set feasibility. Then, the *cycle-conserving algorithm* (cc-EDF and cc-RM) is introduced. The algorithm, at every scheduling event, sets the running speed to the lowest level that guarantees timing constraints using the actual execution time for the completed jobs and the WCET information for future jobs. Notice that the cc-EDF algorithm generates a schedule identical to the SVS schedule if the actual workload is identical to the worst-case. The last proposed algorithm, called Look-Ahead RT-DVS (LA-DVS), runs only under EDF and aims at further reducing the running speed of the current (earliest-deadline) job as much as possible, while still guaranteeing the deadlines of other jobs. Hence, although the actual speed until the next deadline can be quite low, it may be necessary to execute future jobs at high speed levels to meet their timing constraints, in case the current job takes (close to) its WCET. However, this side effect is significantly reduced thanks to frequent early task completions in practice.

Aydin et al. [AMMMA04] proposed three algorithms at increasing complexity and sophistication levels, for periodic real-time tasks. All the algorithms assume a continuous speed range and a negligible switching overhead. The first algorithm computes the running speed as the utilization of the task set (similar to SVS) and it is not changed at runtime. The algorithm works with all the scheduling algorithms which guarantee the full utilization of the processor while guaranteeing the feasibility, such as EDF and Least Laxity First (LLF). The second algorithm *(Dynamic Reclaiming Algorithm, DRA)* uses a queue structure called α-queue where each element contains the deadline and the remaining execution time rem_i of task τ_i. When a task arrives, its absolute deadline and execution time at the optimal speed are inserted in the α-queue. At every scheduling event, the rem_i field of the α-queue's head is decreased by the amount of the elapsed time since the last event. In other words, the α-queue represents the ready queue in the worst-case schedule at that specific time. Once a new job is about to be scheduled, its remaining execution time is summed with the rem_i values in α-queue whose deadlines are less than or equal to the task in question, and then the speed is scaled accordingly. This procedure enables the current job to reclaim the dynamic slack of already completed higher-priority jobs, while still ensuring it does not complete later than the instant when it would complete in the worst-case schedule. The algorithm is improved by incorporating the *One Task Extension* (DRA-OTE) technique which, when there is only one task in ready queue and its worst-case completion time at the current speed falls earlier than next scheduling event, slows the speed down to let the task terminate at the next event. The third algorithm, *Aggressive Speed Reduction - AGR 1*, relies on the idea that when all the ready tasks have deadlines earlier than the next task arrival time, then the computational budget can be exchanged among those tasks without affecting the feasibility. Specifically, in such a situation the algorithm reduces the speed of the current job by allocating some of the CPU time of other low-priority ready tasks. This approach may force other pending tasks to execute at very high speed levels to meet their deadlines in some execution scenarios. To mitigate this, another algorithm (AGR-2) is proposed, which limits the extent of the slowdown for the current task by considering the information about the average case workload.

3.1.2 DPM algorithms

DPM-based energy management algorithms are based on the principle of putting the processor to low-power (sleep) states at runtime. A main problem involved in DPM research is to make sure that the transitions are beneficial in terms of energy savings, because as explained in Section 2.1, there is a minimum time interval (called the *break-even time*) that amortizes the time and energy overhead associated with each transition. In fact, a common technique is to use the *task procrastination* technique which postpones the execution of the ready jobs as much as possible by exploiting the system slack at that time, thereby compacting busy periods and yielding long idle intervals. By doing so, the number of runtime transitions and overhead are also reduced. On the other hand, utmost care must be taken to avoid the violation of the timing constraints in real-time systems, when employing the procrastination technique.

The rest of this section introduces the most interesting offline and online DPM approaches proposed in the literature

All the algorithms discussed in this section consider the break-even times for the CPU explicitly in their analysis. Although some papers consider only a single low-power state, we note that their approach can be easily extended to systems with multiple low-power states by exploiting the "deepest" inactive state with break-even time shorter than or equal to the length of the available idle interval.

Offline DPM algorithms

Rowe et al. [RLZR10] presented two techniques to harmonize task periods with the aim of clustering task executions (i.e., to combine processor idle times whenever possible). The framework assumes a system without the DVFS feature. The first algorithm, *Rate-Harmonized Scheduler (RHS)*, introduces the concept of *harmonizing period (T_H)*. The scheduler is notified by the task arrivals only at the integer multiples of the harmonizing period. The harmonized period is computed as a function of the shortest period. For instance, if the effective arrival time is at 3.5 and the harmonizing period is 1, then the scheduler considers this arrival only at time 4. Since all the arrivals are considered at integer multiples of the harmonizing period, if there is no task to execute, then the processor can be put in sleep state until the next period. The approach considered fixed-priority tasks whose priorities are assigned by the Rate Monotonic policy. Although the exact schedulability can be checked by evaluating the worst-case response time through the Time Demand Analysis, the utilization bound for schedulability reduces to 0.5, in the general case. The second algorithm, called *Energy-Saving RHS (ES-RHS)*, introduces a new task with period equal to T_H (highest priority). Its computation time is evaluated by considering T_H and the spare utilization. The new task enables putting the processor to sleep state when it is invoked and when its computational budget is longer than or equal to the break-even time. The main advantage of ES-RHS with respect to RHS is that the idle times generated by task early terminations extend the sleep interval in the next period. In such a way, multiple short idle intervals are combined to a single longer interval, giving an advantage over RHS. Two low-power states are taken into account, idle and sleep, considering a short and long break-even time, respectively. In addition, a real implementation on a sensor node is reported.

Online DPM algorithms

Lee et al. [LRK03] proposed two leakage control algorithms for procrastinating task executions as long as possible, to prolong and compact idle intervals, both under dy-

namic (LC-EDF) and fixed (LC-DP) priority scheduling. Both algorithms assume periodic tasks with periods equal to the deadlines and a system without DVFS feature. The main idea behind the algorithms is to compute at each job arrival the maximum time the job can be delayed without missing its deadline. Under EDF scheduling, whenever the CPU becomes idle, LC-EDF computes the maximum time duration Δ_k that the task with the earliest arrival time (τ_k) can be delayed by using the following equation:

$$\sum_{i \in \{1,\ldots,n\}/\{k\}} \frac{C_i}{T_i} + \frac{C_k + \Delta_k}{T_k} = 1.$$

Then, the system is put to the low-power state (procrastinated) for Δ_k time units. If another higher-priority task τ_j with absolute deadline shorter than the τ_k's deadline arrives before the end of the procrastination interval, the procedure is executed again, by considering the length of the idle interval already elapsed, δ_k, and obtaining the new value of the procrastination interval Δ_j through the following equation:

$$\sum_{i \in \{1,\ldots,n\}/\{k,j\}} \frac{C_i}{T_i} + \frac{C_k + \delta_k}{T_k} + \frac{C_j + \Delta_j}{T_j} = 1.$$

For fixed-priority systems, the authors resort to the *dual priority* scheme [DW95] in order to to compute the length of the procrastination interval. More precisely, the additional sleep time is computed as the minimum *promotion time* Y_i (relative deadline minus the worst-case response time) among the tasks in the lower run-queue. The promotion time of each task is computed statically as the difference between its relative deadline and the worst-case response time, derived from Time Demand Analysis. The main limitation of such an approach is that it requires a dedicated hardware to implement the algorithms and manage sleep and wake up operations. Although task early terminations are not directly involved in the analysis, the work-conserving (non-idling) nature of the algorithms can indirectly incorporate the dynamic slack at run-time.

Awan and Petters [AP11] proposed an algorithm under EDF, called *Enhanced Race-To-Halt (ERTH)*, which targets at dynamically monitoring and accumulating both static and dynamic slack, in order to apply the DPM technique effectively. The authors considered sporadic tasks with different criticality (hard, soft real-time and best effort) and a processor model with several low-power states. Essentially, the algorithm uses a single counter to keep track of both static and dynamic slack. When the system is idle, the processor is put to the deepest low-power state with break-even time not exceeding the amount of the existing slack at that time. Similarly, if there are some ready tasks, and the amount of available slack is longer than or equal to the break-even time, then the processor is switched off as long as possible without causing any deadline miss. On the other hand, if the amount of slack is less than the break-even times, the processor executes the current workload at the maximum speed and then attempts to switch to a sleep state when idle. The proposed algorithm has been compared with LC-EDF, showing that, under certain conditions, it achieves a lower energy consumption.

3.1.3 Integrated DVFS-DPM algorithms

This section considers the algorithms that use both DVFS and DPM techniques. Specifically, these *integrated* algorithms exploit both speed scaling and low-power states to maximize energy savings, unlike the techniques that use only one feature.

First, the algorithms that make the speed scaling decisions offline are considered then, those that compute the speed scaling factors online are introduced.

Offline speed scaling

All the algorithms reported here are designed for periodic real-time tasks and do not explicitly consider dynamic slack.

Jejurikar et al. [JPG04] proposed an approach (CS-DVS-P) based on the critical speed analysis and task procrastination, for periodic preemptive tasks executed under the EDF scheduling policy. Offline, the algorithm first computes the lowest speed (higher than or equal to the critical speed, s^*) that guarantees the task set feasibility. Then, the maximum amount of time (Z_i) each job of task τ_i can spend in the sleep state within its period without leading to any deadline miss is evaluated using the following equation:

$$\frac{Z_i}{T_i} + \sum_{k=1}^{i} \frac{C_k}{T_k} = 1.$$

At runtime, when there is no pending job, the processor is put in a low-power sleep state (as deep as justified by the break-even time and available slack) until the next job arrival. When a job arrives and the processor is still in sleep mode, an external controller continues to keep the processor in sleep state for an additional time period computed as the minimum of remaining time to wake-up and the precomputed delay of the newly arriving job.

[JG04] extended the algorithm to fixed-priority (CS-DVS-P1) and dual-priority (CS-DVS-P2) systems. With respect to the original algorithm given in [JPG04], only the computation of the Z_i values is different, leaving the online step the same. Moreover, the authors showed that the dual-priority scheduler is able to guarantee longer Z_i values than the fixed-priority scheduler.

Chen and Kuo [CK06] showed that the DPM part of the algorithm proposed by [JG04] may lead to deadline misses, thus they proposed two solutions to avoid them, *Online Simulated Scheduling (OSS)* and *Virtual OSS (VOSS)*. Both algorithms consider periodic independent tasks for fixed-priority systems where priorities are assigned according to the Rate Monotonic policy. Initially, all tasks are assigned the lowest speed that still guarantees the feasibility, subject to the lower bound of critical speed. OSS runs when the ready queue is empty and simulates the execution of tasks that arrive earlier than the earliest absolute deadline, accounting for their idle time. Then, the arrivals of those tasks are delayed for the relative accounted time, while the processor is put in sleep mode until the first job arrival (if and only if the available idle time is longer than the break-even time). VOSS enhances OSS by combining the online simulation with the virtual blocking time. Specifically, in the simulation phase, the algorithm considers as arrival time the value of $r_{i,k} + Z_i$ where Z_i represents the maximum blocking tolerance that each task can afford without causing deadline misses. Z_i is computed offline through the response time analysis. In this way, the arrivals of the tasks taken into account result further delays than those provided in OSS, leading to longer sleep intervals. The complexity of the online step is due to the simulation phase, which is $O(n \cdot \log(n))$, while the offline computation of the virtual blocks has pseudo-polynomial complexity.

Online speed scaling

Jejurikar and Gupta [JG05a] extended the algorithm in [JPG04] to explicitly consider task early terminations on dynamic priority systems. The algorithm is called *Dynamic Slack Reclamation with Dynamic Procrastination (DSR-DP)*. The first improvement

consists in collecting unused computation times (dynamic slack) in a *Free Run Time (FRT)* list, which also includes information of the priority of the task that generated it. To prevent any deadline miss, each job can only use the dynamic slack generated by tasks with higher or equal priority. Such additional CPU time is partially exploited to slow down the processor speed while the job is executing and also to extend the time spent in sleep state. Specifically, the slack distribution algorithm primarily uses the additional slack to scale the speed down and, if the critical speed is reached, the residual time is used to extend the sleep interval.

Irani et al. [ISG07] introduced two techniques for dynamic speed scaling with and without low power states: DSS-S and DSS-NS. DSS-NS is based on using mostly speed scaling while DSS-S executes the workload at the maximum speed to maximize the use of the low-power states. Both the $P(s)$ and $P(s)/s$ functions are assumed to be convex and the scheduler implements the EDF policy. An offline algorithm for DSS-S and two online solutions for DSS-S and DSS-NS were presented. The main idea behind the offline algorithm is to procrastinate tasks and execute them at a speed no lower than the critical speed. Under the assumptions of convexity, the proposed offline algorithm achieves an approximation ratio of 3 with respect to the optimal solution. However, the overheads due to the speed scaling and state transition are not taken into account.

3.2 Energy-aware scheduling on multi-core systems

This sections presents the state-of-the-art algorithms for multi-core platforms, exploiting the taxonomy shown in Figure 3.2. Algorithms are broadly divided according to the heterogeneity of the system they assume to deal with. More precisely, the first kind of algorithms assumes to handle a set of homogeneous cores whose performance and consumptions are identical, whereas the second group takes into account heterogeneous cores (with at least two different kinds of cores). Such algorithms are detailed in Section 3.2.1 and Section 3.2.2, respectively.

For each class, algorithms are further collected according to the approach they implement: partitioned, global or hybrid. The first one statically assigns a task to a specific core, forbidding its jobs to migrate onto another core even though it is idle. This method allows designers to easily check the system feasibility but, in many cases, it leads to a waste of computational resources. Global approaches improve the system utilization by allowing task migration at any time in any processor, but are more difficult to analyze and may introduce significant run-time overhead. Hybrid scheduling approaches try to combine the two previous strategies to reduce their drawbacks and exploit their advantages.

3.2.1 Homogeneous cores

One of the first papers which has considered the partitioning problem of a set of periodic tasks on a multi-core system was proposed by Aydin and Yang [AY03]. The authors compared the behavior of four well-known heuristics (First-Fit, Next-Fit, Best-Fit and Worst-Fit) on a system whose dissipation is highly dependent on the running speed. The work stated that Worst-Fit Decreasing (WFD), which aims at balancing the workload among the cores, is the most effective for reducing the energy consumption while considering cubic power functions. More precisely, spreading the workload among all the cores lets us use many processors which run at a low frequencies. On the other hand, collecting the workload on few cores (and switching off the others) is not

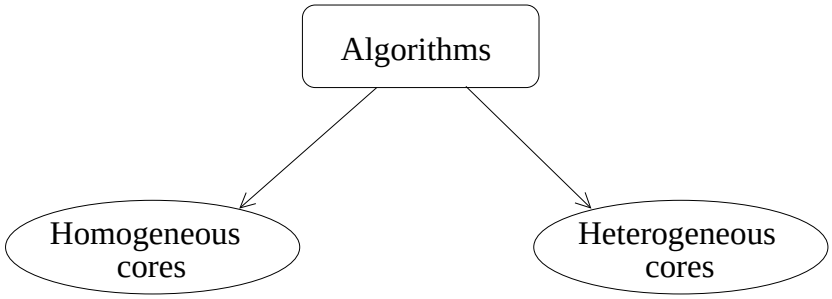


Figure 3.2: Algorithm taxonomy for multi-core systems.

an effective strategy as the energy dissipation of those few cores is much higher than having all cores active and running at low speeds.

Yang et al. [YCK05] proposed an algorithm which partitions a set of frame-based tasks (with same period and deadline) using the Worst-First strategy and then scales speed in particular instant according to the task features. In addition, all cores must share the same running frequency and voltage, meaning that the speed selection is done according to the most loaded CPU. However, cores are allowed to enter into a low-power state independently. Although the algorithm is characterized by a good approximation factor with respect to the optimal scheduling, the authors made several non-realistic assumptions, such as continuous and infinitive frequency range ($s \in [0, \infty]$) and negligible consumption in idle state.

Kandhalu et al. [KKLR11] considered the issue of partitioning a set of periodic real-time tasks on multi-core systems characterized by a single voltage island (all the processors share the same voltage and frequency). Moreover, deadlines are assumed to be equal to periods. Within a voltage island, the core with the highest load is the one which imposes the running frequency. In other words, the load balancing is crucial. Since it has been already proven that, under EDF, the overall energy dissipation is minimized when the load is perfectly balanced among the cores, the authors focused their attention on tasks with fixed priorities. They proved the approximation upper bound for the classical Worst-First Decreasing heuristic and then, their own algorithm (Frequency Assignment Algorithm - SFAA) was provided to overcome several limitation of the state of the art by taking explicitly task periods into account during the partitioning. Finally, the power model was obtained from a NVIDIA's Tegra 2 processor.

Ghasemazar et al. [GPP10] provided a global approach based on the control analysis to minimize the overall energy consumption while guaranteeing the desired throughput. The homogeneous cores, which share the L2 cache, are assumed to be switched off independently. When a new job arrives, the hierarchical scheduler computes the number of core to keep active and then a feedback control loop sets the frequency which lets the system provide the required throughput. Finally, the new job is assigned to a core which is in charge of executing it according to its priority.

Pagani and Chen [SJJ13] carried out an analysis that, independently from the task partitioning algorithm, found the approximation ratio of the Single Frequency Approximation (SFA) scheme on multicore voltage islands with respect to the optimal solution. More precisely, SFA consists of setting the frequency of the voltage island equal to the maximum utilization among the cores. Despite its simplicity, SFA is the easiest algorithm to be implemented and it is widely used in actual systems.

Unlike other papers, Langen and Juurlink [dLJ06] considered a multi-core proces-

sor whose cores can vary their frequency independently. The authors considered a scenario in which the leakage power consumption is as important as the dynamic dissipation and then, they integrated such a contribution into the analysis. More precisely, according to the actual value of the leakage current, the provided algorithm (referred to as LAMPS) computes the number of core to be kept active, their frequencies and voltages in order to minimize the energy dissipation. Results were obtained by comparing the energy consumption with the Schedule and Stretch ($S\&S$) algorithm which aims at spreading as much as possible the workload among the available cores (similarly to what WFD does).

Recently, Carrol and Heiser [CH14] have revisited several common beliefs, showing empirically that on the latest-generation ARM multi-core processors, the static power is almost negligible. First, they extended the concept of critical speed to each core, in order to define the frequency which locally optimizes the energy consumption. Then, a governor for the Linux *cpufreq deamon* was implemented whose algorithm executes the following steps every 100ms:

- computing the slowest frequency which guarantees the feasibility on each core;
- if such a speed is higher than the critical one, then wake an asleep core up;
- if such a speed is lower than the critical one, then put a core in sleep state.

3.2.2 Heterogeneous cores

Petrucci et al. [PLM$^+$12] considered the problem of partitioning a set of independent tasks on heterogeneous systems. Such systems are composed of a set of high performance sophisticated cores and a set of low-power cores. Their analysis relies on the fact that CPU-intensive applications should run on performing cores, while I/O applications should take advantage of low-power cores. More precisely, they introduced a periodic partitioning algorithm (implemented as ILP problem) which migrates tasks among cores according to their actual phase (interval in which the code is mostly either CPU or I/O intensive) in order to better exploit the energy efficiency of each core.

Awan and Petters [AP13] proposed a two-step partitioning algorithm for heterogeneous systems with two kind of cores. Firstly, the algorithm assigns tasks to the core which optimizes its execution while considering only the dynamic energy consumption. Then, the second step reduces the static energy consumption by improving the effective use of low-power states, involving tasks' parameters into the analysis (such as periods and execution times). For example, allocating a task with short period (despite its execution time) may prevent the system to put the core in a deep low-power state even though the overall utilization is low. For such a reason, tasks which forbid the use of deep sleep states should be moved somewhere else in order to reduce the static energy consumption.

Schranzhofer et al. [SCT10] proposed two different partitioning approaches for the problem in question. The first one assigns tasks to cores statically, independently from the actual context, in a static fashion. The second solution assigns a task to a core according to the actual scenario at runtime. More precisely, the mapping is chosen among several possible alternatives which have been pre-computed at design time. Although the first approach is less flexible than the second, it requires a smaller amount of memory to store the additional information (in the form of a mapping table).

Hung et al. [HCK06] addressed the energy-efficient real-time scheduling issue on systems mounting two cores with and without speed scaling feature, respectively. The

workload consists of a set of periodic real-time tasks and jobs which may migrate from one core to the other when necessary. When the power consumption of the non-DVFS cores is independent from the workload, a fully polynomial-time approximation scheme (FPTAS) is provided for energy-efficient scheduling. On the other hand, when the energy consumption of the non-DVFS cores depends on the assigned workload, a 0.5-approximation algorithm is introduced to save energy.

3.3 Similar problems

This section briefly overviews other problems related to the energy issue which consider additional objectives. Since the energy harvesting problem has been addressed in this thesis, a deeper analysis is provided for it in Section 3.3.1.

An interesting problem is related to the joint scheduling of real-time and non real-time tasks, where the goal is to minimize the overall energy consumption, while guaranteeing real-time constraints and reducing the response time of non real-time tasks. As in the case of real-time tasks, providing short response time to the non real-time tasks, in general, conflicts with the energy saving objective. Aydin and Yang [AY04] investigated the impact of speed scaling decisions on the responsiveness of non real-time tasks and overall energy consumption, while still meeting the timing constraints of hard real-time tasks. Saewong and Rajkumar [SR08] proposed to exploit the slack available in real-time tasks to execute non real-time tasks at the maximum speed to minimize their response time.

The energy-aware scheduling of tasks that share resources which must be accessed in non-preemptive fashion is another important problem that was addressed in [ZC04, LKL07, JG05b]. In [ZAZ12], the energy-aware scheduling of periodic real-time tasks with task-level reliability constraints has been considered.

A more general energy-aware co-scheduling problem includes both the CPU and devices in the analysis. Devices are typically considered speed independent, providing low-power states and requiring non-preemptive access [CG06, DA08, YCK07]. Other authors considered the problem of co-scheduling tasks and messages [YPH$^+$09].

Another variant of the problem occurs in settings where the real-time system has to operate within a given strict *energy budget*. This is also called the *energy-constrained* real-time scheduling problem. AlEnawy and Aydin [AA04] proposed several static and dynamic algorithms that select the most valuable jobs to execute in energy-constrained settings, while considering systems with and without speed scaling feature. Later, [AA05] reformulated and addressed the problem for *weakly hard* real-time systems, where each periodic task has to meet m out of k deadlines in k consecutive invocations. Chen and Kuo [CK05] addressed the problem of maximizing the total task execution time given an initial energy budget.

Finally, in the *temperature-aware scheduling* problem, the objective is to minimize maximum system temperature at run-time, or the number of thermal peaks, instead of minimizing the energy consumption. The main technique, called *Dynamic Thermal Management (DTM)*, has two main variants: *proactive* techniques ([CWT09]) have the objective of preventing thermal violations while the *reactive* techniques ([WB08]) are invoked as a response to an imminent thermal violation.

3.3.1 Energy harvesting

The energy harvesting problem considers an *energy replenishment function* $P_r(t)$ (modeling solar panels or piezoelectric harvesters) which provides additional time and environment dependent power for the processors and batteries.

Rakhmatov and Vrudhula [RV03] addressed the problem of minimizing the energy consumption, while guaranteeing a common deadline and avoiding battery failure for real-time tasks. Their first algorithm decides how to schedule the tasks by considering the precedence constraints and battery characteristics. Then, when the system restarts after a battery failure, the algorithm is invoked to exploit idle states or modulate the processing speed.

Chetto et al. [CMM11] considered the real-time scheduling problem under the Rate Monotonic policy for systems with renewable energy. They proposed five reactive heuristics, which are executed whenever the battery becomes empty. The first heuristic keeps the processor in sleep state for a predefined fixed interval x, while the second extends the inactive period until the charge level reaches a certain threshold. The third one (called $EDeg$, later labeled as PFP_{st} in [ACM13a]) uses the entire available slack time. The fourth heuristic stops charging when the battery is completely charged. Another fifth algorithm is invoked when the battery energy level drops below a predefined lower bound.

El Ghor et al. [EGCC11] proposed an algorithm under the EDF policy that applies task procrastination to the energy harvesting problem. Specifically, the pending jobs are executed if there is enough energy to complete them, otherwise the available slack time is exploited to charge the battery.

Chandarli et al. [CAM12] considered the problem of formally guaranteeing the real-time and energy feasibility, given a workload and a replenishment function. When a battery failure occurs, their algorithm, called PFP_{ALAP}, exploits all the available slack for recharging. The feasibility check considers the worst busy period assuming an infinite battery capacity. Then, the analysis is extended by noting that, if the total harvested energy during spare time is lower than the battery capacity, then no battery failure happens. However, the case in which the battery capacity is reached, and then some energy is wasted, is not addressed.

Abdeddaïm et al. [ACM13a] proposed an optimal algorithm, called PFP_{asap}. The algorithm puts the processor in sleep state for the shortest time interval that still guarantees to harvest the required energy for executing only the next computational unit of the highest priority task (which is equivalent to the first heuristic in [CMM11] by considering $x = 1$). Despite the algorithm's optimality, the paper shows that PFP_{asap} introduces a significant number of state transitions and preemptions. Moreover, the overhead associated with transitions to/from low-power states is not considered.

Abdeddaïm et al. [ACM13b] further improved PFP_{ASAP} by considering the effect of incoming high priority load on the actual execution of low priority tasks. Basically, the analysis checks whether the actual execution of a task may affect the feasibility of jobs that would arrive later. If this happens, the actual execution is suspended. Although this approach reduces the drawback due to the reactive nature of the algorithm, it heavily relies on the response time analysis at each step.

Chetto and Queudet [CQ13] proved that optimality is not guaranteed for non-idling online algorithms, such as EDF. Moreover, they also shown that an online clairvoyant algorithm is not optimal if, at any instant t, it is not able to foresee what happens until $t + D$, where D is the longest relative deadline.

Kooti et al. [KDMB12] proposed an algorithm which divides the analysis operation

interval (one day) into a set of frames (30 minutes each) without any correlation with the task periods. They assume no more than n out of m consecutive jobs of a task miss their deadlines. At design time, an ILP solver computes for each frame the allocated energy and the number of hard real-time and best-effort jobs. Then, within each frame, the online step executes the hard real-time tasks while the best-effort tasks are executed only if there is enough energy.

Moser et al. [MCT10] defined a 3-layered solution for energy harvesting systems: Application Rate Control, Service Level Allocation, and Real-Time Scheduling. For the first two layers, a long-term analysis is conducted to guarantee an average high performance level. For the real-time scheduling phase, an algorithm called *Lazy Scheduling Algorithm (LSA)* is proposed. Specifically, by using the EDF policy, the algorithm sets the start time of a job equal to its deadline minus the total harvested energy divided by the power consumption. In this way, idle intervals that can be used to recharge the battery are introduced in the schedule. However, the transition overheads associated with low-power states are not considered. The idea is further followed in [MBTB06], which provides a admission condition that considers the energy demand and supply functions.

Chapter 4

Energy-aware scheduling on single-core systems

This chapter introduces the proposed approaches to address the energy-saving issue on single-core systems with real-time constraints.

In Section 4.1, the first contribution [BBMB13a], extending the idea presented in [BBB12], exploits the limited preemptive task model to further reduce the energy consumption, significantly improving the actual state of the art.

The energy-aware real-time co-scheduling of tasks and messages is considered in Section 4.2 where the algorithm DEAS [MBP+11] (Discrete Energy-Aware Scheduling) is presented. Besides the real-time requirements, addition constraints related to the bandwidth management are involved into the analysis.

The last two sections consider the energy-aware issue from a practical point of view. Section 4.3 presents a kernel module [BPMB11] implemented in a Real-Time Operating System (RTOS), showing the actual effectiveness of several energy-aware algorithms. Then, Section 4.4 aims at bridging the gap between theoretical algorithms and actual platforms. More precisely, the analysis [BMB14] simulates several well-known algorithms while considering realistic power models, in order to truly evaluate the effectiveness of the actual beliefs.

4.1 Energy efficiency exploiting the Limited Preemptive model

This work exploits the limited preemptive scheduling model in order to reduce preemption costs and further extend sleep intervals under fixed-priority systems. Moreover, DVFS and DPM techniques are integrated to further reduce energy consumption.

The limited preemptive model has been adopted as it takes advantage of both fully-preemptive and non-preemptive modes, mitigating their drawbacks. As shown in literature [BBY13], limited preemptive scheduling increases the schedulability even when the preemption overhead is neglected. The improvement is even more significant when considering the preemption cost, which generally includes the context switch time for suspending the running task and dispatching the new one, the time taken to flush the pipeline, and the cache-related preemption delay due to cache misses. Moreover, limited preemptive scheduling allows an implicit mutual exclusion management (when

critical sections are encapsulated inside non preemptive regions) and permits reducing the minimum stack memory requirements.

The algorithm consists of an offline DVFS stage which selects the speed that minimizes energy consumption during active intervals while guaranteeing the feasibility of the task set under limited preemptive scheduling. For those architectures taking advantage of speed scaling techniques for reducing energy consumption (DVFS-sensitive architectures), the speed computed by the proposed offline algorithm is shown to be lower than the one selected by existing DVFS algorithms under fully preemptive or non-preemptive models. Conversely, for those architectures in which the use of low-power states is more convenient (DPM-sensitive architectures), the offline stage returns the highest available speed. In the second stage, a DPM technique is applied online to prolong idle intervals as long as possible to take advantage of low-power states. It is shown that such a technique is able to significantly decrease energy consumption with a negligible runtime overhead. This is possible thanks to the offline phase, used to compute the longest delay that can be added after an idle interval to keep the processor in sleep mode, so avoiding complex online computations.

Experimental results illustrate the benefits of the presented techniques under different architectural parameters, including the break-even time and the preemption overhead. With respect to other proposed algorithms [CK06] our method delays task executions rather than job arrivals, allowing a further reduction of the number of preemptions. Since the algorithm is invoked when the processor becomes idle, spare times due to early terminations are automatically reclaimed. No extra hardware is required, except for a timer (active also when the processor is in sleep mode) needed to handle the wake-up events. Concerning complexity, even though the offline analysis is pseudo-polynomial, at runtime the algorithm has a constant complexity, $O(1)$.

The analysis proceeds introducing in Section 4.1.1 the system model. The motivation example in Section 4.1.2 shows the ample room of improvement given by the limited preemptive model. Section 4.1.3 reports the schedulability analysis for the limited preemptive task model that is adopted in this paper. Section 4.1.4 presents the proposed solution and an implementation of the algorithm, while Section 4.1.5 reports the experimental results obtained by exhaustive simulations.

4.1.1 System model

In accordance with the model in Chapter 2, we consider a set Γ of n fixed priority periodic tasks, $\tau_1, \tau_2, \ldots, \tau_n$, executing upon a single processor platform with preemption support. Priorities are assigned according to Rate Monotonic. Without loss of generality, we assume that tasks are indexed in decreasing priority order (i.e., if $0 < i < j \leq n$, then τ_i has higher priority than τ_j).

In addition, each task τ_i consists of a sequence of non-preemptive chunks and can be preempted only at the end of a chunk. For the sake of the analysis, the duration of the longest chunk (at the current speed s) is denoted as $q_i^{max}(s)$, and the one of the last chunk is denoted as $q_i^{last}(s)$. Note that the last non preemptive chunk of a task is crucial for decreasing its response time, because it reduces the interference from higher priority tasks. For this reason, it is convenient to make the last chuck as long as possible. However, q_i^{max}(s) can not be arbitrarily large to limit the blocking time imposed to higher priority tasks. Note that, under such a model, tasks do not need to be independent, but can interact through shared resources, provided that critical sections are entirely contained within a non-preemptive chunk.

The worst-case execution time (WCET) of τ_i is computed as $C_i^{NP}(s) = \alpha_i C_i^{NP} + (1-\alpha_i)C_i^{NP}/s$, where C_i^{NP} denotes the time to execute τ_i in a non-preemptive mode at the maximum speed ($C_i^{NP} = C_i^{NP}(s_m)$) and α_i represents the portion of execution time that does not scale with the speed (e.g. I/O operations). Moreover, the symbol $C_i(s)$ denotes the worst-case execution time of task τ_i in limited preemptive mode, including the preemption overhead. Relative deadlines can be smaller than, equal to, or greater than periods. All parameters are assumed to be in $\mathbb{N}^+$.

4.1.2 Motivational examples

To illustrate the benefit of limited preemption scheduling to save energy, two analysis are here reported. The first one shows the perk which comes from the increased number of feasible task sets in terms of lowest feasible speed, whereas the second, which is extracted from [BBMB13b], highlights the side effect of the preemption overhead on the overall energy consumption.

Impact of the higher number of feasible task sets

Consider a processor with two speeds, $s_1 = 0.5$ and $s_2 = 1$, without low-power states (the processor is always on), executing two tasks, τ_1 and τ_2, with the following parameters: $C_1 = 30$, $T_1 = D_1 = 80$, $C_2 = 25$ and $T_2 = D_2 = 200$ (computation times are referred to speed s_2). Tasks are scheduled using Rate Monotonic and, for the sake of simplicity, preemption costs are considered negligible and $\forall \tau_i : \alpha_i = 0$. The processor utilization factor at speed s_2 is $U = 0.5$ and the task set is feasible under fully-preemptive, non-preemptive, and limited preemptive scheduling. Switching to s_1, however, computation times become $C_1 = 60$ and $C_2 = 50$, making the task set unfeasible under both fully-preemptive and non-preemptive modes. Nevertheless, a feasible schedule can be found under the limited preemptive model by splitting task τ_2 into three chunks of length 10, 20, and 20 units of time, respectively, under speed s_1. The schedules produced by the Rate Monotonic under the three different preemption modes are shown in Figure 4.1.

This example shows that, using the limited preemption model, the processor can run with a speed lower than that allowed by fully-preemptive and non-preemptive models, hence saving more energy.

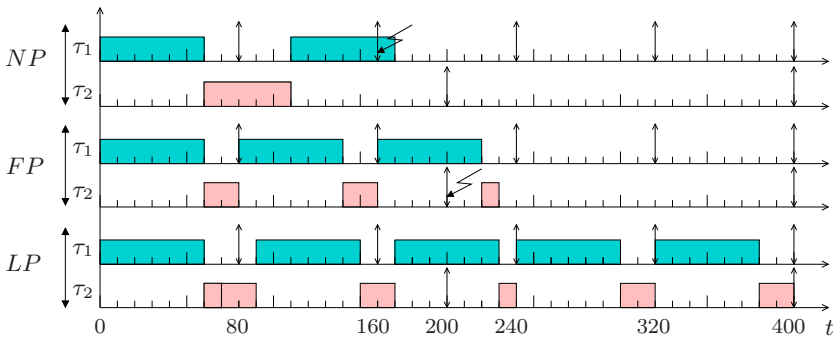


Figure 4.1: Schedules produced by Rate Monotonic at speed $s = 0.5$ under Non-Preemptive (NP), Fully-Preemptive (FP), and Limited Preemptive (LP).

Impact of the preemption overhead

Let us consider synthetic task sets composed of 10 periodic tasks, randomly generated using the UUniFast algorithm [BB05], with computation times uniformly distributed in the interval $[100, 500]$.

Simulations have been carried out assuming a processor with 19 discrete uniformly distributed speeds and power model $P(s) = 0.9s^3 + 0.1$. For the sake of simplicity, a single sleep state σ is considered, with $P_\sigma = 0.0025$, and $E_\sigma = 0.1 \cdot B_\sigma$, where the break-even time $B_\sigma = 500$ (as long as the longest computational time).

All simulations have been performed on the VOSS algorithm, by Chen and Kuo [CK06] as it is the state of the art under fixed priority systems. It is worth noticing that VOSS is an online algorithm with a complexity of $O(n \cdot \log(n))$, which has to be paid at each idle interval.

Figure 4.2 reports the energy improvement of VOSS for different preemption costs ($\xi \in \{0, 5, 10\}$), with respect to a plain DVFS solution with no preemption overhead.

Setting $\xi = 10$, the energy improvement at $U = 0.7$ drops from almost 10% down to less than 4%. This is due to the large preemption number in the considered systems. As noted by Kim et al. [KKM04], executing at the lowest possible speed leads to an increase in the number of preemptions. The reason is that there is less idle time, so that higher priority arrivals are more likely to happen when another task is executing, leading to a preemption. This causes an additional workload that must be executed by the system, increasing the energy consumption.

Increasing the utilization, the performance loss due to preemptions is larger. Comparing the curves with $\xi = 0$ and $\xi = 10$, the energy loss due to the preemption overhead goes from $1 - 2\%$ at utilization 0.4, to $6 - 7\%$ at utilization 0.8. Note that the number of preemptions is almost constant at different utilizations, due to speed scaling. The performance loss at higher utilizations is instead due to the larger impact of the preemption overhead when executing at higher speeds. The additional overhead is executed at energy-expensive speeds, increasing the overall consumption.

This analysis briefly shows that a model which keeps the preemption number under control may reduce the overall dissipation more effectively.

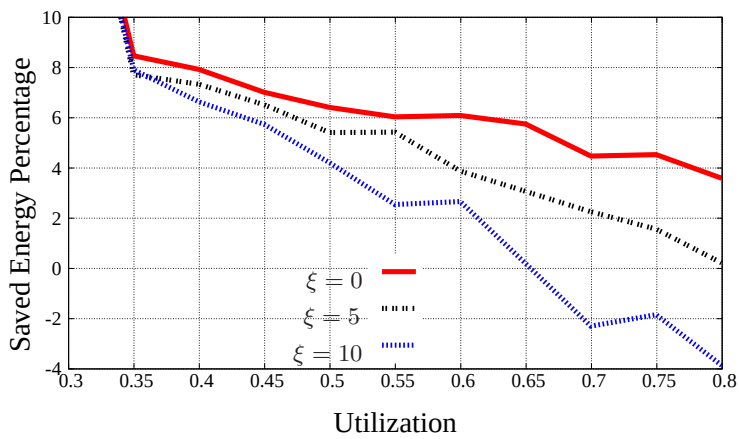


Figure 4.2: Percentage of saved energy with $\xi \in \{0, 5, 10\}$.

4.1.3 Background on limited preemption

The limited preemption scheduling model has been introduced to limit the preemption overhead of fully preemptive schedulers, without incurring in the blocking overhead of non-preemptive solutions. According to this model, each task is divided into a set of non-preemptive regions, so that preemptions can take place only at chunk's boundaries. Two different sub-models have been defined in the literature. In the *Floating* Non-Preemptive Region model [BB10, YBB09], the location of each non-preemptive region is not known a priori, but might vary during task execution. Instead, in the *Fixed* Non-Preemptive Region model [Bur94, BLV09], a set of fixed preemption points is statically defined for each task, so that a task might be preempted only at these well-determined points. This last model has been shown in [BBY11a] to dominate all other techniques, since it is able to schedule a strictly larger number of task sets than the fully preemptive, the non-preemptive and the floating non-preemptive region models. This model, which will be adopted throughout the remainder of the paper, has several benefits, including:

- A bounded preemption number, strictly smaller than the number of chunks;
- A simpler and tighter evaluation of the preemption overhead, as a task can be preempted only at a small number of deterministic locations;
- A smaller preemption cost due to a smaller cache-related preemption delay, by a reduced number of cache misses;
- A smaller worst-case memory stack, which is used to store the contexts of the running and suspended tasks [BBM+10];
- A simplified management of the mutual exclusions, as critical sections contained in a non-preemptive chunk do not need any shared resource protocol.

An exact scheduling analysis for such a model was provided by Bertogna et al. [BBY11a]. Here, the main results are reported, adapting them to the task model considered in this paper (in particular, considering the dependence of the execution parameters to the processing speed).

Note that the number of chunks into which a task is divided depends on the selected speed and is denoted as $p_i(s)$. The duration of the k-th chunk of task τ_i at speed s is denoted as $q_{i,k}(s)$. Hence, the task computation time with limited preemption (at speed s) can be also expressed as the sum of the chunk durations: $C_i(s) = \sum_{k=1}^{p_i(s)} q_{i,k}(s)$. Note that the preemption overhead is included within each chunk length $q_{i,k}(s)$, and the worst-case execution time which considers preemption overhead is

$$C_i(s) = C_i^{NP}(s) + \xi \cdot (p_i(s) - 1). \tag{4.1}$$

The maximum length $q_i^{max}(s)$ of a chunk is computed for a specific speed, as explained in Section 4.1.4. Then, the last chunk is assigned the maximum length, setting it to $q_i^{last}(s) = q_i^{max}(s)$, in order to reduce as much as possible the interference on the considered task. The remaining chunks are all assigned the maximum length $q_i^{max}(s)$, except for the first one, which takes the remaining computation time[1]. Therefore,

$$p_i(s) \stackrel{\text{def}}{=} \left\lceil \frac{C_i^{NP}(s) - q_i^{max}(s)}{q_i^{max}(s) - \xi} \right\rceil + 1. \tag{4.2}$$

[1] Actual chunk sizes might be slightly smaller to accommodate potential critical sections within a non-preemptive region. In this way, there is no need of any shared resource protocols.

$$\begin{cases} q_{i,1}(s) = C_i(s) - (p_i(s) - 1)q_i^{max}(s) \\ q_{i,j}(s) = q_i^{max}(s) \qquad \forall j \in [2, p_i(s)]. \end{cases} \tag{4.3}$$

The maximum blocking time $B_i(s)$ actually experienced by a generic task τ_i can then be computed as

$$B_i(s) = \max_{i<j\leq n} \left\{ q_j^{max}(s) - 1 \right\}. \tag{4.4}$$

Another important parameter useful for checking the feasibility of a task τ_i is the maximum amount of blocking that τ_i can tolerate from lower priority tasks without violating any of its deadlines. Such a time is referred to as *blocking tolerance* and is denoted as $\beta_i(s)$. As shown in [BBY11a], the blocking tolerance of task τ_i can be computed as the minimum blocking tolerance among all its jobs arriving in the largest level-i active period L_i:

$$\beta_i(s) = \min_{k\in[1,K_i]} \beta_{i,k}(s), \tag{4.5}$$

where K_i is the number of jobs in L_i: $K_i = \left\lceil \frac{L_i}{T_i} \right\rceil$, and L_i is the largest level-i active period, computed recursively as

$$\begin{cases} L_i^{(0)}(s) = B_i(s) + C_i(s) \\ L_i^{(l)}(s) = B_i(s) + \sum_{j=1}^{i} \left\lceil \frac{L_i^{(l-1)}(s)}{T_j} \right\rceil C_j(s), \end{cases} \tag{4.6}$$

until $L_i^{(l)}(s) = L_i^{(l-1)}(s)$[2].

Finally, as shown in [BBY11a], the blocking tolerance of job $\tau_{i,k}$ can be computed as

$$\beta_{i,k}(s) = \max_{t\in\Pi_{i,k}} \left\{ t - kC_i(s) + q_i^{last}(s) - W_i(t,s) \right\}, \tag{4.7}$$

where $\Pi_{i,k}(s)$ is the set of arrivals of jobs interfering with $\tau_{i,k}$:

$$\begin{aligned} \Pi_{i,k}(s) \stackrel{\text{def}}{=} & \left[(k-1)T_i, (k-1)T_i + D_i - q_i^{last}(s)\right] \bigcap \\ & \{hT_j - 1, \forall h \in \mathbb{N}, j \leq i\} \bigcup \\ & \left\{(k-1)T_i + D_i - q_i^{last}(s)\right\}, \end{aligned} \tag{4.8}$$

while $W_i(t,s)$ represents the cumulative execution request of all tasks with priority greater than τ_i over any interval $[a,b]$ of length t. It is computed as

$$W_i(t,s) \stackrel{\text{def}}{=} \sum_{j=1}^{i-1} \text{RBF}_j(t,s). \tag{4.9}$$

For any task τ_i and any non-negative number $t \in \mathbb{N}^+$, the **request bound function** $\text{RBF}_i(t,s)$ denotes the maximum sum of the execution requests at the specific speed s that could be generated by jobs of τ_i arriving within a contiguous time-interval $[a,b]$ of length t, considering the preemption costs. It has been shown [LSD89] that the request bound function for a task τ_i is:

$$\text{RBF}_i(t,s) \stackrel{\text{def}}{=} \left(\left\lfloor \frac{t}{T_i} \right\rfloor + 1 \right) C_i(s). \tag{4.10}$$

[2] As shown in [BBY11a], $\beta_{i,1}(s)$ can be used as an upper bound of $B_i(s)$, whenever the latter value is not known.

As a result of the provided analysis, a task set is considered feasibile at speed s under the limited preemptive task model if and only if all the task blocking tolerances are not negative, i.e. $\forall \tau_i \in \Gamma : \beta_i(s) \geq 0$.

4.1.4 Proposed approach

The proposed approach consists of a two-stage algorithm: the first step is executed offline and computes the slowest available speed that guarantees the task set feasibility (also considering preemption costs and energy model). The computed speed is set at the system start and is never changed during execution. The second part of the algorithm is executed at runtime and aims at prolonging the idle intervals as long as possible for exploiting the sleep state, as short idle intervals might not be usable due to the break-even time.

The offline stage of the algorithm is analyzed first, then the online stage is detailed. An example concludes the presentation in order to better show how the algorithm works in practice.

DVFS Algorithm

The offline stage of the method, reported in Algorithm 1, consists of finding the slowest speed that guarantees the task set feasibility, considering the worst-case preemption costs and the particular power function $P(s)$. The speed found by this procedure is never changed at run time.

Algorithm 1 DVFS algorithm

```
1:  function DVFS_ALGORITHM (Γ, ξ)
2:      s* ← compute_critical_speed ()
3:      S' ← {s ∈ S | s ≥ s*}
4:      for each s ∈ S' do
5:          β_min ← ∞
6:          for i ∈ [1, n] do
7:              q_i^max ← min (C_i(s), β_min + 1)
8:              β_i ← compute_task_tolerance (i, s, ξ)
9:              β_min ← min (β_i, β_min)
10:             if β_min < 0 then
11:                 break
12:             end if
13:         end for
14:         if β_min ≥ 0 then
15:             return [s, β_min]
16:         end if
17:     end for
18:     return No speed found
19: end function
```

The algorithm receives as input the task set Γ and the worst-case preemption cost ξ. The first line of the code computes the critical speed s^*, as described in Section 2.1. Then, the speed subset S' is created by sorting the speeds in ascending order and discarding those that are smaller than s^*, since they are not convenient from an energy point of view. Speeds greater than s^* might instead be needed when the task set is

not feasible at s^*. For DPM-sensitive architectures, S' will contain only $s^* = 1.0$, even though lower speeds might be feasible. For each speed in S' (cycle at line 4), all parameters introduced in Section 4.1.3 are computed to find a feasible solution. When a speed that leads to a feasible solution is found, the procedure provides the feasible speed and the corresponding minimum blocking tolerance, denoted as β_{min}.

The cycle at line 6 checks, from the highest to the lowest priority task, whether the considered task can be executed at such a speed without missing deadlines. The feasibility test is done at line 10 by checking whether β_{min} (the minimum blocking tolerance among the tasks analyzed so far) is negative. In fact, $\beta_{min} < 0$ means that a task may be blocked by a lower priority task for a time longer than its slack. The value of β_{min} is first initialized for each speed at line 5 and then updated at line 9 based on the blocking tolerance of the current task (computed at line 8 according to Equation 4.5).

Once all the tasks have been considered at a specific speed, if the minimum blocking tolerance is not negative, then the algorithm completes successfully as the slowest speed that guarantees the feasibility has been found. At this point, the length of each chunk is easily computed by Equation 4.3.

Note that Algorithm 1 increases the complexity of the preemption point placement procedure (which is pseudo-polynomial) by a factor of m (the speed number).

DPM Algorithm

Once the slowest feasible speed has been found for scheduling the task set under limited preemption, a further power reduction can be obtained by exploiting low-power sleep states. Indeed, whenever the processor can be left idle for an amount of time larger than B_σ (the break-even time) without missing any deadline, it is convenient to switch to the sleep state to save more energy. The longer the processor can remain in sleep mode, the smaller the energy consumption.

In this section we present a new online DPM algorithm that exploits limited preemptive scheduling to extend idle intervals as much as possible. The idea behind the proposed algorithm is that, whenever a new job arrives and the processor is idle, this job can be delayed by at least the minimum blocking tolerance β_{min}. In this way, the processor can safely remain in sleep mode for a longer time. An advantage of the presented method is that it does not require any particular online computation of the slack times of the incoming jobs. All meaningful parameters are statically computed before runtime, and no external hardware is needed to compute the length of the idle times and to enforce the sleep states.

The larger blocking tolerances allowed under limited preemptive scheduling [BBY11b] can extend the sleep time to save a significant amount of power, comparable to the power saved by more aggressive online DPM algorithms that require a much higher runtime computational effort. As an example, the algorithm presented in [CK06] requires building the schedule after each idle instant until the earliest deadline, computing the idle time of each job in the considered window, and postponing the arrival of each job by the corresponding idle time. Everything needs to be done online, whereas the algorithm presented here is able to obtain a similar performance without any of the above online operations.

The pseudo code of the DPM procedure is reported in Algorithm 2 and is invoked every time a job ends and the ready queue is empty. The algorithm takes as input the parameters of the task set (Γ) and as a result it prolongs the idle intervals as much

as possible by postponing the execution of the jobs that may arrive in between, still preserving their deadlines.

Algorithm 2 DPM algorithm

1: **function** DPM_ALGORITHM (Γ)
2: $t \leftarrow current\_time\ ()$
3: $t_{act} \leftarrow next\_arrival\ ()$
4: $t_{up} \leftarrow t_{act} + \beta_{min}$
5: **if** $(t_{up} - t) \geq B_\sigma$ **then**
6: $sleep\_for\ (t_{up} - t - B_{\sigma \rightarrow s})$
7: $wake\_up\ ()$
8: **end if**
9: **end function**

After reading the current time t, the algorithm invokes a function that returns the arrival time of the next job. Then, the minimum blocking tolerance computed by Algorithm 1 is added to the arrival time in order to find when the system can be awaken without missing any deadline.

In case the interval spendable in sleep mode is longer than the break-even time B_σ, the routine invoked at line 6 handles all the operations to switch into sleep mode. More precisely, this function sets the job arrival interrupt mask, sets an external timer to send an external waking up interrupt at time $t_{up} - B_{\sigma \rightarrow s}$ and physically switches the system off.

As soon as the external waking up interrupt arrives, the code execution is recovered from line 7, which unmasks job arrival interrupts and handles pending job activations.

The particular way in which the DPM algorithm is implemented allows extending the sleep state and taking advantage of different slack sources:

- unused processor bandwidth related to task set utilizations smaller than one (at the critical speed) and idle times due to the use of a speed higher than the optimal one, since only a discrete set of speeds is available;
- spare capacities associated to early task terminations are automatically reclaimed by the work-conserving nature of the scheduler and collected in the first idle interval.

Note that no external hardware controller is needed to implement the DPM algorithm, except for a simple timer (available in most of the processors) that is programmed by the processor itself before entering the low-power sleep state.

Assuming that task activations are retrieved in constant time, the complexity of the presented DPM algorithm is $O(1)$, making it suitable even for the simplest microprocessors.

Algorithm example

To better explain the proposed algorithm, let us consider a system with four speeds $s_1 = 0.3$, $s_2 = 0.6$, $s_3 = 0.7$ and $s_4 = s_m = 1.0$, power function $P(s) = 0.9s^3 + 0.1$ and break-even time $B_\sigma = 10$. The task set consists of two periodic tasks, τ_1 and τ_2 characterized by $C_1 = 18$, $C_2 = 42$ (at $s = 1.0$), $T_1 = 60$, $T_2 = 150$ and hyperperiod of 300. For the sake of simplicity, preemption costs are considered negligible and $\forall \tau_i : \alpha_i = 0$.

The DVFS algorithm starts computing the critical speed $s^* = 0.4$, which lets us discard s_1 from the analysis. The first speed taken into account is s_2, at which computation times become $C_1(s_2) = 30$ and $C_2(s_2) = 70$, causing a negative blocking tolerance. Thus, the procedure considers next speed s_3, at which computation times become $C_1(s_3) = 26$ and $C_2(s_3) = 60$, the blocking tolerance is $\beta_{min} = 34$, and the task set is feasible. Thus, the algorithm stops. According to this configuration, τ_1 runs in a non-preemptive way for all its execution and τ_2 is split into two chunks of 26 and 34 units of time, respectively.

The task set execution at the slowest feasible speed without using the DPM algorithm is reported in Figure 4.3. Many idle intervals shorter than B_σ are present in the schedule, leading to a waste of energy.

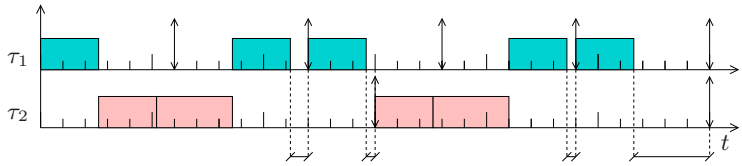


Figure 4.3: Task execution at speed s_3 without using the DPM algorithm on a DVFS-sensitive architecture.

The advantage of introducing the DPM algorithm is shown in Figure 4.4. The algorithm is invoked for the first time at the end of the second job of τ_1 and all the small idle times are collected into a single longer interval, postponing the execution of the third job of τ_1 and the second job of τ_2.

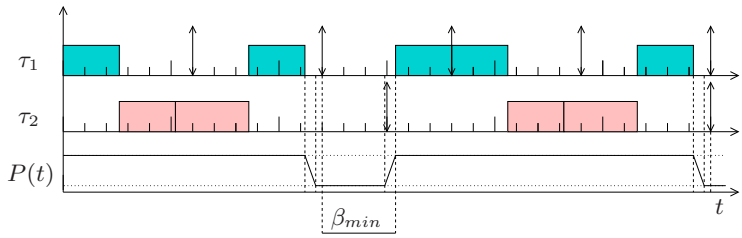


Figure 4.4: Task execution at speed s_3 using the DPM algorithm on a DVFS-sensitive architecture.

A new instance of the DPM algorithm is launched before the end of the hyperperiod. Since β_{min} is longer than the break-even time B_σ, the processor switches to the sleep state.

For DPM-sensitive architectures, the only speed taken into account is the maximum one ($s = s^* = 1.0$) which leads to a feasible schedule. The two tasks contain only a single chunk each, meaning that they are executed in a non-preemptive way, with $\beta_{min} = 42$. As shown in Figure 4.5, scheduling the task set without using the DPM algorithm generates several idle intervals, which are compacted when the DPM algorithm is enabled (as depicted in Figure 4.6).

4.1.5 Experimental results

The synthetic task sets used in the tests are composed of 10 periodic tasks randomly generated using the UUniFast algorithm [BB05], where the total utilization U is varied in a given range and each computation time $C_i^{NP}(s_m)$ is uniformly distributed in

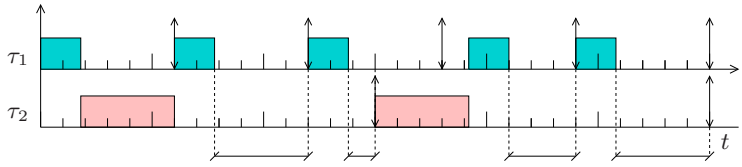


Figure 4.5: Task execution at speed s_m without using the DPM algorithm on a DPM-sensitive architecture.

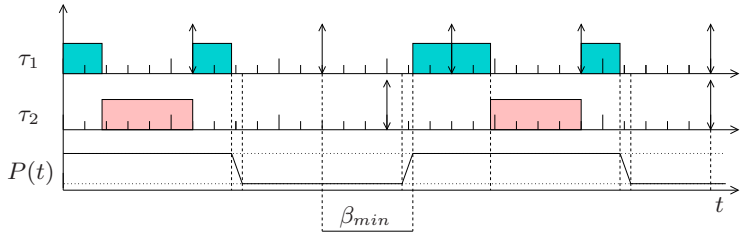


Figure 4.6: Task execution at speed s_m using the DPM algorithm on a DPM-sensitive architecture.

[100,500]. For the sake of simplicity, $\alpha_i = 0.2$ is set for each task. In these tests, relative deadlines are set equal to periods, which are derived once computation times and task utilizations have been generated. Tasks are scheduled under fixed priorities assigned with the Rate Monotonic algorithm and each simulation run is performed until the hyperperiod.

A processor with 19 discrete speeds has been considered, varying in the range of $[0.1, 1]$ with step 0.05.

Two power models have been considered in the experiments: $P^{(1)}(s) = 0.9s^3 + 0.1$ and $P^{(2)}(s) = 0.278s + 0.722$. The first one is often used in literature to model DVFS-sensitive architectures and is characterized by a critical speed $s^* = 0.4$. The second one represents the power consumption of a NXP LPC1768 (Arm Cortex M3), modeling a DPM-sensitive architecture with $s^* = s_m$ (making speed scaling not energy-convenient). The power consumed in the sleep state and the energy required for a complete state transition (from active to sleep and then back to active) are $P_\sigma^{(1)} = 0.05$, $P_\sigma^{(2)} = 0.4$, $E_\sigma^{(1)} = 0.051 \cdot B_\sigma$ and $E_\sigma^{(2)} = 0.45 \cdot B_\sigma$.

The experiments are divided in two parts: the first set shows the performance of the two phases of the proposed method under several scenarios, whereas the second set aims at comparing the proposed approach with the VOSS algorithm presented by Chen and Kuo [CK06].

Performance of the proposed approach

The first experiment aims at testing the impact of the DVFS algorithm under different scheduling approaches. In particular, the average lowest speed achieved by the limited preemptive scheduler is compared with the ones obtained by the fully preemptive and non-preemptive schedulers, for different task set utilizations and under different preemption costs. Only two preemption costs are shown: $\xi = 0$ and $\xi = 10$. The second value represents a system with a preemption cost equal to one tenth and one fiftieth of the shortest and longest possible task execution (as $C_i^{NP} \in [100, 500]$), respectively. For each utilization, the average lowest speed was computed over 700 feasible task sets.

Figure 4.7 reports the resulting average lowest feasible speed as a function of the utilization factor. Note that, although the non-preemptive scheduler does not suffer from preemption overhead, it always requires the highest speed. Moreover, the speed found under fully-preemptive scheduling is always higher than or equal to the one under limited preemption, even with zero preemption cost. Such a benefit comes from the capability of limited preemptive schedulers of increasing the number of feasible task sets. Note that the introduction of a preemption cost $\xi = 10$ leads to a significant speed increase for the fully preemptive model, while its impact on the limited preemptive scheduler is almost negligible. Also observe that, for utilizations higher that $U = 0.9$, only the limited preemptive approach can achieve a feasible schedule for a significant number of generated task sets (at least one out of two). Finally, the plateau observed for $U > 0.92$ under the limited preemptive model is due to the fact that almost all the feasible task sets require a speed of one, while most generated task sets are discarded since their feasibility could be guaranteed only with a speed greater than the maximum one. This leads to an average value near to $s = 1$ for several utilization points.

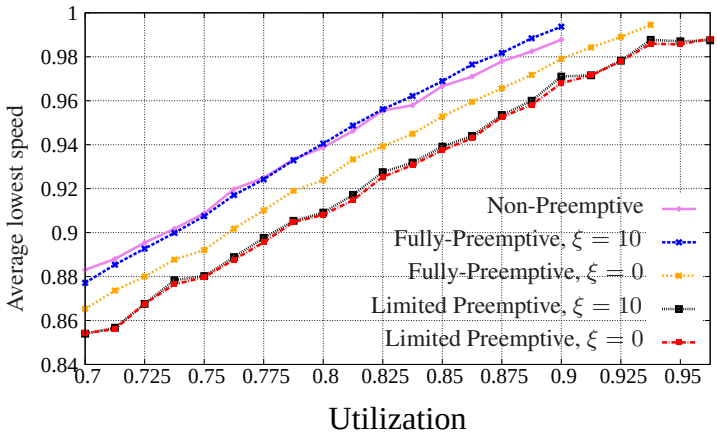


Figure 4.7: Average lowest speed for different schedulers and preemption costs.

The second experiment aims at evaluating the energy saved by the two-step algorithm with respect to the case in which the processor is kept always active at $s = 1$. The power model considered in this test is $P^{(1)}(s)$, representing a DVFS-sensitive architecture, since in DPM-sensitive architectures the DVFS stage would not introduce any contribution, always returning the maximum speed.

Results are reported in Figure 4.8 for $\xi = 10$ and two values of B_σ (0 and 500 time units). The *Pure DVFS* curve represents the energy consumption obtained by using only the DVFS step. The introduction of the DPM stage allows a further energy reduction, which is about 8%, in the best case. Note that high break-even times push the curve closer to the one of *pure DVFS*, as the algorithm is not able to switch the processor off during all idle intervals. For the sake of completeness, the figure also shows the behavior of the algorithm under a *pure DPM* approach, where only the second stage is considered at the maximum speed. Since the power model is intrinsically speed scaling-convenient, DPM consumes more than the others, even assuming a null break-even time.

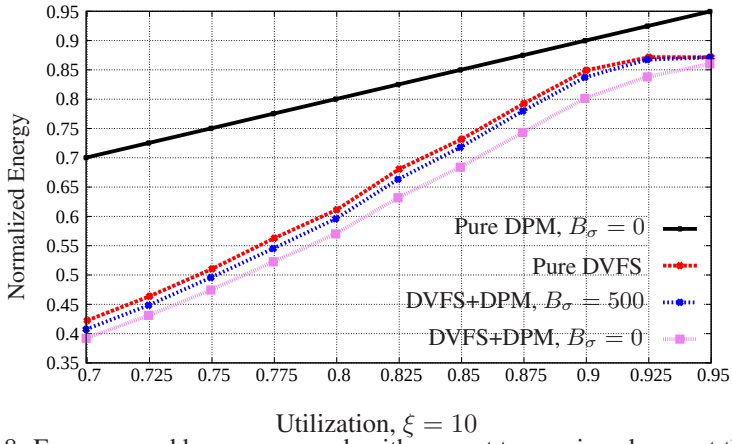


Figure 4.8: Energy saved by our approach with respect to running always at the maximum speed $s = 1$.

Comparison with VOSS

A second set of experiments was carried out to compare the proposed approach against existing solutions available in the literature. Among the existing works that deal with fixed priority systems and that do not require additional hardware controllers, the algorithm that showed the best performance in terms of energy saving is VOSS, presented in [CK06]. We therefore decided to compare our algorithm only against VOSS, as the improvements over other existing solutions would be even greater. However, it is worth noticing that VOSS is an online algorithm with a complexity of $O(n \cdot \log(n))$ to be paid at each idle interval, whereas our online algorithm is $O(1)$. Finally, an improved version of VOSS is adopted that computes the feasible speed using the tighter Response Time Analysis [ADM11] (including preemption costs) instead of Liu and Layland's bound [LL73].

Figure 4.9 shows the energy percentage saved by the proposed approach with respect to VOSS as a function of the total utilization, using the $P^{(1)}(s)$ model and for different preemption costs ($\xi \in \{0, 10\}$) and break-even times ($B_\sigma \in \{250, 500, 750\}$).

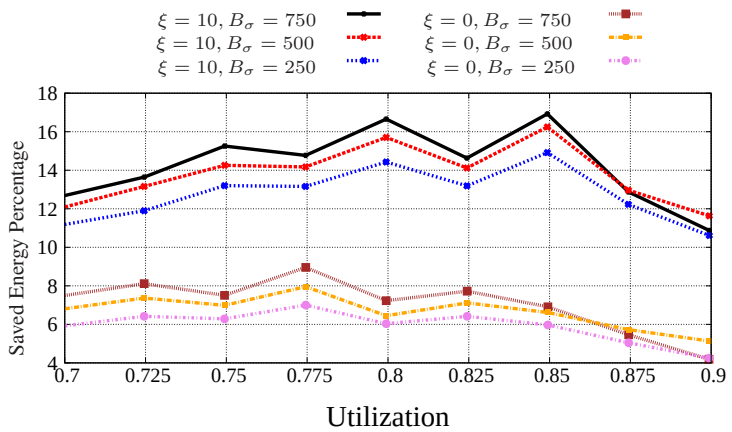


Figure 4.9: Energy percentage saved by the proposed approach with respect to VOSS, for a DVFS-sensitive architecture.

As shown in the graph, the energy improvement of our approach (considering both DVFS and DPM algorithms), when the preemption cost is neglected, is around 7%. Using more realistic values for the preemption overhead (one tenth of the shortest task), the improvement increases up to 16%. This is a consequence of the reduced number of preemptions of our approach, which makes it even more competitive for higher preemption costs. The impact of the break-even time is smaller, although the longer B_σ, the higher the gain. The reason is that our method exploits longer blocking tolerances than VOSS, overcoming the break-even time limit more easily. When B_σ is either too long or too short, the performance of the DPM algorithms are equivalent. At high utilizations, the margin of improvement is barely usable, due to the reduced number of feasible task sets and the short idle time, so the behavior of the two algorithms is similar. Note that, for $\xi = 10$, the analysis ends at $U = 0.9$ as there are no feasible task sets under fully preemptive scheduling.

The two algorithms have also been compared under the second power model $P^{(2)}$, typical of DPM-sensitive architectures, under which the speed returned by our offline DVFS algorithm is always equal to the maximum available (never exploiting the speed scaling feature), and the whole energy improvement is due to the DPM algorithm.

Figure 4.10 reports the improvements achieved by the proposed approach with respect to VOSS as a function of the total utilization, with $B_\sigma \in \{250, 500, 750\}$ and $\xi \in \{0, 10\}$.

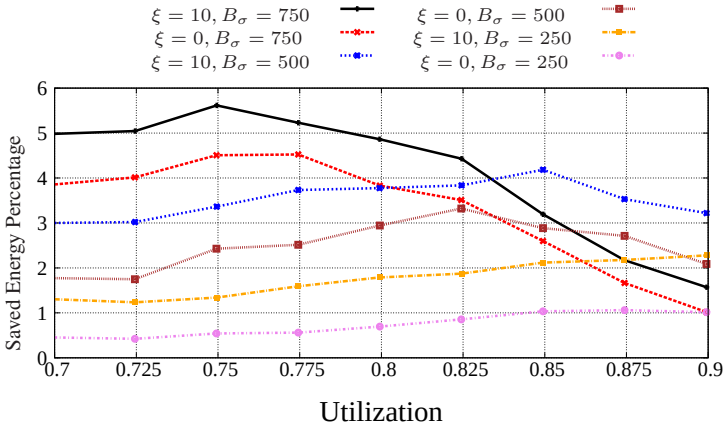


Figure 4.10: Energy percentage saved by the proposed approach with respect to VOSS, for a DPM-sensitive architecture.

Note that the proposed approach always outperforms VOSS. Also, increasing the preemption cost leads to a higher improvement, as the context switch overhead of the fully-preemptive model becomes higher. For the same value of ξ, the higher B_σ, the higher the energy gain, as the blocking tolerance used to delay task execution under the limited preemption model is longer than that in fully preemptive mode. This happens up to a certain utilization, after which no algorithm is able to postpone task execution for a time longer than B_σ. When $U = 0.75$, our algorithm consumes almost 6% less than VOSS. The improvement would be even bigger when accounting for the additional number of online operations required by VOSS at every idle time ($O(n \log(n))$ instead of $O(1)$).

An additional experiment showed that by increasing α_i, the DVFS algorithm is able to reach slower speeds (as a more significant fraction of the task code is not affected

by speed), so leading to higher savings than those reported in Figure 4.9. For DMP-sensitive platforms, the results shown in Figure 4.10 are not influenced by α_i as speeds lower than s_m are not taken into account.

4.2 Energy-aware co-scheduling of tasks and messages

This paper addressed the problem of reducing the energy consumption in embedded nodes with time and communication constraints. The proposed solution, referred to as DEAS (Discrete Energy-Aware Scheduling), reduces the energy dissipation by exploiting both DPM and DVFS techniques, balancing them according to the specific architecture characteristics, actual workload, and bandwidth allocation.

In distributed systems, processor and network bandwidth must be taken into account to guarantee performance requirements. In particular, in wireless distributed embedded systems, energy consumption and quality of service represent two crucial design objectives. Although a lot of research has been done to reduce power consumption while guaranteeing real-time and bandwidth requirements, most papers focus either on task scheduling or network communication, separately. However, co-scheduling of task and messages would explore more degrees of freedom and could lead to higher energy saving.

To simplify the analysis, in this paper the communication bandwidth is assumed to be statically allocated according to a TDMA scheme and the platform is forced to be active during such intervals.

This work improves the approach previously proposed by Santinelli et al. [SMP+10] in several directions. First of all, tasks are assumed to be sporadic, rather than periodic. Second, the DVFS model is more realistic, since the CPU frequency is assumed to vary within a set of discrete values, rather than in a continuous range. Third, the algorithm is completely redesigned to consider the effects of the execution platform, while containing the overall computational complexity. Finally, the experimental section includes new test cases and simulation results.

Section 4.2.1 presents the differences in the system model. A background on schedulability analysis is reported in Section 4.2.2, while Section 4.2.3 illustrates the proposed algorithm. Section 4.2.4 ends the analysis with the simulation results carried out to evaluate the performance of the approach.

4.2.1 System model

We consider a distributed real-time embedded system composed by autonomous nodes interconnected through a shared media (e.g., wireless communication). A generic node executes a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of sporadic tasks scheduled by Earliest Deadline First (EDF) [LL73].

In addition to the parameters in Chapter 2, each task τ_i produces a message m_i characterized by a payload M_i and a deadline L_i relative to the task activation, such that $L_i > D_i$. The absolute deadline of the message produced by job $\tau_{i,j}$ is denoted by $l_{i,j} = r_{i,j} + L_i$.

The analysis we are proposing assumes a given bandwidth allocation specified according to a Time Division Multiple Access (TDMA) scheme, modeled as a set of disjointed slots $B = \{slot_1, \ldots, slot_r\}$, where each slot $slot_k$ is described by a start time b_k^s and an end time b_k^e. Such slots are externally assigned by a network coor-

dinator that guarantees that messages are transmitted/received within their deadlines. However, the design of the coordinator is out of the scope of this work.

To simplify the power management algorithm, task scheduling is decoupled from message communication by making the following assumptions:

1. The messages generated by a task are moved into a shared communication buffer and transferred to the transceiver internal buffer whenever the bandwidth is available. The transfer time is considered to be negligible;

2. To allow messages transfer between buffers, the processor must be active during communication slots, whereas CPU activity is not required outside such intervals;

3. The bandwidth slots allocated by the external network coordinator are such that any scheduling algorithm that guarantees the timing constraints of the task set also meets the messages deadlines.

A graphic explanation about all the defined parameters is reported in Figure 4.11 for both the computational and the communication components.

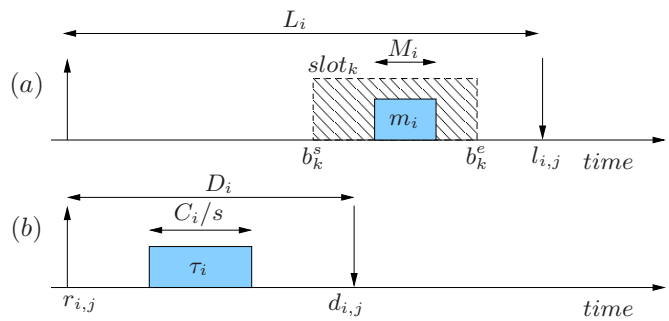


Figure 4.11: Timing parameters for a task (b) and its generated message (a).

Each node consists of a CPU (processing element) and a Transceiver (transmitting and receiving element). The CPU supports speed scaling and can be in *active*, *standby* (σ_1) or *sleep* (σ_2) state. In active mode, the power consumption is computed according to Equation 2.3 and the available speeds are bounded in number. The break-even times of the low-power states are negligible and non-negligible, respectively.

In this work, we assume the usage of a standalone transceiver with minimal functionality, with only two states: ON and OFF. As the transceiver implements only the physical layer of the communication stack, the computational unit has to manage the remaining layers. This choice forces the device to be OFF if the CPU is in $Standby$ or $Sleep$ states. As our analysis needs only these two fundamental working states, it is applicable to all existing communication devices. In the following, the transceiver power consumption in the ON and OFF states are denoted as P_{on} and P_{off}, respectively.

	Radio On	Radio Off
CPU Sleep	Not allowed	$P_{\sigma_2} + P_{off}$
CPU Standby	Not allowed	$P_{\sigma_1} + P_{off}$
CPU On	$P(s) + P_{on}$	$P(s) + P_{off}$

Table 4.1: Allowed power modes.

Table 4.1 summarizes all the allowed modes with their total power consumption.

4.2.2 Background on schedulability analysis

Schedulability analysis of the task set is performed using the demand bound function dbf [BMR90] to describe the application computational requirements, and the supply bound function sbf [SL08] to characterize the service provided by the processor.

In the case of EDF, Baruah [Bar03] showed that the dbf of a sporadic task set, in a generic interval, can be computed as

$$dbf(t_1, t_2) = \sum_{i \in \Gamma} \left(\left\lfloor \frac{t_2 + T_i - D_i}{T_i} \right\rfloor - \left\lceil \frac{t_1}{T_i} \right\rceil \right) C_i. \tag{4.11}$$

In the processing model considered in this paper, the processor can run at a speed s whenever the processor is in active mode, while it is steady if the processor is in standby or sleep mode. Hence, the sbf linearly increases with slope s when the CPU is active, and remains constant in standby and sleep states. For a given power state, the sbf in an interval $[t_1, t_2)$ is computed as

$$sbf(t_1, t_2, f) = (t_2 - t_1)s. \tag{4.12}$$

The real-time constraints of a scheduling component are met if and only if, in any interval of time, the resource demand of the component never exceeds the resource supply curve. That is, if and only if

$$\forall t_1, t_2 \in \Re^+, \quad t_2 > t_1, \quad dbf(t_1, t_2) \leq sbf(t_1, t_2, s). \tag{4.13}$$

Ripoll [RCM96] gave an upper bound L_a on the time interval in which Equation (4.13) must be checked, under the assumption that $D_i \leq T_i$ for each task τ_i. This work extends such an analysis taking into account the system speed s as

$$L_a(f) = \quad max \left\{ D_1, \ldots, D_n, \frac{\sum_i (T_i - D_i) U_i}{s - U} \right\}.$$

Spuri [Spu96] also defined the *Busy Period* (BP) as the longest interval of time where the processor is never idle, computed assuming synchronous and offset-free activations of tasks. This work extends the concept of BP considering a system working at speed s. Hence, $BP(s)$ can be used as another upper bound for the schedulability test. Therefore, if $U < s$, to guarantee the feasibility it is sufficient to check Equation (4.13) just at the task deadlines in $[t_1, t_1 + L^*(s))$, where $L^*(s)$ is defined as

$$L^*(s) = min\{L_a(s), BP(s)\}. \tag{4.14}$$

To speed up the schedulability test, the QPA algorithm proposed by Zhang and Burns [ZB09] could be used in our approach.

4.2.3 Proposed approach

The proposed approach mixes at runtime DVFS and DPM techniques to reduce energy consumption while meeting all task deadlines. The combination of DVFS and DPM is done by forcing a CPU sleep interval followed by an active interval executed at a fixed speed. Such a speed is selected to minimize the energy (per unit of computation) between the current and the next invocation of the analysis.

The j-th instance of the analysis is performed either at the end of an active interval or at the end of a communication slot. The former instant represents the beginning of

an idle period that can be prolonged further by the analysis, while the latter is selected to exploit the slack, if any, collected during the forced activity inside the slot.

The following terminology is used to identify particular timing instants.

- t denotes the current time;
- $next\_act(t)$ denotes the next activation time after t;
- t_{a_j} denotes the time at which the j-th instance of the analysis is invoked. If there are pending jobs at time t, t_{a_j} is set at the current time t, otherwise t_{a_j} is postponed at $next\_act(t)$:

$$t_{a_j} = \begin{cases} next\_act(t), & \text{if no pending jobs at } t; \\ t, & \text{otherwise.} \end{cases} \tag{4.15}$$

  The index j referring to a particular instance will be omitted whenever not necessary;
- t_{w_i} denotes the latest time after t_{a_j} at which the processor can return active with speed s_i and still guarantee the schedulability of the task set;
- t_{idle_i} denotes the first idle time after t_{w_i} assuming the processor is executing at speed s_i;
- t_{e_i} denotes the effective time at which the processor can become idle considering the activity constraint inside bandwidth slots. Hence, if t_{idle_i} falls before b_k^s, t_{e_i} is set at t_{idle_i}; otherwise t_{e_i} is forced to occur at the end of the bandwidth slot, that is, $t_{e_i} = b_k^e$.

When the analysis is invoked at time t_a, the following actions are performed:

1. For each speed s_i, the analysis derives the longest inactive interval δ_i exploitable in sleep state from t_a, such that the task set is still feasible when the CPU is turned active at $t_a + \delta_i$. A negative value of δ_i implies that the task set can not be schedulable at that speed. δ_i is determined as the minimum among the inactive intervals computed for each deadline, that is

$$\delta_i(t) = \min_{d_j \in [t, t+L^*(f_i))} \left\{ d_j - \frac{dbf(t, d_j)}{s_i} - t \right\}; \tag{4.16}$$

2. To ensure that the CPU is active during the assigned bandwidth slots, the wake up time t_{w_i} is set equal to the minimum between $t_a + \delta_i$ and the beginning of the next slot b_k^s

$$t_{w_i} = min\{t_a + \delta_i(t_a), b_k^s\}; \tag{4.17}$$

3. For each speed s_i, the analysis also computes the next idle time t_{idle_i} from t_{w_i} assuming worst-case executions. In particular, t_{idle_i} is computed as the minimum value satisfying the following recurrent relation:

$$\begin{aligned} t_{idle_i}^{s+1}(t_a) = & \sum_{\tau_j \text{ active}} \frac{c_j(t_a)}{s_i} + \\ & \sum_{j \in \Gamma} \left(\left\lfloor \frac{t_{idle_i}^s}{T_j} \right\rfloor - \left\lfloor \frac{t_a}{T_j} \right\rfloor \right) \frac{C_j}{s_i}. \end{aligned} \tag{4.18}$$

initialized with value $t^0_{idle_i}(t_a) = t_a + \sum_{\tau_j \text{ active}} \frac{c_j(t_a)}{s_i}$.

The analysis then computes the effective idle time t_{e_i} taking into account the bandwidth constraint.

$$t_{e_i} = \begin{cases} t_{idle_i}, & t_{idle_i} < b^s_k; \\ b^e_k, & \text{otherwise.} \end{cases}$$

4. Under a speed s_i, the energy consumption E_i in the interval $[t_a, t_{e_i}]$ is computed as the sum of the energy spent in sleep mode in $[t_a, t_{w_i}]$ and in active mode in $[t_{w_i}, t_{e_i}]$, that is

$$E_i(t_a, t_{w_i}, t_{e_i}) = (t_{w_i} - t_a)P_\sigma + (t_{e_i} - t_{w_i})P(s_i). \tag{4.19}$$

   Since each speed s_i causes a different amount of computation in the interval $[t_a, t_{e_i}]$, denoted as $W_i(t_a, t_{e_i})$, the normalized parameter *Energy Per Cycle* (EPC_i) is introduced, representing the energy cost per instruction cycle. It is computed as

$$EPC_i(t) = \frac{E_i(t_a, t_{w_i}, t_{e_i})}{W_i(t_a, t_{e_i})}; \tag{4.20}$$

   A detailed analysis about the computation of W_i is carried out in the next subsection.

5. Among the possible speeds that guarantee feasibility, the approach selects s^* featuring the minimum EPC_i. t^*_w and t^*_e denote the wake up time and the effective idle time resulting from the selected speed s^*, respectively;

6. If the interval $[t, t^*_w)$ is shorter than $t_{a\sigma} + t_{\sigma a}$, it is not possible to adopt the sleep state to wake up within t^*_w, so the standby state is chosen; otherwise, the sleep state is selected;

7. The instant of the next occurrence of the analysis $t_{a_{j+1}}$ is set equal to t^*_e; however, if the next idle time is advanced due to early completions, the analysis is triggered as soon as the idle occurs and $t_{a_{j+1}}$ is updated accordingly.

Figure 4.12 illustrates an example that clarifies the steps of the proposed approach. In the example, the CPU supports three different speeds sorted in ascending order: s_1, s_2 and s_3.

In the example, speed s_1 leads to an unfeasible schedule, since δ_1 is negative, whereas s_2 and s_3 produce feasible solutions, since both δ_2 and δ_3 are positive.

Notice that, when considering speed s_2, $t_a + \delta_2$ falls before b^s_k, hence $t_{w_2} = t_a + \delta_2$, whereas for s_3, t_{w_3} is set equal to the beginning of the slot b^s_k, as $t_a + \delta_3 \geq b^s_k$.

For both speeds s_2 and s_3, t_{e_2} and t_{e_3} takes the value of b^e_k, as both t_{idle_2} and t_{idle_3} occur after b^s_k.

To choose between the two feasible speeds (s_2 and s_3), the normalized energy consumption is computed. Such a value is intrinsically derived from the platform power model.

Finally, the algorithm which implements the proposed approach is reported in Algorithm 3.

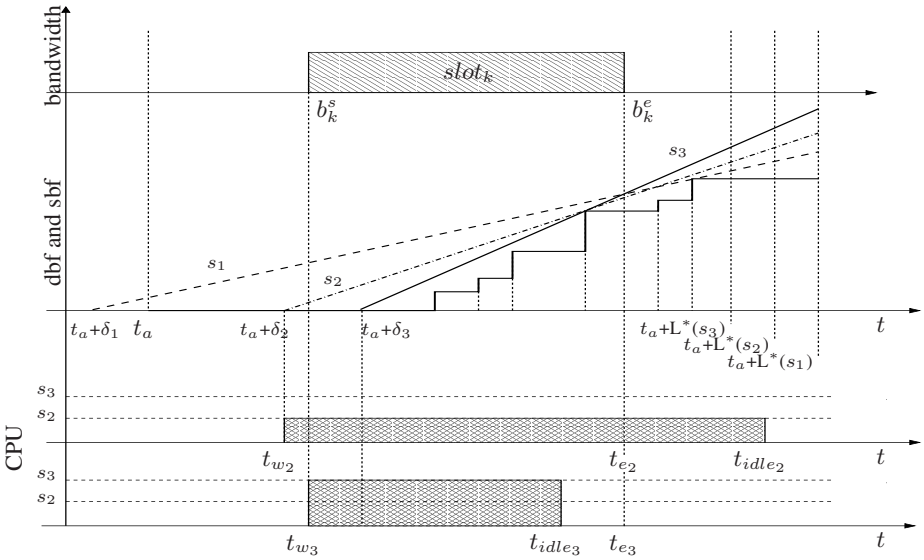


Figure 4.12: Example of the analysis' behaviour.

Algorithm 3 Discrete Energy Aware Scheduling - DEAS

```
1:  function DEAS(t)                                   ▷ ∀k, t ∉ [b_k^s, b_k^e)
2:      compute t_a according to Equation (4.15);
3:      for all s_i do
4:          compute δ_i(t_a) as in Equation (4.16);
5:          if δ_i(t_a) < 0 then
6:              set s_i not feasible and continue;
7:          end if
8:          compute t_{w_i} according to Equation (4.17);
9:          compute t_{e_i}, W_i;
10:         compute EPC_i according to Equation (4.20);
11:     end for
12:     compute s* feasible that minimizes EPC_i;
13:     set wake up time at t_w^*;
14:     set CPU speed to s*;
15:     if t_w^* − t ≥ t_aσ + t_σa then
16:         put the processor in sleep state;
17:     else
18:         put the processor in standby state;
19:     end if
20: end function
```

Workload computation

The procedure $compute\_te\_W$, which computes the effective idle time t_{e_i} and the effective workload W_i is formally defined in Algorithm 4. The algorithm, based on the current workload, computes next idle times t_{idle_i} till t_{e_i} is found, taking into account bandwidth constraints. However, note that the procedure output is composed by t_{e_i} and W_i only. To reduce the DEAS algorithm complexity, such computations are integrated

into a single routine.

Algorithm 4 Procedure to compute t_{e_i} and W_i

```
1:  function compute_te_W(s_i, t_{w_i})
2:      W_i = 0; t_start = t_{w_i};
3:      loop
4:          t^0_{idle_i} = t_start + Σ_{τ_j} active c_j(t_start)/s_i;
5:          do
6:              compute t^s_{idle_i} according to Equation (4.18);
7:              if t^{s-1}_{idle_i} < b^e_k ≤ t^s_{idle_i} then
8:                  t_{e_i} = b^e_k;
9:                  W_i += (t_{e_i} − t_start) s_i;
10:                 return;
11:             end if
12:         while t^{s-1}_{idle_i} ≠ t^s_{idle_i};
13:         W_i += (t^s_{idle_i} − t_start) s_i;
14:         if t^s_{idle_i} ∉ [b^s_k, b^e_k] then
15:             t_{e_i} = t^s_{idle_i};
16:             return;
17:         else
18:             if next_act(t^s_{idle_i}) ≥ b^e_k then
19:                 t_{e_i} = b^e_k;
20:                 return;
21:             end if
22:             t_start = next_act(t^s_{idle_i});
23:         end if
24:     end loop
25:     return (t_{e_i}, W_i)
26: end function
```

Figure 4.13 shows the effective workload of three key scenarios. For each case, the effective workload is represented by the sum of the slashed areas. Such a value is expressed as number of machine cycles, so the working speed must be considered.

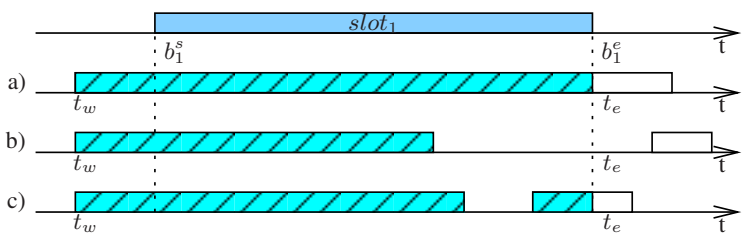


Figure 4.13: Examples of effective workload.

First, to determine t_{idle_i} the iterative approach is initialized as previously described. Whenever $t^s_{idle_i}$, at a generic step s, crosses the end of the current bandwidth slot $t^{s-1}_{idle_i} < b^e_k \leq t^s_{idle_i}$ (cases a and c), the procedure stops setting $t_{e_i} = b^e_k$ and accounting in W_i the workload between the beginning of the current Busy Period and the end of the bandwidth slot b^e_k. Note that, if the recurrent relation converges ($t^{s-1}_{idle_i} = t^s_{idle_i}$) outside the bandwidth slot, no slot occurs during the analyzed Busy Period, hence

$t_{e_i} = t^s_{idle_i}$ and W_i is increased by $(t^s_{idle_i} - t_{w_i})s_i$. If $t^s_{idle_i}$ converges inside the bandwidth slot (cases b and c), W_i is incremented by $(t^s_{idle_i} - t_{w_i})s_i$. Then, the routine checks whether a new task activation occurs after the end of the current bandwidth slot, i.e. $next\_act(t^s_{idle_i}) \geq b^e_k$. In such a case (case b), the effective idle time t_{e_i} is set to b^e_k and W_i is increased by $(t^s_{idle_i} - t_{w_i})s_i$; otherwise (case c), the contribution of the next Busy Period must be taken into account considering $next\_act(t^s_{idle_i})$ as a starting instant.

Complexity

Equation (4.15), executed at line 2, has the complexity of an extraction from an ordered list of task activation times; that is, $O(1)$. On the other hand, the insertion complexity is $O(log_2(n))$, where n is the number of tasks. Given n and the maximum number of deadlines a single task can produce in the analysis interval, p, the maximum number of analysis points of the dbf is np. The upper bound of p is computed as the number of occurrences of the task with the shortest period in the analysis interval: $\left\lceil \frac{max_i L^*(f_i)}{min_i\{T_i\}} \right\rceil$. Supposing to arrange the active deadlines in a sorted list, with complexity $O(log_2(n))$ (as the active deadlines are always n) to keep the ordering, the computation of dbf, executed every time the algorithm is invoked, has a total complexity of $O(log_2(n)np)$.

The computation of the δ_i, at line 4, involves a complexity $O(np)$. The computation performed at line 9 has a complexity of $O(nq)$, where q is defined as the maximum number of activations a task can generate in $[t_{a_j}, t_{a_j} + max_i L^*(f_i) + max_k\{b^e_k - b^s_k\}]$. The reason is that the algorithm analyzes all the activations, computing the actual workload and any idle gap. The q upper bound is computed as $\left\lceil \frac{max_i L^*(f_i) + max_k\{b^e_k - b^s_k\}}{min_i\{T_i\}} \right\rceil$. The computation of the EPC_i has complexity $O(1)$. Hence, the for loop, executed at line 3, has a complexity of $O(nF(p+q))$, where F is the total number of available speeds.

Globally, the proposed algorithm has a complexity of $O((log_2(n) + F)qn)$, being $q \geq p$.

Example

The behavior of the DEAS algorithm is now illustrated using the example in Figure 4.14. The assigned bandwidth is composed by one slot in the interval $[b^s_1, b^e_1]$ equal to [12,15]. The CPU allows 2 speeds equal to $s_1 = 0.5$ and $s_2 = 1.0$, and schedules a task set of 2 synchronous implicit periodic tasks with periods $T_1 = D_1 = 5$ and $T_2 = D_2 = 7$, and worst-case execution cycles $C_1 = 1$ e $C_2 = 1$. For the task set under analysis, we have $L^*(s_1) = 4$ and $L^*(s_2) = 2$. The result of the off-line computation for L_{max} is 7. The power consumptions in the active state are $P(s_1) = 3$ and $P(s_2) = 6$, while $P_{\sigma_2} = 1$ and B_{σ_2} is considered negligible for the sake of simplicity.

The algorithm has its first invocation at $t_{a_1} = 0$ because two jobs are already pending. Both speeds guarantee the task set feasibility with wake up times $t_{w_1} = 3$ and $t_{w_2} = 4$, respectively.

Executing at speed s_1, the first idle time t_{idle_1} occurs at $t = 13$ because, from time $t_{w_1} = 3$, the busy period consists of three instances of τ_1 and two instances of τ_2, for a total execution of 10 units of time. Due to the bandwidth activity constraint, t_{e_1} is set to 15. Instead, running at speed s_2, the next idle time t_{idle_2}, from time $t_{w_2} = 4$, occurs at time $t = 8$ and, since it falls before b^s_1, we have $t_{e_2} = 8$. Once the interval

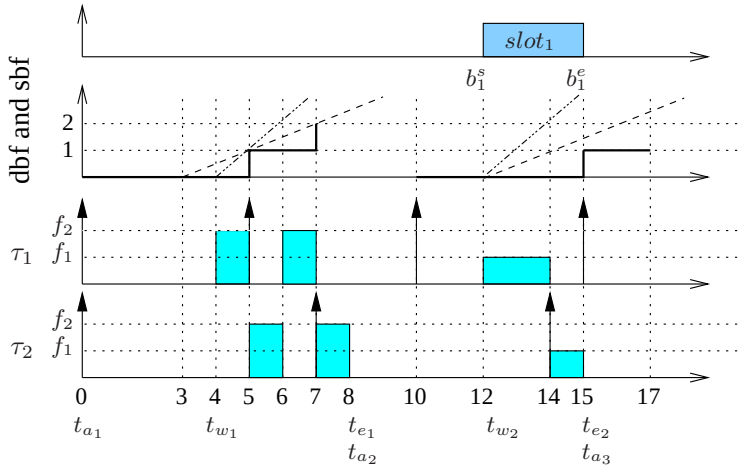


Figure 4.14: DEAS example.

$[t_{w_i}, t_{e_i}]$ is determined, the algorithm computes $EPC_1 = 0.71$ and $EPC_2 = 0.70$, and therefore sets $s^* = s_2$.

Using s_2, the second invocation of the algorithm occurs at $t = 8$, causing the postponement of t_{a_2} to $t = 10$. Running at speed s_2, the CPU can wake up at 13, but, for the activity constraint, t_{w_2} is set to $b_1^s = 12$. Consequently, $t_{idle_2} = 13$ and $t_{e_2} = b_1^e = 15$. Instead, using speed s_1, the algorithm obtains $t_{w_1} = 12$, $t_{idle_1} = 18$, and $t_{e_1} = b_1^e = 15$.

In such a scenario, the energy consumptions are $EPC_1 = 0.73$ and $ECP_2 = 1$, and therefore the chosen speed is $s^* = s_1$.

4.2.4 Experimental results

A set of experimental results is here reported to show the effectiveness of our approach with respect to other classical solutions. The results are obtained by simulation using a synthetic workload under three power consumption profiles derived from the Microchip dsPic33FJ256MC710 microcontroller:

- *DPM-sensitive* $P(s) = 5.6s^2 + 246.12s + 25.93$;
- *DVFS-sensitive* $P(s) = 330.62s - 53.32$;
- *Mixed* $P(s) = 150.55s^2 + 24.5s + 100.78$.

The frequency range of the CPU used in the simulation is $[12.5, 40]$MHz. The sleep state consumption P_σ is 1.49 mW and the wake up time takes about 20 ms. The standby state has a higher consumption P_s of 9.9 mW, but a shorter wake up time within 8 cycles. All the simulations have been executed using a set of 8 evenly distributed frequencies.

For comparison purposes, the proposed algorithm DEAS have been compared with the three following scheduling policies:

- *EDF* with no energy considerations, where the processor is assumed always active at the maximum frequency, even during idle intervals;

- *pureDVFS* on top of EDF, where the CPU runs with the minimal speed, computed off-line, that guarantees feasibility according to the task set. The actual speed is the lowest frequency greater than the minimal one;

- *pureDPM*, where, as soon as there is an idle time and no assigned bandwidth, the task execution is postponed as much as possible and then scheduled by EDF at the maximum speed.

An execution scenario is characterized by the tuple (U, n_t, B, n_B), where U denotes the utilization of the task set, n_t the number of tasks, B the communication bandwidth (expressed as a percentage of the hyperperiod), and n_B the number of chunks in which the bandwidth is split. All the slots are generated with the same length, whereas slot positions are randomly generated with a uniform distribution.

Given the total utilization factor U, individual task utilizations are generated according to a uniform distribution [BB05].

Payload and message deadlines are generated to meet the hypothesis on messages guarantee. The computed values are not described here because they have no effect on the task scheduling algorithm.

Trying to find a trade-off between the simulation accuracy and the simulation time (it increases exponentially with the number of tasks), each result was computed as the average consumption of 30 executions. To simplify comparisons, the results are normalized against the value obtained applying the EDF policy to the same tuple (U, n_t, B, n_B).

In the first experiment, the energy consumption is evaluated as a function of the utilization U and the number of tasks n_t. All the three algorithms have been tested with Bandwidth $B = 0.3$, $n_B = 5$ chunks and three different utilization factors. The results show that both U and n_t do not affect energy consumption significantly, therefore the graph is not reported.

The next experiments evaluate the energy consumption, under different power models, as a function of the utilization factor U with $n_t = 7$, $B = 0.3$ and $n_B = 10$.

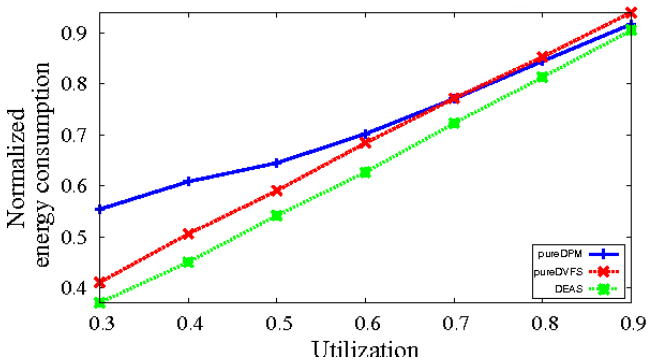


Figure 4.15: Analysis of consumptions with a DPM-sensitive power model.

Results show that DEAS always outperforms the other algorithms for all power models and for any utilization.

As shown in Figure 4.15, due to the activity constraint posed by the bandwidth slots, DEAS outperforms pureDPM even in DPM-sensitive models. Instead, Figure 4.16

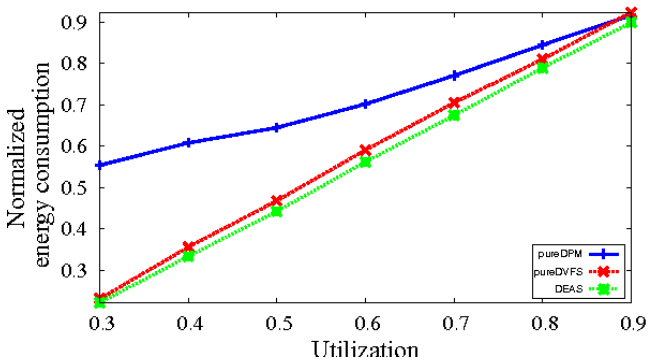


Figure 4.16: Analysis of consumptions with a DVFS-sensitive power model.

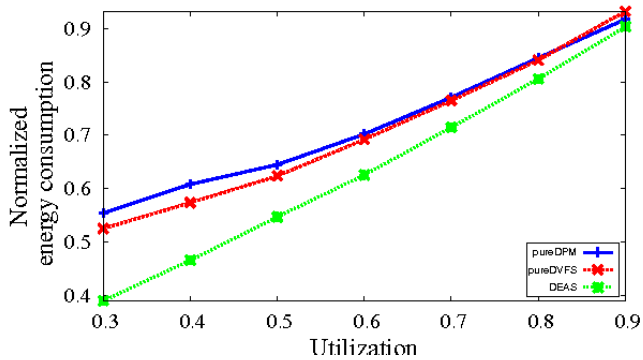


Figure 4.17: Analysis of consumptions with a Mixed power model.

shows that in DVFS-sensitive power models such a constraint has no effect on pureDVFS: keeping the system active represents the default behavior and, with respect to the analyzed power model, the best solution. For this reason, DEAS and pureDVFS have similar performances.

Under a DPM-sensitive and in a mixed context, Figure 4.15 and Figure 4.17 shows that pureDVFS acts better than pureDPM for low U values, because the CPU can not be switched off inside bandwidth slots. Instead, for higher utilization values, the consumptions are similar. Note that all the graphs show that DEAS is always able to select the right balance between DVFS and DPM depending on the specific characteristics of the architecture.

4.3 Energy-aware framework in tiny RTOS

This work presents a module for managing power consumption in tiny kernels for real-time embedded systems with limited resources, such as memory, CPU and power supplier. The proposed module achieves considerable energy savings, satisfying the application's timing constraints and exploiting a high modular design. The proposed solution has been implemented in the Erika Enterprise kernel to manage CPUs, timers, and servomotors. Experimental results show the effectiveness of the approach high-

lighting how the module can be used to select the most appropriate policy for a specific application on a given architecture.

The analysis proceeds in Section 4.3.1 presenting the architecture of the kernel module, its working flow and its interaction with the operating system. Section 4.3.2 describes the policies implemented in the module, while Section 4.3.3 reports the experimental results performed on the hardware.

4.3.1 Architecture

The energy saving module (also referred to as the *Power Manager*) is part of the kernel and interacts with the scheduler, the hardware devices, and the application, as illustrated in Figure 4.18. While the scheduler selects the next task to execute, the Power Manager chooses an appropriate running configuration (i.e., speed and voltage).

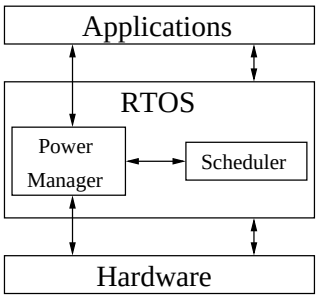


Figure 4.18: Interaction of the Power Manager with the other system components.

A block diagram of the Power Manager is reported in Figure 4.19. It consists of three hierarchically organized modules: the Application Programming Interface (API), the CPU Manager and the Devices Manager. Such a modular implementation allows the programmer to easily remove sub-components when not needed by the application, so helping to reduce the footprint.

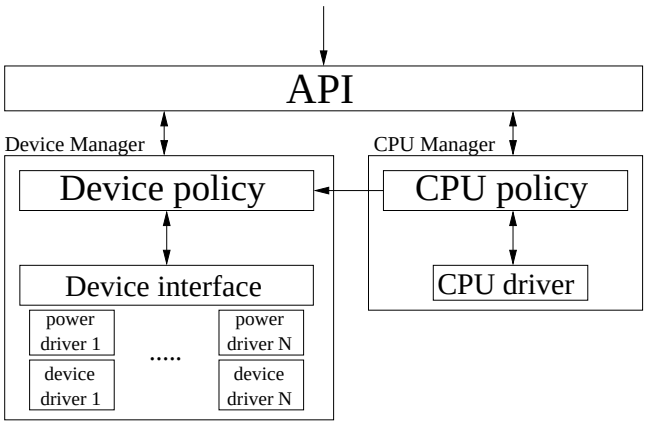


Figure 4.19: Block diagram of the Power Manager.

The API module implements the interface defined for the interaction with the kernel and the applications. The CPU Manager is responsible for the power management of the CPU. Using a set of special callback functions called *hooks*, the kernel informs

the module about four scheduling events: task activation, task termination, task preemption, and task dispatch.

The *CPU policy* submodule implements the energy saving policies, which typically select the best speed to meet the applications constraints, while satisfying a given set of performance requirements. The *CPU driver* is in charge of setting the CPU parameters, such as frequency and energy saving state. It is located at the lowest abstraction level as its code is hardware-dependent.

The Devices Manager handles internal and external peripherals. Inside it, the *Device policy* submodule contains all the device policies, developed according to the *Device Interface*, which offers a single access point to the devices. For each of them, two stacked components, *Power driver* and *Device driver*, abstract the device behavior using a discrete set of states, as shown in Figure 4.20. Each state is characterized by a specific power consumption and quality of service level.

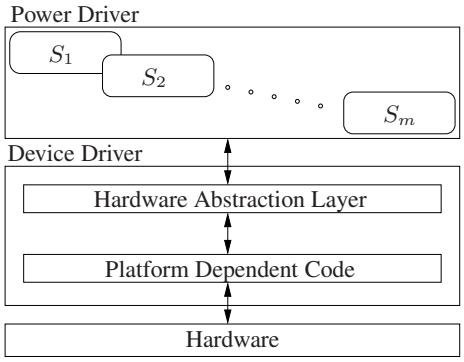


Figure 4.20: Device stack.

The link between the CPU and the Device module is necessary to adjust the configurations of devices, whenever a speed scaling or mode switching event occurs. For instance, when a new speed is set, the system timers need to be automatically reconfigured to offer the same tick period.

When an internal error occurs, a user-defined callback function is invoked, demanding the user to manage the exception. A typical scenario could occur on speed scaling: if a device detects that the modified configuration is not able to guarantee the same performance of the previous state, the callback is invoked to solve the situation. For instance, if an UART transceiver with a modified system speed is not able to sustain the communication baud rate, the user has to specify how to fix this issue.

Sample scenarios

This section describes two examples to better explain how the modules interact with each other. The first scenario, shown in Figure 4.21, supposes that a new task instance becomes running. The kernel, after having managed the event, informs the CPU Manager by invoking the corresponding hook (1). Once the event is notified, the active policy selects the best frequency to execute the actual workload within the specified timing constraints. The new speed is communicated to the CPU driver, which makes it effective (2).

Once the new frequency is fully operational, the CPU manager notifies the new configuration to the Device Manager (3), which in turn informs the devices under its control (4) (5). Finally, each device sets its hardware registers to obtain the same

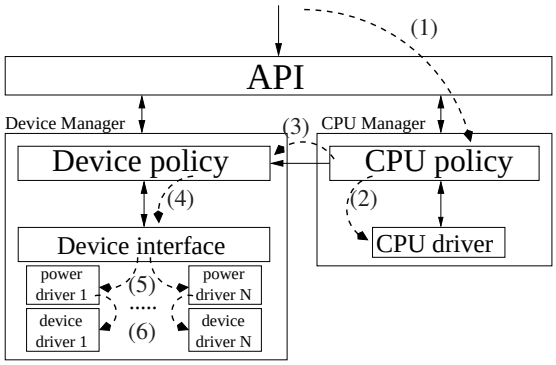


Figure 4.21: First example of modules interaction.

performance with the new configuration (6). If this is not possible, the module invokes the error callback to solve the situation.

Figure 4.22 presents another situation in which a running task has to control a servomotor to properly hold a given load with the minimum energy, using a torque sensor. The controller reads the torque sensor, computes the load value and notifies it, through the API, to the Device Manager (1).

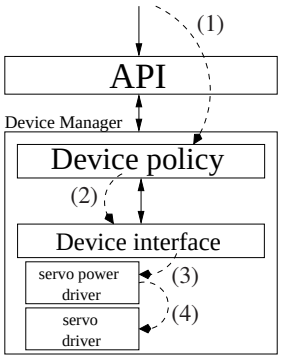


Figure 4.22: Second example of modules interaction.

The active policy chooses the appropriate state able to hold the actual load with the minimum energy consumption and notifies it, through the Device interface (2), to the corresponding Power driver (3), which translates the communicated state in an appropriate set of commands for the specific servo driver, modifying the actuator performance (4).

4.3.2 Implemented policies

This section presents the policies implemented inside the CPU Manager and the Device Manager. Such policies are configured offline and are automatically invoked at runtime without any user interaction.

CPU policies

The policies implemented in the CPU Manager work for a single CPU and adopt a discrete speed set and s' denotes the lowest speed ensuring the task set feasibility in

the worst-case scenario.

The following three policies are implemented:

- *OnLine Dynamic Voltage Scaling* (OLDVS) is a policy proposed by Lee and Shin [LS04] that selects the minimum available speed to prolong a task execution time up to its WCET;
- *Bonus Sharing DVFS* (BSDVFS) is a variant of OLDVS proposed in this work to take switching overheads into account;
- BSDVFS* is a variant of OLDVS*, originally introduced by Gong et al. [GSL07], extended in this work to take switching overheads into account.

All the analyzed policies compute the most suitable frequency to exploit tasks early terminations. Note that the tasks deadlines are not considered to slow the CPU down: all policies exploit the unused computation time, if any, from the previous jobs, prolonging the execution of the current job (at a lower speed) until its worst-case finishing time e_i (that is, the time at which the task would finish in the worst case at speed s').

To better illustrate the implemented approaches, the three policies are instantiated on a CPU with a set S of three speeds $S = \{0.5, 0.75, 1\}$ and are applied to a task set consisting of two tasks with WCETs equal to $C_1 = 40$ and $C_2 = 30$ (note that all WCETs values refer to the tasks executing at the highest speed $s = 1$). For the sake of simplicity, we assume that task set parameters are such that $s' = 1$.

Figure 4.23 shows a schedule in which each task executes for its WCET on the CPU running at speed s'. Having no early terminations, the speed is not changed and no energy can be saved in this case.

If τ_1 and τ_2 arrive at $a_1 = 0$ and $a_2 = 5$, their worst-case finishing times will be at $e_1 = 40$ and $e_2 = 70$, respectively.

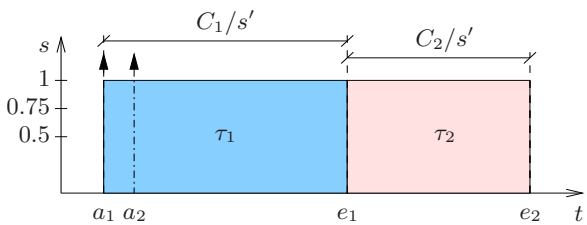


Figure 4.23: Tasks execute for their WCETs.

To apply any of the policies listed above, the system has to keep track of the actual execution time used by a task τ_i. Such a monitoring can be efficiently implemented by starting a timer each time a task becomes running and stopping it when the task is preempted or completed. If ε denotes such an interval executed at a speed s, the remaining WCET of the task can be computed as

$$c_i = C_i - \varepsilon\, s. \tag{4.21}$$

Moreover, a *bonus time*, denoted as B, is introduced to account for the unused time accumulated by previous tasks' executions: when τ_i finishes, the saved time c_i is added to B, which can be exploited as an extra time available for the next scheduled task.

Figure 4.24 shows the schedule produced by OLDVS when τ_1 finishes at time $t = 8$. At the beginning, τ_1 runs at the highest speed since no computation time is saved at time $t = 0$ (thus $B = 0$). At time $t = 8$, τ_1 completes, saving $c_1 = 32$ units of time.

Thus, B is incremented by c_1 and τ_2 can exploit B to execute at a slower speed such that $C_2/s = (C_2 + B)/s'$. In general, having a bonus time B, a task τ_i with residual WCET c_i can still meet its deadline by running at the speed

$$s_{OLDVS} = \min_{s_j \in S} \left\{ s_j \geq \frac{c_i}{c_i + B} s' \right\}. \tag{4.22}$$

In the example shown in Figure 4.24, since $B = 32$ and $C_2 = 30$, the lowest feasible speed is $s_{OLDVS} = 0.5$. With such a speed, τ_2 would finish in the worst-case at $t = 68$.

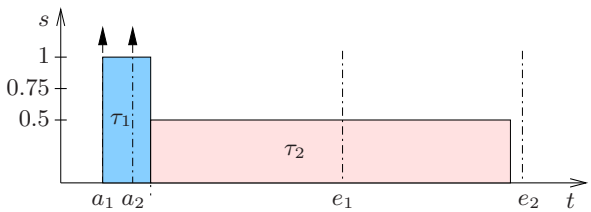


Figure 4.24: Schedule produced by OLDVS.

Figure 4.25 illustrates the schedule produced by BSDVFS, when taking switching overheads into account. In such a scenario, the minimum speed under which a task τ_i with residual WCET c_i and bonus time B can still meet its deadline is computed as follows:

$$s_{BSDVFS} = \min_{s_y \in S} \left\{ s_y \geq \frac{c_i}{(c_i + B)/s' - \Delta_{BSDVFS}(s_x, s_y)} \right\} \tag{4.23}$$

where

$$\Delta_{BSDVFS}(s_x, s_y) \triangleq \mu_{s_x \to s_y} + \mu_{s_y \to s'}.$$

The term $\mu_{s_y \to s'}$ accounts for the overhead needed for restoring the speed at s' in the case the next running task is not able to slow the CPU down further. In the considered example, the switching overheads are considered symmetric ($\mu_{s_x \to s_y} = \mu_{s_y \to s_x}$) and proportional to the speed gap. In particular: $\mu_{0.5 \to 0.75} = 2$, $\mu_{0.5 \to 1.0} = 5$, and $\mu_{0.75 \to 1.0} = 2$.

For the given task set, the feasibility test is satisfied only for $s = 0.75$ and $s = 1$, since for $s = 0.5$ τ_2 completes at $t = 78$ (i.e., beyond time $e_2 = 70$). Therefore, the running speed is set to 0.75, causing τ_2 to finish in the worst case at $t = 50$.

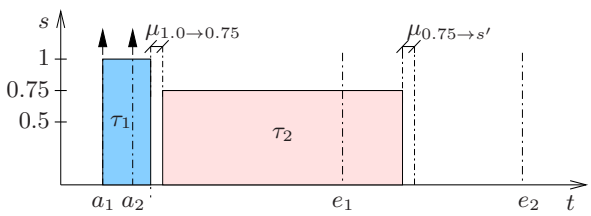


Figure 4.25: Schedule produced by BSDVFS.

The idea behind BSDVFS* consists of splitting a task τ_i in two parts, τ_i^L and τ_i^H, with WCETs c_i^L and c_i^H, executed at different speeds, s_L and s_H, set as the lower and the higher adjacent speed of s_{BSDVFS}. The switching instant, and therefore the two values (c_i^L,c_i^H) are computed to prolong τ_i's execution until its worst-case finishing time e_i.

Hence, they are computed as follows:

$$\begin{cases} c_i^L + c_i^H = c_i \\ \dfrac{c_i^L}{s_L} + \dfrac{c_i^H}{s_H} + \Delta_{BSDVFS^*}(s_x, s_L, s_H) \leq \dfrac{c_i + B}{s'} \end{cases} \tag{4.24}$$

where

$$\Delta_{BSDVFS^*}(s_x, s_L, s_H) \triangleq \mu_{s_x \to s_L} + \mu_{s_L \to s_H} + \mu_{s_H \to s'}.$$

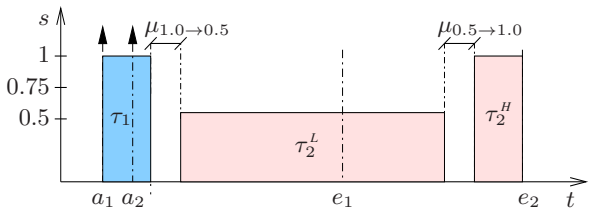


Figure 4.26: Schedule produced by BSDVFS*.

It is worth observing that BSDVFS* is the only method able to fully exploit the time bonus to make the current task to complete at its worst-case finishing time e_i. Note that if a task instance finishes earlier, most of the execution is spent at the lower speed s_L, so achieving higher energy reduction.

Figure 4.26 shows the schedule produced by BSDVFS*, using $s_L = 0.5$ and $s_H = 1$, which are the adjacent speeds of $s_{BSDVFS} = 0.75$. In this example, $\mu_{s_H \to s'} = 0$ since $s_H = s' = 1$.

According to Equation (4.24), the execution times of τ_2^L and τ_2^H result to be $c_2^L = 22$ and $c_2^H = 8$, respectively. Note that τ_2 finishes exactly at time $e_2 = 70$.

Device policies

The policies implemented in the Device Manager support timers and servomotors.

Servomotors are devices driven by Pulse-Width Modulation (PWM) signals, whose absorption peak is concentrated at the beginning of the signal period with a constant intensity and a duration proportional to the detected angle error.

The Power driver offers m states, each one identified by a specific PWM period. The policy inside the Device Manager associates a specific power state to the required torque according to a pre-specified internal look-up table.

To be implemented, the servo driver (or a PWM peripheral) uses a timer to generate the control signals and the Device Manager interacts with such peripherals to vary the PWM period.

Despite of the negligible energy consumption, timers are managed by the module to maintain the consistency of the system time independently of the running speed. The Device Manager does not provide any policy for them and the Power driver offers only two states, ON and OFF, corresponding to the timer active and timer inactive modes, respectively.

4.3.3 Experimental results

The Power Manager has been developed as module of the Erika Enterprise kernel and tested on the Evidence FLEX boards equipped with a Microchip dsPIC33FJ256MC710 microcontroller.

CPU

A set of experiments has been carried out to evaluate the impact of the three policies on the energy consumption. The experimental measurements refer to the whole board. The CPU driver supports eight different frequencies: 40, 35, 30, 20, 16, 10, 8 and 2 MIPS (Million of Instructions Per Second).

The switching overhead depends on the specific frequency levels, because the lowest frequency is obtained directly from the external clock signal, while the other frequencies are derived by a PLL. Switching the PLL on takes about 1ms, while turning it off or adjusting it to any other frequency takes between 4μs and 40μs.

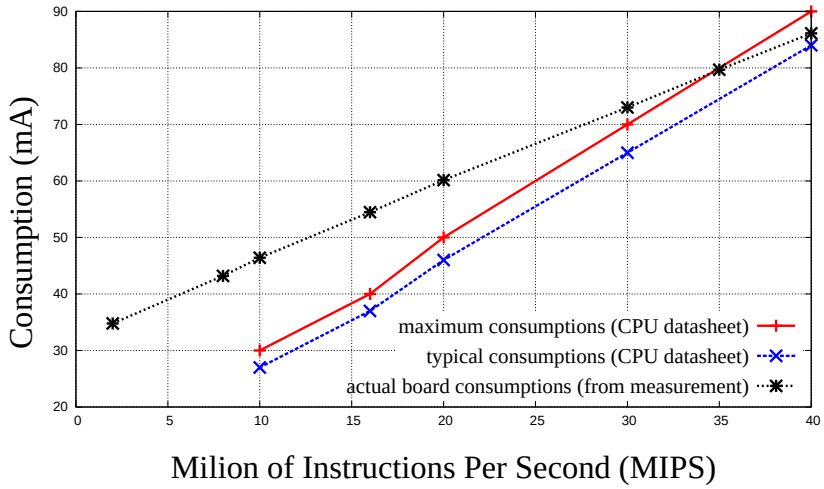


Figure 4.27: CPU power consumptions.

Figure 4.27 shows the current consumption of the CPU as a function of the frequency. The upper curve in the figure refers to actual measurements on the entire board, whereas the others two are derived from the datasheet and refer to the CPU only. Note that, for the considered architecture, halving the frequency doubles the execution time, but does not reach a current consumption of 50%. For instance, reducing the frequency from 40 to 20 MIPS, the current consumption goes from 86.12 mA to 59.12 mA. Such a result indicates that DVFS approaches are not effective on this architecture, where higher saving would be achieved by algorithms that run the application at higher speeds.

In the next experiment the three policies have been tested on ten periodic tasks with a total worst-case utilization $U_{wc} = 0.98$. The lowest frequency which guarantees the task set feasibility in the worst case is 40 MIPS (corresponding to a speed $s' = 1$).

Figure 4.28 shows the energy consumption (normalized with respect to the case of no online policy) as a function of the ratio of the actual utilization (U_{real}) and the worst-case one (U_{wc}). As observed above, in the considered architecture, policies using higher speed achieve a lower energy consumption. Therefore, although BSDVFS* is able to exploit slower frequencies than BSDVFS, its average consumption is similar to BSDVFS, because it is compensated by longer execution times. Note that at high utilization ratios ($U_{real}/U_{wc} > 0.5$) OLDVS is characterized by higher energy consumptions because, by neglecting switching overheads, it is able to select lower speeds. On the other hand, at low utilization ratios ($U_{real}/U_{wc} \leq 0.5$), both BSDVFS and BSDVFS* achieve higher consumptions because, at the end of each job, they re-

store the speed to s' to ensure task set feasibility. Such an effect is enhanced for very low utilization ratios due to the higher overhead (1 ms) introduced when switching to the minimum frequency.

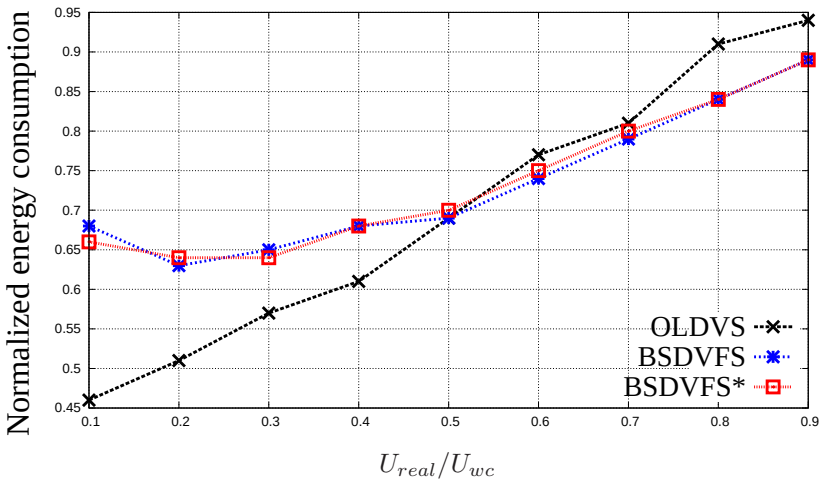


Figure 4.28: Normalized energy consumptions of the policies.

Devices

Another set of experiments has been performed on a servomotor to derive a policy for driving the device with the minimum average energy consumption. The servomotor used in this test is a Hitec HS-645MG, characterized by a minimum absorption of 12.56mA, and a peak current of 1A. The peak occurs at the beginning of the PWM period and has a duration proportional to the detected angular error.

The experimental tests compare the measured mean power consumption as a function of the applied torque, using three different PWM periods: 10, 20 and 40 ms. As shown in Figure 4.29, the effectiveness of each period depends on the energy needed to correct the accumulated error between two consecutive updates.

Note that for very small torques (< 0.5 kg $\times$ cm) the consumption is not affected by the PWM period, because the angular error on the axis is below the threshold used by the internal position controller. Low torques ($\in [0.5, 1)$ kg $\times$ cm) typically generate similar errors for any PWM period, leading to similar energy costs per update; therefore, longer periods produce less updates per time unit and consume less energy. For torques higher than 1.0 kg$\times$cm, a period of 40 ms copes with higher errors, accumulated between two consecutive updates, resulting in a higher consumption. Moreover, this period cannot guarantee an angular error less than $5°$ with torques greater than 1.5 kg$\times$cm; hence, the measures for such a period are not considered for higher torques. For medium torques ($\in [1, 1.6)$ kg $\times$ cm), the errors produced by 10ms and 20 ms PWM periods are similar, hence the longer period (20 ms) leads to a better performance. The shorter period (10ms) is more suited for heavier loads, because it frequently corrects smaller errors, so leading to lower consumptions.

As a result, the implemented policy binds torque ranges with the period that minimizes the energy consumption, according to the results reported in Figure 4.29. To optimize the implementation, the results are stored in a look-up table that associates the period leading to the minimum consumption to the corresponding torque range,

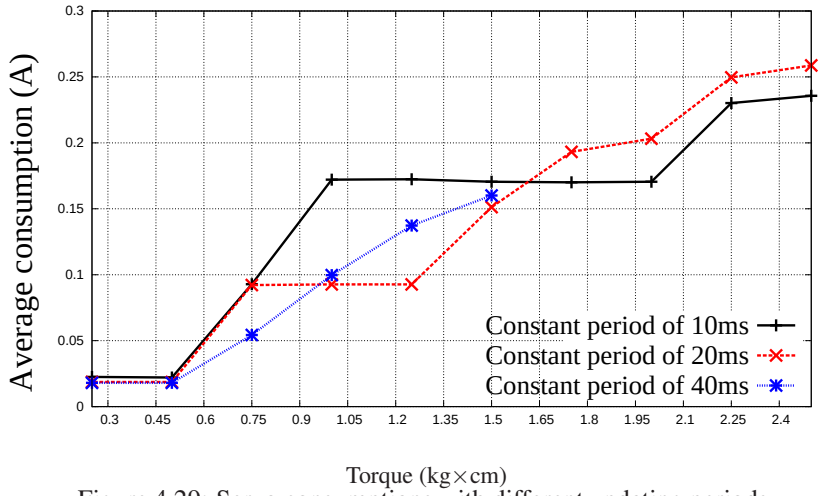


Figure 4.29: Servo consumptions with different updating periods.

which also defines a power state. Table 4.2 shows the specific state values derived from Figure 4.29.

Torque range ($kg * cm$)	Power state	Period (ms)
$[0.0, 1.0)$	STATE_2	40
$[1.0, 1.6)$	STATE_1	20
$[1.6, 2.5]$	STATE_0	10

Table 4.2: Look-up table used by the servomotor policy.

4.4 Algorithm evaluation

This work discusses the factors to consider when deciding which technique to implement on a given single-core architecture, highlighting the limitations of the current mainstream.

The principal DVFS and DPM algorithms are compared on three representative platforms, evaluating their behaviors for different task characteristics. The current belief that considers DVFS algorithms less effective than DPM ones is questioned, evaluating the system parameters that mostly affect the validity of this assumption on actual hardware.

The following algorithms, which are detailed in Chapter 3, have been selected for their popularity and their reasonable runtime complexity:

- DVFS algorithms:
  - SVS [PS01] (Static Voltage Scheduling): only the static slack $(1 - \sum_{\tau_i} \frac{C_i}{T_i})$ is exploited;
  - DRA-OTE [AMMMA01] (Dynamic Reclaiming Algorithm - One Task Extension): mostly the dynamic slack due to task early terminations is used to scale the speed down;

- LA-DVS [PS01] (Look-Ahead RT-DVS): both static and dynamic slacks are considered;

- DPM algorithms:
  - CS-DVS-P [JPG04] (Critical Speed DVS with Procrastination): the maximum time that a task can be delayed is computed offline;
  - LC-EDF [LRK03] (Leakage-Control EDF): job delays are entirely computed at runtime.

Results confirmed that the actual assumption stating that DPM algorithms work generally better than DVFS ones on actual hardware is true. However, experiments also highlighted that such a consideration can easily be invalidated when other aspects are involved in the analysis. For instance, short periods can drastically reduce the effectiveness of DPM algorithms, and tasks that intensively interact with peripherals can make DVFS algorithms more effective. In addition, it was empirically shown that task early terminations help both DPM and DVFS algorithms in further reducing the energy consumption.

This analysis proceeds showing in Section 4.4.1 a motivational example. Section 4.4.2 provides the power models of several real processors while Section 4.4.3 analyzes the algorithms' performance.

4.4.1 Limits of existing approaches

The energy needed to execute a job is the product of the active power and the execution time at the selected speed. Note that a higher speed reduces the execution time, but increases the power consumption. For this reason, the concept of critical speed s^* has been introduced for defining the speed that minimizes the overall active energy consumption. Analytically, s^* is computed as the speed that minimizes the active energy consumption per cycle $\frac{P(s)}{s}$, and can be derived from $\frac{dP(s)/s}{ds} = 0$.

As an example, assume a processor with ten speeds uniformly distributed from 0.1 to 1.0, and with active power consumption $P(s) = 0.9s^3 + 0.1$. The dominant non-linearity in the power function makes it a DVFS-sensitive architecture, where the speed that minimizes the energy consumption is $s^* = 0.4$. When instead the power consumption is characterized by a significant constant component (independent of the speed), as in $P(s) = 0.3s + 0.7$, the critical speed results to be equal to the maximum available ($s^* = 1$), hence speed scaling is not effective to minimize the active energy consumption. Such a behavior is typical in DPM-sensitive architectures, which integrate a significant amount of memory, I/O controllers and other devices whose power consumption does not depend on the processing speed.

One big limitation of the above approach is that it only analyzes the active power consumption, neglecting the power consumed when the processor is not executing. Therefore, the critical speed gives a reliable indication of the best operating frequency *only if the system is assumed to consume no power when the processor is idle*. As the static power consumed during idle intervals gets bigger, the critical speed is less suitable to characterize the best operating frequency of the processor. To have a more precise characterization of the power consumption, it is therefore necessary to account for the time spent in low-power states.

Depending on the ability of the system to exploit deeper sleep states, the best operating speed can be higher or smaller than the critical speed. For example, a DVFS-sensitive architecture with $s^* = 0.4$ could be better operated at a higher speed if the

slack created could be spent in a low-power state with $P_\sigma < P(s^*)$. Conversely, it could be beneficial to reduce the speed of a DPM-sensitive platform, even with $s^* = 1$, if the idle intervals are not sufficiently large to allow entering a low-power state.

Intuitively, the highest consumption is obtained when the algorithm is never able to put the processor in sleep state, so that the slack time is entirely spent in the more consuming idle state. Conversely, a lower bound on the overall power consumption can be derived considering that any idle interval fully exploits the deepest sleep state.

Figure 4.30 shows the average power consumption of a NXP LPC1768 chip ($P(s) = 0.3s + 0.7$) for different task utilizations. The power consumed in idle and in sleep state is taken from real measurements, as detailed in Section 4.4.2. The straight lines represent the upper and lower bounds on the DPM power consumption, while the staircase line shows the DVFS consumption obtained from executing the generic task set at the slowest available feasible speed. Even if the device is DPM-sensitive, with a critical speed equal to one, the performance of a simple DVFS approach is rather close to the ideal DPM performance. Unlike the actual general opinion that DPM algorithms work always better than DVFS ones, the example shows that task set characteristics are crucial to decide which technique works best, as short periods and reduced idle intervals (or, equivalently, large BETs) can forbid the use of deeper low-power states, making DVFS approaches more competitive.

What said above is even more relevant when task computation times do not entirely scale with the speed, that is when each task τ_i has $\alpha_i > 0$. Indeed, memory-bound tasks with a large constant part $\alpha_i C_i$ tend to privilege DVFS techniques, as similar execution times can be obtained executing at a lower speed, with a smaller power consumption.

Finally, when the entire consumption of the SoC is considered, the dissipation in low-power states is not as low as expected, because many components can not be turned off (such as the main memory), thus reducing the impact of DPM-based algorithms.

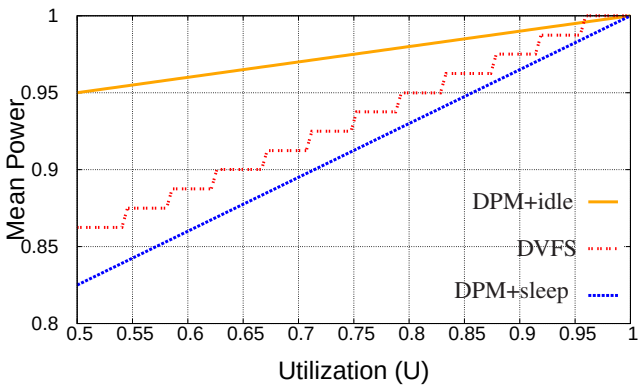


Figure 4.30: DPM bounds vs. simplest DVFS algorithm.

4.4.2 Measurements

This section aims at providing a detailed analysis of the power characteristics of the following processors:

1. Microchip dsPic33FJ256MC710, with frequencies within $[4, 40]$MHz with step of 1MHz;

2. NXP LPC1768 (ARM Cortex M3), characterized by frequencies in $[36, 96]$MHz with step of 4MHz;

3. Intel Pentium4, providing frequencies from 375MHz up to 3.0GHz with step of 375MHz.

These architectures have been selected to cover a wide spectrum of real platforms, from small digital signal processors to average embedded controllers, including reliable general-purpose processors.

Except for the last processor, the other measurements refer to the entire platform (i.e., core, cache, memory and peripherals) as everything is embedded on a single chip.

Performance evaluation

The following measurements have been carried out running the Coremark benchmark [COR] as workload, which implements CPU bound code ($C_i(s) = C_i/s$).

The benchmark scores at the maximum speed for the analyzed platforms are 57 (without hard FPU), 218.7 and 14413.0, respectively.

These results have been achieved on systems with light workload, meaning that the processor was entirely assigned to the benchmark in execution.

Active state

The power functions of the platforms, considering normalized speed and normalized power, are reported in Figure 4.31, whereas the numerical values of the parameters of Equation 2.3 are summarized in Table 4.3. Note that, on these platforms, the critical speed is always equal to the maximum available ($s^* = 1.0$).

Processor	K_3	K_2	K_1	K_0
LPC1768	0.0	0.0	0.3	0.7
dsPic33	0.0	0.0	0.55	0.45
Pentium4	0.0	0.09	0.44	0.47

Table 4.3: Parameters of the power models.

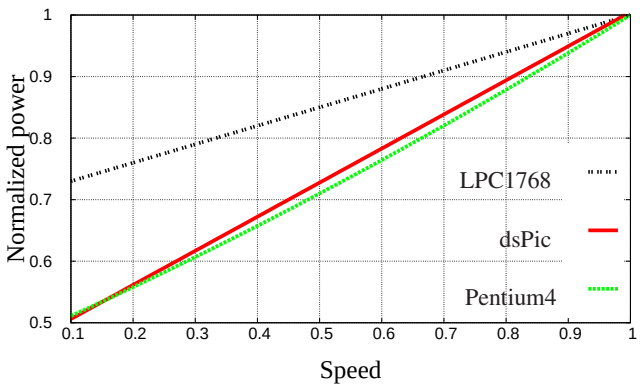


Figure 4.31: Power consumption models.

Concerning the components of the power consumption, the static dissipation, mostly due to leakage currents, is represented by the coefficient K_0 as it is speed independent,

while the rest of consumption is ascribed to the dynamic dissipation. Note that the static component includes the consumption due to the cache, which can not be turned off.

Considering the absolute power consumption, the dsPic and the LPC1768 processors have similar consumptions (0.56W and 0.695W, respectively), but the latter is around four times more performing. On the other hand, the Pentium4 consumes 88.8W when active.

Low-power states

Concerning the low-power states, the dsPic processor provides two states: Idle and Sleep. The first one switches few components off, leading to a consumption equal to 66% of the power at the maximum speed, with a state transition of 8 clock cycles. The Sleep state consumes 31% of the maximum power, with a break-even time around 15ms (the clock crystal and PLL are put off).

The LPC1768 offers Sleep, Deep Sleep, Power Down, and Deep Power Down states, which consume, with respect to the consumption at the maximum speed, 90%, 70%, 70% and 65%, respectively. The overhead is negligible for the lighter state, whereas it takes about 10 milliseconds for the deepest.

On the Pentium4, the ACPI module exploits only a single state, called Idle, with a relative consumption of 34% and a break-even time of a few hundreds milliseconds [NRM+06].

4.4.3 Experimental results

This section presents a set of simulation experiments carried out for evaluating the considered algorithms on the different platforms under different scenarios.

The synthetic task sets are composed of 10 periodic tasks randomly generated using the UUniFast algorithm [BB05]. For each utilization step of 0.05, 30 different task sets were generated and tested.

The speed scaling overhead (in the order of μs) was considered negligible with respect to the task execution times (in the ms). The frequencies of buses and memories were assumed to be constant and independent of the processor frequency.

The following simulations are divided into three categories: analysis of the algorithm performance in the worst case, online improvement due to task early terminations and impact of speed-independent code.

Worst-case analysis

This section presents the average power consumption obtained by the considered algorithms assuming always the worst-case execution. Since task early terminations are not considered, the algorithm DRA-OTE is not taken into account as it would exhibit the same performance of SVS. Computation times are assumed to scale linearly with the speed ($\forall \tau_i \in \Gamma : \ \alpha_i = 0$).

The results are reported in Figure 4.32 considering two scenarios in which the shortest task period is larger than or comparable to the sleep break-even time, respectively. When the shortest period is much larger than the break-even time, DPM-based algorithms tend to work better, especially at lower utilizations when a considerable amount of slack is available. When instead the shortest period becomes comparable to the break-even time the performance of DPM-algorithms drops significantly.

The first case is shown in Figure 4.32(a), 4.32(c) and 4.32(e), where periods are generated in the range $[25, 250]$ms for the dsPic and LPC1768, and in the range [300, 3000] ms for the Pentium4. Among DPM algorithms, CS-DVS-P has always a lower consumption than LC-EDF. Among DVFS-based approaches, SVS and LA-DVS obtain the same performance on the dsPic because the power function is linear. For example, consider a job of 10ms. When it is executed at $s = 0.5$, hence for 20ms, the energy consumption is 14.554 mJ. If the same job is executed for 15ms at $s = 0.\bar{3}$ and for 5ms at $s = 1.0$, leading to the same overall execution time (20ms), then the overall energy consumption is again $P(0.\bar{3}) \cdot 15 + P(1.0) \cdot 5 = 14.554$ mJ. Hence, an aggressive DVFS algorithm may be ineffective when the power consumption is linear. However, LA-DVS consumes less than SVS on the other two platforms because the limited number of speeds does not allow SVS to exploit the entire static slack. The best DPM algorithm (CS-DVS-P) has always a better performance than DVFS ones, although the difference is rather small for the LPC1768 due to the limited savings allowed in sleep mode with this architecture. Instead, LC-EDF has always the largest power consumption.

In the second case, the minimum task period is decreased, becoming comparable to the low-power state transition overhead (Figure 4.32(b), 4.32(d) and 4.32(f)). More precisely, periods were generated in the range $[8, 80]$ms for the dsPic and LPC1768, and in the range $[100, 1000]$ms for the Pentium4. As expected, while the performance of DVFS strategies remains the same, that of DPM algorithms drops significantly, making DVFS strategies more competitive.

Average execution analysis

This section considers the average power consumption obtained by the algorithms taking into account that jobs may terminate earlier than their worst case. More precisely, the actual execution time of each job of τ_i is generated in the range $[C_i/10, C_i]$.

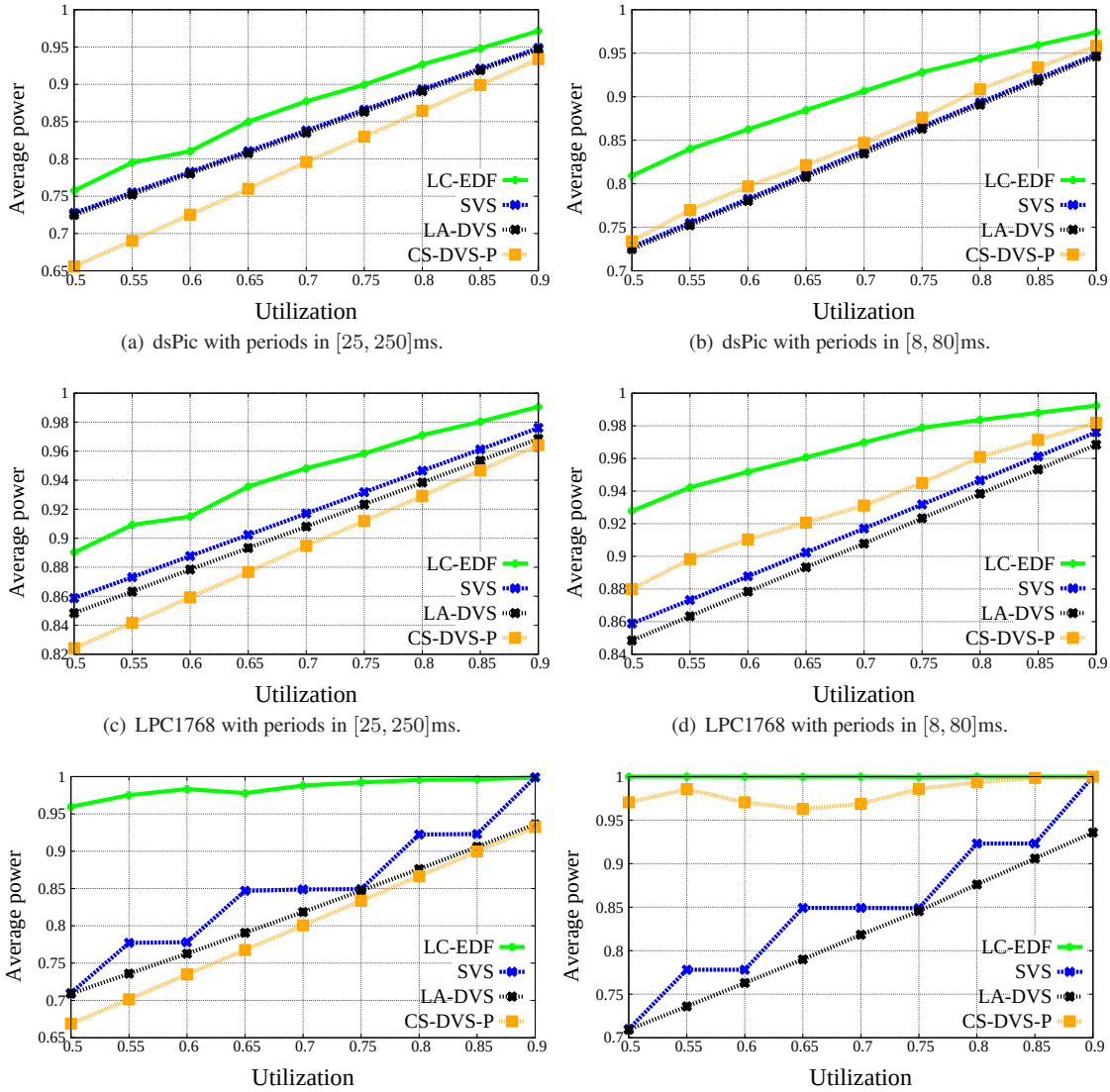
The results for the three platforms are shown in Figure 4.33, assuming periods in $[25, 250]$ms for the dsPic and LPC1768 platforms, and in $[300, 3000]$ms for the Pentium4 (shortest period larger than sleep BET).

The trend among DPM algorithms is not altered, with CS-DVS-P still guaranteeing better performance than LC-EDF. Among DVFS techniques, SVS does not improve as it cannot take advantage of the online slack freed by task early terminations. However, DRA-OTE has always worse performance than SVS, because it scales the speed down in order to make jobs last as long as the worst case. Since the critical speed is equal to the maximum one, DRA-OTE increases the energy consumption without introducing any benefit. LA-DVS is the best DVFS algorithm as jobs can usually end before scaling up to high speeds, leading to a lower consumption.

Similarly to the previous analysis, decreasing task periods makes DPM algorithms much less effective than DVFS ones. The above trend is the same when computation times have different variances between worst and best case execution times.

Speed-independent code

This section analyzes the case in which computation times do not fully scale with the speed. Such a behavior is modeled by the α parameter. More precisely, $\alpha = 0$ means that the computation fully scales with the speed ($C(s) = C/s$), whereas $\alpha = 1$ leads to a constant computation time, completely speed independent.



(a) dsPic with periods in $[25, 250]$ms.

(b) dsPic with periods in $[8, 80]$ms.

(c) LPC1768 with periods in $[25, 250]$ms.

(d) LPC1768 with periods in $[8, 80]$ms.

(e) Pentium4 with periods in $[300, 3000]$ms.

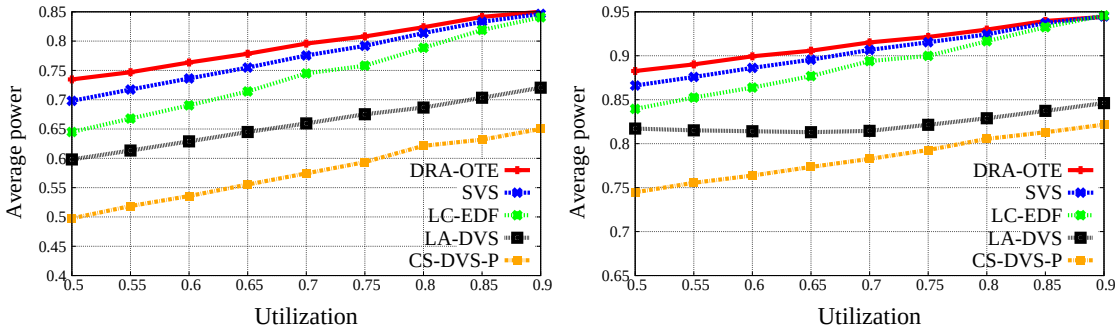
(f) Pentium4 with periods in $[100, 1000]$ms.

Figure 4.32: Average power consumption assuming worst-case execution.

For example, a job with $\alpha = 0$ that lasts for 10ms at the maximum speed, will take 20ms when executed at $s = 0.5$. On the other hand, if $\alpha = 0.5$, only half of the execution time is scaled with the speed, so that the job would last for 20ms when executed at $s = 0.\bar{3}$. Both cases have a similar completion time, but the second one has a lower speed, resulting in a smaller power consumption.

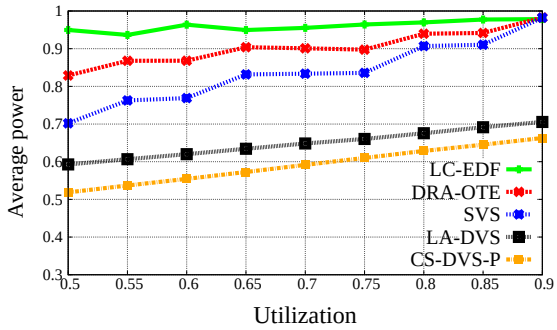
Figure 4.34 reports the average power consumption on the LPC1768 processor when $\forall \tau_i \in \Gamma : \alpha_i = 0.5$. Execution times and periods are generated as in the previous experiment, within $[B_i = C_i/10, C_i]$ and $[25, 250]$ms, respectively.

The introduction of speed-independent code improves the performance of DVFS algorithms. More precisely, with respect to the equivalent analysis in Figure 4.33(b)



(a) dsPic with periods in $[25, 250]$ms.

(b) LPC1768 with periods in $[25, 250]$ms.



(c) Pentium4 with periods in $[300, 3000]$ms.

Figure 4.33: Average power consumption considering task early terminations.

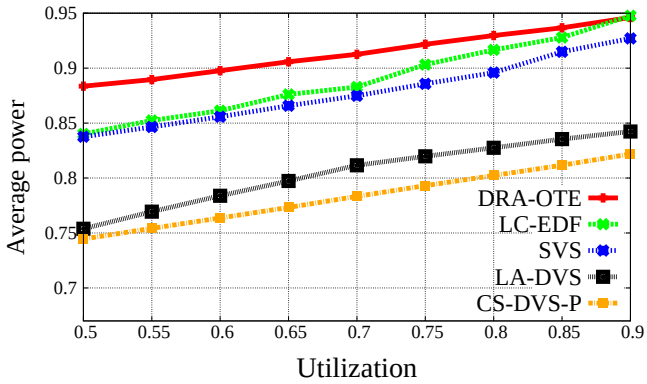


Figure 4.34: Average power consumption considering task early terminations and $\forall \tau_i$: $\alpha_i = 0.5$.

($\alpha = 0.0$), the performance of LA-DVS gets very close to that of CS-DVS-P, while SVS becomes more convenient than LC-EDF. The only DVFS algorithm that does not take a significant advantage from speed-independent sections of code is DRA-OTE.

The performance on the other two platforms shows the same trend. In general, the effectiveness of DVFS algorithms increases with the fraction of speed-independent code, as foreseen in Section 4.4.1.

Chapter 5

Energy-aware scheduling on multi-core systems

Due to the massive introduction of multi-core processors in actual computational systems, the energy issue has become more complex. On one hand, the scheduling problem is still an open issue and, on the other hand, the number of dissipation points is higher with non-trivial dependencies among them (such as voltage islands and shared memory).

The paper [BLBL13], which is detailed in Section 5.1, investigated such a problem by analyzing several well-known heuristics on homogeneous multi-core systems. A partitioned approach was considered as it requires only a polynomial complexity, even though, it may be too pessimistic for systems whose workload significantly differs from the worst-case execution. Although the work mostly considered high performance systems, results can be easily extended to multi-core embedded platforms.

5.1 Energy-aware partitioning on homogeneous multi-core platforms

This work considers the problem of partitioning and scheduling a set of real-time tasks on a realistic hardware platform consisting of a number of homogeneous processors.

Partitioned techniques statically assign each task to a specific CPU forbidding a task to migrate onto another processor even though it is idle. Such an approach allows designers to easily check the system feasibility, even though, it may be too pessimistic, leading to a waste of resources. On the other hand, global algorithms increase the system adaptability by allowing task migration at any time, but are more difficult to analyze and may introduce significant runtime overhead. Conversely, hybrid scheduling approaches mix the two techniques, aiming at reducing their drawbacks and enhancing their advantages.

The analysis takes into account two opposite heuristics: Worst-Fit Decreasing and Best-Fit Decreasing. The first approach aims at exploiting all the available processors and reducing the overall performance while the second attempts to reduce the number of active processors by optimizing execution. Despite the actual state of art, which tries to uniformly distribute the workload on the cores, the approach which minimizes the number of active cores is the most energy efficient. This result is a direct consequence

of modern hardware which is characterized by high static dissipation whose impact is around the 75% of the overall consumption.

A real platform (Dell PowerEdge R815 [DEL]), designed for high-performance computation, is taken into account in this work. An additional contribution is that we consider the power consumption of the entire system, not only the dissipation due to the processor. This is particularly relevant, as considering only a single component may give misleading results that are not valid in a general case.

In the following, Section 5.1.1 introduces the system model taken into account in the analysis; Section 5.1.2 explains how the heuristics work, while Section 5.1.3 compares them on the system under analysis.

5.1.1 System model

In accordance with the task model in Chapter 2, we consider a multi-processor platform composed of m homogeneous processors ($\phi_j,\ j = 1, \ldots, m$) whose running speed, picked up from a discrete set, can be set independently from each others.

The energy consumption model is computed according to Equation (2.6), while low-power states are assumed to be ACPI compliant. More precisely, each processor provides several low-power states characterized by different power consumption and different time overhead for entering and leaving such states. More generally, the system provides the following operative states:

- S0: The system is fully operative (both processors and memory);
- S1: Although caches are flushed and code execution is suspended, processors and memory are active;
- S2: Processors are switched off and the dirty cache is flushed to the memory. Other devices may be kept on;
- S3: Similar to S2 but more devices are put in sleep;
- S4 (hibernation): Data in memory is copied in to the hard drive and all the system is powered down;
- S5: The system is completely off except for the logic that allows the system to switch on. Putting the system in to S0 requires a complete boot and no data is retained.

Each processor can be put independently in S1, S2 and S3.

For the sake of simplicity, periods and computation times are assumed to be much longer than state transition and speed scaling overheads, which can then be discarded from the analysis.

The ACPI module is in charge of putting the processor in a predefined low-power state (statically selected in the BIOS) whenever there is no process running. Typically, the most used state is S3, as it provides a good trade-off between power consumption and time overhead for almost all the applications.

The workload Γ consists of n fully-preemptive periodic tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$ whose parameters are assumed to be in $\mathbb{N}^+$. The utilization term $U_i(s) = C_i(s)/T_i$ determines the processor bandwidth that task τ_i requires at speed s. Assuming to schedule

the periodic workload by the EDF policy [LL73], the tasks assigned to the processor ϕ_j do not miss any deadline if and only if

$$U_{\phi_j} = \sum_{\tau_i \in \Gamma | \tau_i \to \phi_j} U_i(s_i) \leq 1. \tag{5.1}$$

The overall task set is feasible if Equation 5.1 is satisfied for each processor.

Finally, we define the hyperperiod H as the time interval after which the schedule repeats itself. In the case of periodic tasks, such analysis horizon is computed as the least common multiple of periods, $H = lcm(T_1, T_2, ..., T_n)$.

5.1.2 Heuristics

Two heuristics are here considered: Worst-Fit Decreasing (WFD) and Best-Fit Decreasing (BFD).

First of all, both heuristics sort the task set by descending utilization: $\bar{\Gamma} = \{\tau_i \in \Gamma | u_{i-1} \geq u_i \geq u_{i+1}\}$. Then, starting from the first element in $\bar{\Gamma}$, WFD assigns each task to the processor with the highest unused utilization, while BFD chooses the one whose spare utilization fits better.

Let us consider three processors and five tasks with the following utilization values: $u_1 = 0.6$, $u_2 = 0.5$, $u_3 = 0.3$, $u_4 = 0.3$ and $u_5 = 0.1$. The WFD heuristic would start assigning each of the first three tasks to one free processor, τ_1 to ϕ_1, τ_2 to ϕ_2, and τ_3 to ϕ_3. Then, the spare utilization on each processors become 0.4, 0.5 and 0.7, respectively. Next, the fourth task is assigned to ϕ_3 which has the highest spare capacity, reducing it down to 0.4. Finally, τ_5, characterized by the lowest utilization, is assigned to ϕ_2. The final partitioning is shown in Figure 5.1.

Conversely, the BFD heuristic assigns τ_1 to ϕ_1 and then tries to allocate τ_2 to ϕ_1, but the residual utilization of ϕ_1 is not enough to accommodate τ_2, which is then allocated to ϕ_2. Unlike the previous case, τ_3 is allocated on ϕ_1 and all the remaining two tasks are assigned to the second core ϕ_2. The BFD result is reported in Figure 5.2.

In conclusion, WFD led to a task partitioning that left 0.4, 0.3 and 0.4 as spare capacity on the three cores, while BFD utilized entirely the second core and partially the first (90%), leaving the third processor off.

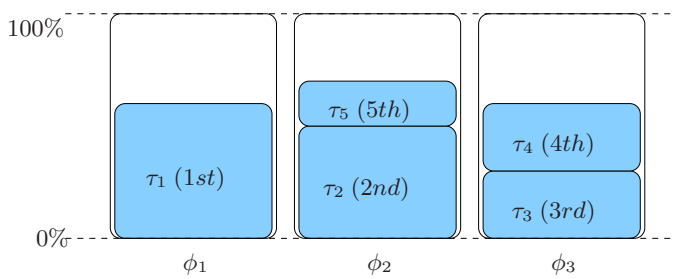


Figure 5.1: Task partitioning obtained by the WFD heuristic.

As this simple example has shown, BFD aims at reducing the number of active cores, while WFD attempts to exploit all processors to reduce the overall working performance.

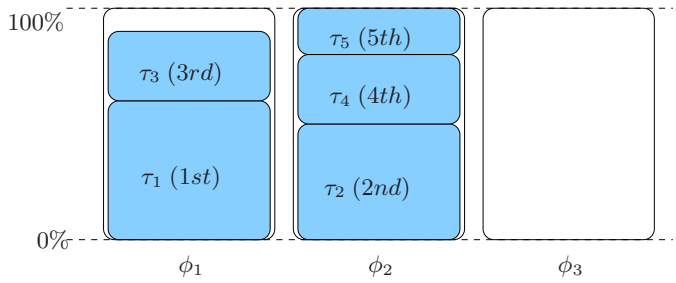


Figure 5.2: Task partitioning obtained by the BFD heuristic.

Once tasks have been partitioned, the remaining utilization on each core can be exploited to further reduce energy consumption. In this paper, we consider two extreme approaches:

- DVFS: when the system starts, for each core, the slowest speed that guarantees the task set feasibility is set;
- DPM: the workload is executed at the maximum performance and then ACPI exploits low-power states when there are no pending tasks.

Since we deal with a discrete set of frequencies, if DVFS is not able to exploit the whole residual utilization, DPM is used in addition to take advantage of it.

5.1.3 Experimental results

In this section, the power measurements related to the multi-core platform is first reported to provide the consumption profile. Then, the two heuristics introduced in Section 5.1.2 are compared in terms of energy consumption.

Consumption profile

The considered platform, a Dell PowerEdge R815 rack server, is equipped with 48 homogeneous cores supporting the following frequency range $\{0.8, 1.0, 1.3, 1.5, 1.9\}$GHz which leads to the speed set: $\{0.42, 0.53, 0.68, 0.79, 1.0\}$. Each core can set its frequency independently of the others. Cores are divided in 8 clusters, each containing 6 cores. The platform runs the GNU/Linux kernel 3.10.0-rc3 [LIN].

The power measurements reported in the paper have been obtained by monitoring the absorbed power from the entire platform, including memory, I/O peripherals, and buses.

In the considered scenario (48 cores, 5 frequencies each and two low-power states), the configurations to be checked are 7^{48}. Since the number is extremely high, we measured the consumption for a set of key configurations and then the others are obtained by interpolation. More precisely, for each speed, we measured the consumption of setting all the active cores to the speed under analysis while varying the number of fully loaded cores (from 1 to 48). The workload consists of an endless execution of the EEMBC Coremark benchmark [COR], which implements mathematical operations (CPU-intensive code) and keeps the processor always busy (no I/O phases). As a result, the execution time perfectly scales with the frequency. Concerning the unused

processors, the two low-power states used by the ACPI driver have been considered. The first one is called IDLE and it is automatically inserted by the module when there are no pending tasks, while the OFF state is manually set by the user.

The first consideration related to the measurements reported in Figure 5.3 concerns the impact of the low-power state. More precisely, the OFF feature was supposed to guarantee the lowest consumption but it exploits S1 and introduces a significant overhead for updating the kernel data structures. On the other hand, the IDLE state exploits S3, guaranteeing lower consumption and shorter overhead.

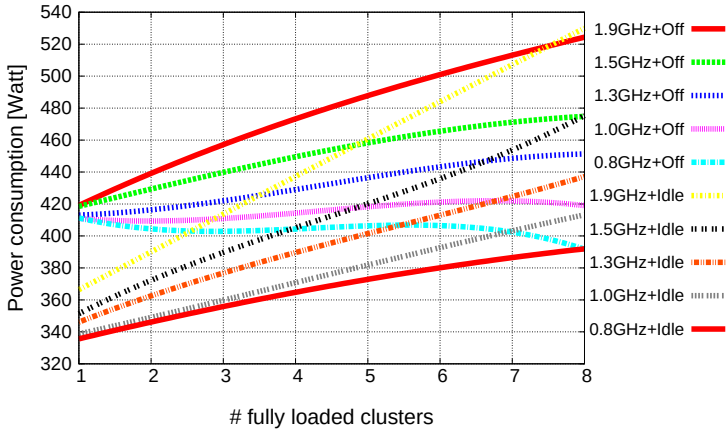


Figure 5.3: Power consumption as a function of active clusters and low-power states.

Given such measurements, the configurations in which all the processors exploit the same frequency but only some of the cores in the cluster are active (e.g., 3 processors out of 6) are assumed to dissipate the interpolated value. We empirically tested such an assumption for several configuration and it turned out to be an acceptable approximation. When the processors exploit different frequencies, we consider the average consumption as the weighted average of the consumption assuming all the active processors running at the same frequency. Formally, it can be expressed as:

$$P_{CPU}(t) = \sum_{s_j} w_j \bar{f}(s_j),$$

where w_j is the actual number of CPU at speed s_j divided by the total number of active cores and $\bar{f}(s_j)$ represents the consumption obtained by setting all the active cores at speed s_j. For example, if there are two clusters at frequency $1.0GHz$ and one at $1.5GHz$, the consumption would be:

$$P_{CPU}(t) = \frac{12}{18}360 + \frac{6}{18}375 = 365W,$$

where 360 and 375 are the consumptions of having three clusters at 1.0 and 1.5 GHz, respectively.

It is worth noting that when the system keeps active only a single cluster, the power consumption can be varied from 335 to 365 (from 0.8GHz to 1.9GHz, respectively), meaning that speed scaling techniques can affect less than the 10%. Even when the spread between minimum and maximum consumption is the widest (all the processors are awake), the dynamic dissipation is only around 25%. In other words, the static dissipation is ascribed to dissipate at least the 75% of the overall power.

Performance evaluation

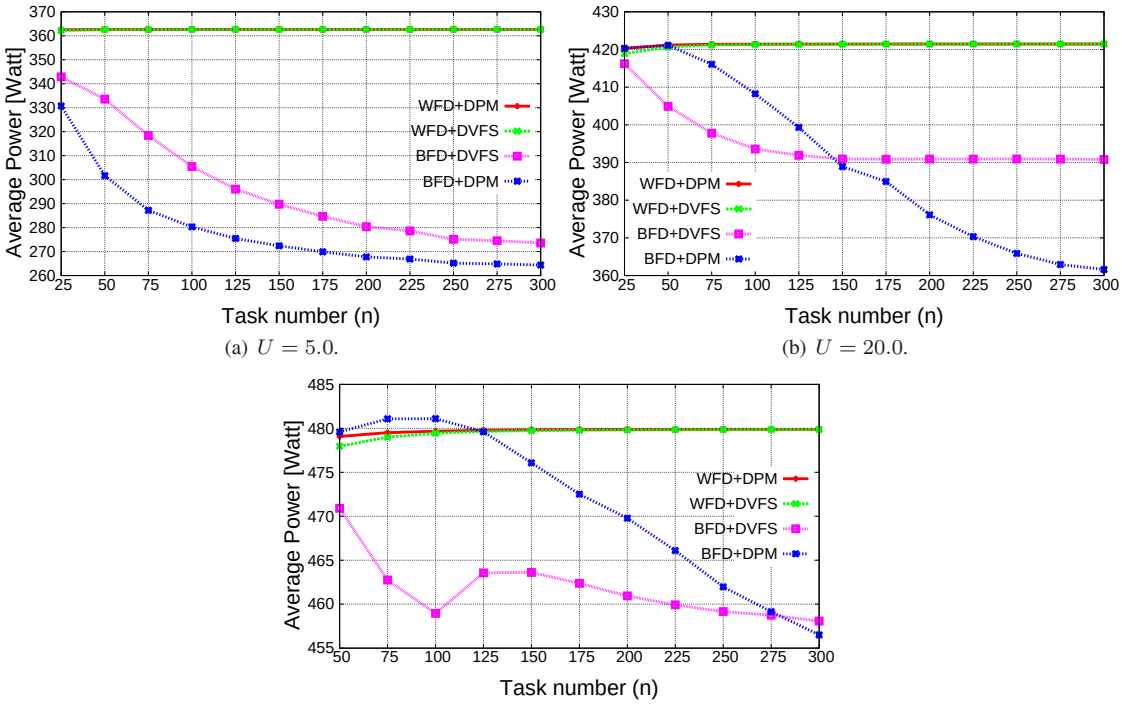
The following analysis compares the performance of the partitioning heuristics in terms of average power dissipation for different task set parameters. The workload has been generated using the UUniFast [BB05] algorithm. Once the heuristics are executed, the average power consumption is computed analytically by assuming that all jobs are released at the beginning of the hyperperiod, meaning that each processor ϕ_j is active for $U_j \cdot H$ while it is off until next hyperperiod. This assumption is pessimistic because it leads to having the maximum number of active cores $(\forall \phi_j : U_j > 0)$ for the longest time $\left(\min_{\phi_j} U_j\right)$.

The first experiment considers the average power consumption obtained from the heuristics varying the number of tasks (from 25 to 300 with step 25) at three different utilizations: low ($U = 5.0$), medium ($U = 20$) and high ($U = 35.0$). The results are shown in Figure 5.4(a), Figure 5.4(b) and Figure 5.4(c), respectively. Note that, using the EDF policy, the utilization upper bound is equal to the number of processors (in our case, 48), but it is not easily achievable due to fragmentation.

The first significant result consists of showing the high average consumption obtained from WFD with respect to BFD. Such a result is mainly ascribable to the power model of the board, as it privileges approaches that switch off as many processor as possible rather than reducing their performance. Indeed, all the previous work assumed a cubic power function $\left(p(s) = \beta \cdot s^3\right)$, which heavily reduces consumption when the speed is scaled down. Note that performance is not affected by the number of tasks, as WFD is slightly affected by fragmentation. Finally, note that applying either DPM or DVFS techniques produces the same result.

As already discussed, BFD fits well with this kind of architecture as it aims at compacting the workload on few cores and putting the others in low-power state. This approach drastically reduces the static dissipation, which accounts for at least 75% of the overall dissipation. According to the presented results, the higher the number of tasks, the lower the average power consumption. This can be easily explained by considering that in a larger task set tasks have smaller utilization, hence the heuristic can better compact them, reducing fragmentation. Concerning the strategies applied after task partitioning, the DVFS approach is more effective up to a certain point, after which DPM becomes more convenient. Such a turning point depends on the overall task utilization and the number of tasks. Basically, for a given utilization, small task sets create more significant fragmentation and such a spare capacity is better exploited by slowing the speed down. On the other hand, when the number of tasks increases, fragmentation reduces and the smaller slack time is optimized by the DPM approach. Generally, we can state that the higher the utilization, the higher the number of tasks that make the DPM strategy more effective. In the first case (Figure 5.4(a)), the utilization is low ($U = 5.0$) and DPM is already more suitable. When the utilization is $U = 20.0$ (Figure 5.4(b)), the turning point is at $n = 150$, while it reaches $n = 275$ for $U = 35.0$ (Figure 5.4(c)).

For the sake of completeness, Figure 5.5 presents a different reading key, showing the average power consumption with 150 tasks for different utilizations ($U \in [5.0, 40.0]$ with step 2.5). As previously highlighted, the turning point is at $U = 20.0$ with 150 tasks. Moreover, Figure 5.5 shows what happens when computational times do not entirely scale with the frequency. More precisely, the higher the value of α, the lower the average power consumption obtained by the DVFS approach. As already highlighted by Bambagini et al. [BBMB13a], this is due to the fact that scaling speed



(a) $U = 5.0$.

(b) $U = 20.0$.

(c) $U = 35.0$.

Figure 5.4: Average power consumption vs. task number with $U \in \{5.0, 20.0, 35.0\}$.

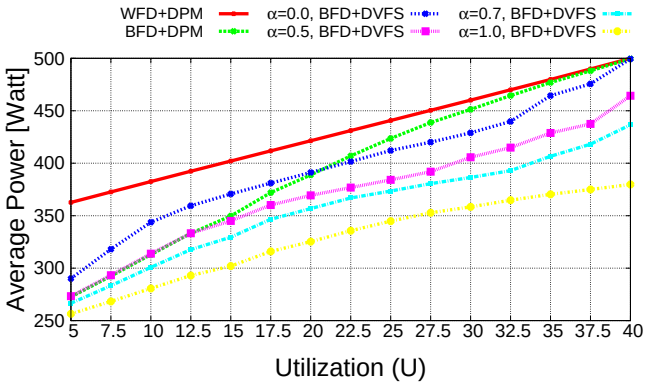


Figure 5.5: Average power vs. U with $n = 150$ and different α.

with high α is equivalent to schedule a task set with lower utilization than the original one, meaning that lower speeds can be effectively exploited while no deadline is missed.

Chapter 6

Energy-aware scheduling in distributed systems

Nowadays, many embedded systems are interconnected together via wired or wireless links, in order to exchange data and pursuit a global target, which can not be guaranteed from only a single computational unit.

The problem of mapping a real-time workload on a distributed system has been studied in [PBB+12] and here reported in Section 6.1. Besides the energy issue, the problem formulation is enriched with bandwidth constraints and redundancy requirements (which is a way to increase the system fault tolerance).

6.1 Energy and bandwidth-aware co-allocation

The energy consumption in distributed systems depends on several inter-related factors, including task partitioning, process redundancy, fault tolerance, task and message scheduling, and communication bandwidth allocation. Although some of these issues have been considered in the literature in isolation, a systematic approach considering all the constraints is still missing.

The presented work addresses the problem of allocating a task set and the required communication bandwidth on a distributed embedded system, aiming at reducing energy consumption while guaranteeing timing and redundancy constraints. While redundancy improves reliability and precision, it increases the energy consumption due to the additional workload and traffic. The problem is complicated by the fact that there are situations in which reducing the energy consumption on a node may increase the energy dissipated in other nodes, so shortening the lifetime of the entire system.

More specifically, the problem consists of co-scheduling tasks and messages in a feasible way with the objective of minimizing the overall energy consumption and guaranteeing a minimum number of task copies and system lifetime. Two heuristic approaches are proposed and compared against a complete method and simulated annealing. Simulation results show the effectiveness of the proposed approaches.

Section 6.1.1 presents the system model, in terms of processing nodes, communication bandwidth, power model, computational and communication workload. The problem statement is introduced in Section 6.1.2, including the specification of all the system constraints and the performance metric used to evaluate the goodness of an allocation. Section 6.1.3 illustrates the proposed approaches, whereas Section 6.1.4 reports

the experimental results.

6.1.1 System model

This work considers a distributed system of *m* homogeneous nodes Ψ_1, Ψ_2 ..., Ψ_m logically connected as illustrated in Figure 6.1. Each node performs sensory acquisition and sends messages to a *coordinator* node (Ψ_0), according to a star topology, using a TDMA-based communication protocol that guarantees a bounded transmission delay. In each TDMA time wheel, each node Ψ_i is assigned a bandwidth slot of length w_i. Note that, the *coordinator* Ψ_0 is not subject to the analysis carried out in this work, since it is assumed to be always on and available to the nodes.

Since the system is intended as a network of nodes, throughout the rest of the paper, the terms *system* and *network* are used interchangeably.

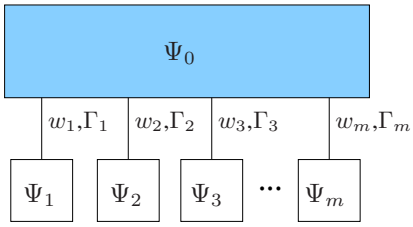


Figure 6.1: A sample network with a node coordinator.

The system is assumed to run an application Γ consisting of a set of *n* sporadic real-time tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ with implicit deadlines. Each job $\tau_{j,k}$ sends a message of payload M_j to node Ψ_0 at its termination, which can occur any time within the interval between its arrival $a_{j,k}$ and its absolute deadline: $(a_{j,k}, a_{j,k} + D_j]$. The overhead of transferring a message generated by a task into the transmission buffer is already accounted in its worst-case execution time. The utilization of task τ_j is denoted by u_j and is computed as C_j/T_j.

Each node Ψ_i hosts a subset Γ_i of Γ. The utilization of node Ψ_i and the system utilization are denoted as U_i and U_{tot}, respectively. Note that a task τ_j can run in more than one node, but each node can execute at most one copy of τ_j. The number of running instances of task τ_j on the entire network is denoted as μ_j.

For each task τ_j, the application specifies a *reward function* denoted as γ_j, which grows with the number of task instances and, hence, measures the satisfaction of the task redundancy across the system.

Tasks in each node are scheduled by the Earliest Deadline First (EDF) [LL73] algorithm, but the analysis can easily be extended to different scheduling policies.

The timing parameters introduced above are summarized in Figure 6.2.

Bandwidth model

The communication between the nodes and the *coordinator* is managed by a TDMA-based communication protocol, according to a star topology. Each node Ψ_i is assigned a time slot w_i in each TDMA wheel where it can transmit its messages. The messages produced by the tasks running in node Ψ_i are enqueued in a transmission buffer of length Θ_i adopting a FIFO policy and then sent as soon as the bandwidth becomes available.

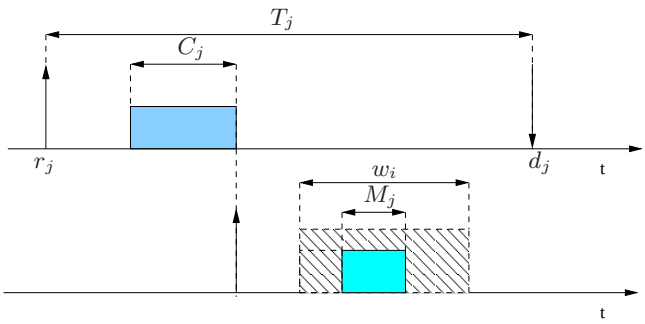


Figure 6.2: Timing parameters.

A necessary condition to guarantee that each node can transmit its messages without overflowing the transmission buffer requires the wheel length W to be no shorter than the minimum period in the task set, hence W is set as

$$W = \min_{j \in [1..n]} T_j. \tag{6.1}$$

Each task τ_j of node Ψ_i is assigned a time budget Q_j sufficient to ensure the correct delivery of the related message within $2T_j$ from the associated job release time $r_{j,k}$. Then, w_i is set as the sum of the budgets Q_j's of the tasks allocated to node Ψ_i:

$$w_i = \sum_{\tau_j \in \Gamma_i} Q_j. \tag{6.2}$$

The task budget Q_j is computed as follows:

$$Q_j = \frac{M_j}{\left\lfloor \frac{T_j}{W} \right\rfloor}. \tag{6.3}$$

Observe that $\left\lfloor \frac{T_j}{W} \right\rfloor$ represents the number of complete TDMA wheels available during a task period T_j, i.e., the number of time slots that the node running τ_j can exploit to transmit the message generated by such a task. Since M_j is the length of any message generated by τ_j, the rationale behind the computation of Q_j is to assign a time budget sufficient to send a message of length M_j every T_j time units.

If w^{com} denotes the sum of the slots w_i assigned to the nodes, the overall communication load on the network is w^{com}/W. Since for each node Ψ_i, the time slot w_i is computed in such a way the node can send all messages generated by its tasks, it follows that a necessary and sufficient condition to guarantee all message deadlines is:

$$w^{com} \leq W. \tag{6.4}$$

To save energy, the adopted bandwidth scheme reserves a slot $S = W - w^{com}$ over each W in which the communication is not allowed and all the nodes turn their transceiver off.

Consider a sample scenario featuring three nodes (Ψ_1,Ψ_2,Ψ_3) and a task set composed by two tasks (τ_1, τ_2), where node Ψ_1 hosts both the tasks, node Ψ_2 hosts τ_1, and node Ψ_3 hosts τ_2. The resulting TDMA wheel partition is illustrated in Figure 6.3. Note that, throughout the paper, the slot lengths w_i are ordered by the node indexes inside the wheel, while the budgets Q_j inside the slot are ordered by the task indexes.

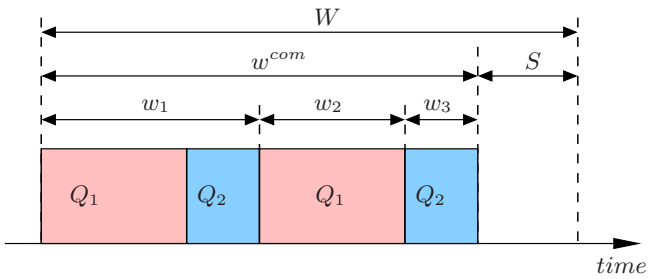


Figure 6.3: Bandwidth allocation example.

Power Model

Since this work focuses on a network-wide approach for reducing energy consumption, a simplified power model is adopted for the nodes to estimate their power consumption. A finer model would be unnecessarily detailed in this context, considering that each node could apply further runtime strategies to save energy.

In particular, each device (i.e., CPU or transceiver) is only assumed to be in either *active* or *sleep*. In the active state, the power consumed is denoted by P_a^{CPU} and P_a^{com}, respectively. In the other state, the dissipation is P_s^{CPU} and P_s^{com}, respectively.

The overhead to switch between power states for both the devices is assumed to be negligible, both in terms of time and energy.

The node power consumption is estimated as follows:

$$\begin{aligned} P_i &= U_i P_a^{CPU} + (1 - U_i) P_s^{CPU} + \\ &\quad \frac{w_i}{W} P_a^{com} + \left(1 - \frac{w_i}{W}\right) P_s^{com}. \end{aligned} \tag{6.5}$$

The node and system mean power consumption is estimated assuming that each node exploits the idle intervals in task and message schedules turning the CPU and the transceiver off during idle periods and outside bandwidth slots, respectively, as follows:

$$\begin{aligned} P &= U_{tot} P_a^{CPU} + (m - U_{tot}) P_s^{CPU} + \\ &\quad \frac{w^{com}}{W} P_a^{com} + \left(m - \frac{w^{com}}{W}\right) P_s^{com}. \end{aligned} \tag{6.6}$$

The current energy level of node Ψ_i is denoted by E_i and every node is provided with the same initial energy $E^{(0)}$.

Lifetime model

The system lifetime is a key element for evaluating sensor networks. Its definition is not univocal because it depends from a lot of application parameters (e.g., number of sensors, network coverage, quality of service, etc) and its value is an aggregation of all nodes statuses. The network can only fulfill its purpose as long as it is considered *alive*, but not after that. Therefore, the lifetime is an indicator for the maximum utility a sensor network can provide.

There are a lot of lifetime definitions [DD09], but the most common and most frequently used in the literature is the m-of-m lifetime. In this definition, the distributed

system lifetime L ends as soon as the first node fails, thus $L = \min_i L_i$, with L_i representing the lifetime of node Ψ_i computed as E_i/P_i.

A common constraint on the lifetime imposes a minimum amount of time ($\overline{L}$) in which the system has to stay alive. The minimum lifetime could be formalized as:

$$\min_i \frac{E_i}{P_i} > \overline{L}.$$

Allocation specific parameters

An *allocation* describes the distribution of the tasks instances among the nodes of the system. Since a task τ_j can run on different nodes simultaneously, an allocation matrix $X \in \{0,1\}^{m \times n}$ is defined to keep track of such a distribution on the network. The generic element x_{ij} of the allocation matrix X is a boolean variable indicating whether task τ_j is present on node Ψ_i or not. Note that each node can execute at most one copy of τ_j. In the following, a set of previously defined parameters are formalized based on X.

The subset Γ_i of tasks running on node Ψ_i is expressed as

$$\Gamma_i = \{\tau_j | x_{ij} = 1\}, \tag{6.7}$$

while the length w_i of the communication bandwidth slot assigned to the node Ψ_i is computed as

$$w_i = \sum_{j=1}^{n} x_{ij} Q_j. \tag{6.8}$$

The total utilization U_i of node Ψ_i and the system utilization U_{tot} are formally defined as $U_i = \sum_{j=1}^{n} x_{ij} u_j$ and $U_{tot} = \sum_{i=1}^{m} U_i$.

The actual number μ_j of instances of task τ_j across the system is computed as

$$\mu_j = \sum_{i=1}^{m} x_{ij}. \tag{6.9}$$

The worst case scenario for the transceiver buffer occurs when two instances of the same task generate a message between the end of the previous Q_j and the start of the next one, as depicted in Figure 6.4. In the example illustrated in the figure, the Q_j is supposed to occur at the beginning of each wheel. More than two generations are not possible because the wheel is equal to the minimum task period. This leads to consider that the minimum required size Θ_i^{min} of the transceiver internal buffer on node Ψ_i is computed as:

$$\Theta_i^{min} = 2 \sum_{j=1}^{m} x_{ij} M_j.$$

6.1.2 Problem statement

This paper addresses the problem of allocating a real-time task set over a distributed system composed by homogeneous embedded nodes. Such an allocation has to meet a set of constraints:

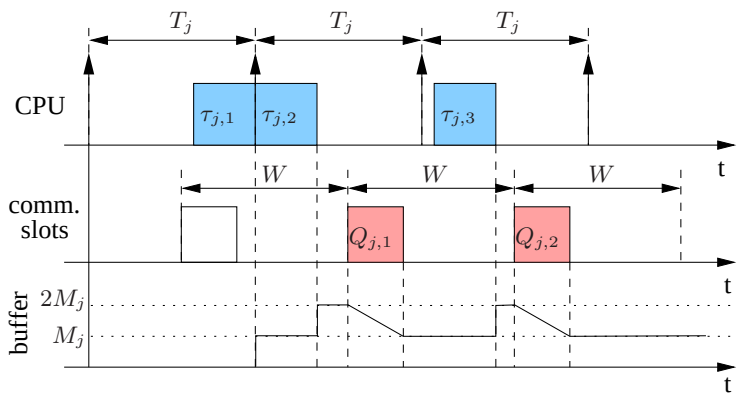


Figure 6.4: Example of worst-case buffer use.

Task schedulability All the tasks running in each node have to terminate within their deadlines. Under the assumption that all the tasks have relative deadlines equal to periods and are scheduled by EDF, such a constraint is expressed as follows:

$$\forall i \in [1, m] \quad U_i \leq 1; \tag{6.10}$$

Bandwidth To guarantee the schedulability of the messages produced by the tasks, the sum w^{com} of all the slots assigned to the nodes must not exceed the wheel W:

$$w^{com} \leq W; \tag{6.11}$$

Buffer For each node, the transmission buffer (whose length Θ is equal for each node) must be long enough to contain all the messages generated in the worst case:

$$\forall i \in [1, m] \quad \Theta \geq \Theta_i^{min}; \tag{6.12}$$

Redundancy To improve the accuracy level of measurements, a minimum number μ_j^{min} of running instances for each task τ_j must be guaranteed to be executed in the system, that is:

$$\forall j \in [1, n] \quad \mu_j \geq \mu_j^{min}. \tag{6.13}$$

Lifetime To guarantee a desired system lifetime $\overline{L}$, the initial energy $E^{(0)}$ available in each node must be sufficient to keep the entire network alive for the required duration, that is:

$$\forall i \in [1, m] \quad \frac{E^{(0)}}{P_i} \geq \overline{L}. \tag{6.14}$$

Performance evaluation

To evaluate the performance of a generic allocation, three distinct normalized indexes ($\in [0, 1]$) are introduced to measure redundancy, energy saving, and uniformity. Such factors are combined to obtain a normalized scalar function able to compare the goodness of different allocations.

The performance function is modeled to operate also in the case of unfeasible allocations. Such a function is built so the co-domain range of the performance function is $[-3, 1]$, equally distributed between the feasible situations ($[0, 1]$) and the unfeasible ones ($[-3, 0)$).

As already explained in Section 6.1.1, more instances of each task can execute on the network. Note that an implicit limit on the running instances of a task τ_j exists as at most *m* nodes can run a single instance of τ_j.

For each task τ_j, a monotonic function γ_j is provided by the application designer to specify the reward according to the actual number of task instances running in the system, from μ_j^{min} to m. γ_j is a function with bounded output value in $[0, 1)$. The redundancy index ρ is defined as

$$\rho \triangleq \frac{\sum_{j=1}^{n} \gamma_j}{n}.$$

The overall energy consumption of the system is a key parameter taken into account by the analysis carried out in this work. The index ξ measures the energy saving of an allocation with respect to the highest power consumption P_{max} that the system can experience, that is:

$$\xi \triangleq \frac{\max(0, P_{max} - P)}{P_{max}}.$$

Notice that P_{max} refers to an ideal allocation in which all the nodes are fully loaded (i.e. all nodes have utilization equal to 1) and the bandwidth slots are assigned in such a way that w^{com} is equal to W.

Based on the definition of system lifetime assumed in this work, how task instances are spread across the system nodes is of key importance, as a more uniform load distribution results in a longer global lifetime (due to the proportionality between utilization and power consumption at node level).

The utilization uniformity of the allocation is evaluated through the variance σ^2 of the total utilization U_i of each node Ψ_i computed as follows:

$$\sigma^2 \triangleq \frac{1}{m} \sum_{i=1}^{m} \left(U_i - \frac{U_{tot}}{m} \right)^2.$$

As for the previous indexes, the uniformity of the allocation is evaluated through an index α normalized in the interval $[0, 1]$:

$$\alpha \triangleq \frac{\max(0, \sigma_{max}^2 - \sigma^2)}{\sigma_{max}^2},$$

where σ_{max}^2 denotes the variance in the worst-case scenario in which half nodes are fully loaded and half are empty, leading to a value of σ^2 equal to 0.25.

Considering the trade-off between redundancy and power consumption, a parameter $\eta \in [0, 1]$ is introduced to balance the two contributions. Notice that such a parameter remains constant throughout all the optimization process.

For a feasible allocation, the performance function Φ is defined as follows:

$$\Phi_\eta^+(X) = \alpha[\eta\rho + (1 - \eta)\xi], \tag{6.15}$$

whereas, for an unfeasible allocation, it is defined as follows:

$$\begin{aligned}\Phi^{-}(X) &= -\frac{1}{n}\sum_{j}^{n}\frac{\max(0,\mu_j^{min}-\mu_j)}{\mu_j^{min}}+\\ &\quad -\frac{\sum_i^m \max(0,U_i-1)}{(U_{tot}-1)m}+\\ &\quad -\frac{\max(0,w^{com}-W)}{m\sum_j^n Q_j}.\end{aligned} \tag{6.16}$$

Such a function is composed by three terms, each introduced to weigh the violation of one of the evaluated constraints. Note that while the second term is defined at a node level, since it concerns CPU overrun, the last term operates at a system level, as the communication wheel is shared among all the nodes.

6.1.3 Proposed algorithms

This section presents three types of algorithms that try to maximize a user-defined performance function under the constraints formalized in Section 6.1.2.

The cost of each approach is measured through the number of performance evaluations, rather than through the total processing time, since the execution time of an algorithm is affected by several factors (e.g., processor speed, memory, and implementation efficiency). The complexity to evaluate a specific allocation is $O(nm)$.

Complete search

A branch and bound search is provided to evaluate the distance of the optimal solution with respect to the other methods, at least for a small system size. The exhaustive search uses an EDF feasibility test on single nodes to prune unfeasible branches.

Note that, at low total utilizations, the number of feasible (single node) allocations is exponential on the number of tasks, since very low tasks utilizations make any task combination feasible, and drops down at higher utilizations.

Since the embedded nodes composing the system are homogeneous, several solutions are symmetric, thus leading to the same performance. To avoid the generation of symmetric allocations, this approach combines the solutions previously found to reduce the number of possible configurations.

Heuristics

This section presents two heuristic approaches, *Heuristic A* and *Heuristic B*.

Heuristic A starts from an empty allocation and, at each step, generates all the possible configurations that differ from the current one by a single task (by switching a single element of matrix X from 0 to 1). Then, the configuration with the highest performance is selected. In the case of multiple configurations featuring the highest performance, the first occurrence is selected. The algorithm stops as soon as it fails to improve the current best performance.

Since at each step the approach performs at most $n \times m$ evaluations, its worst-case complexity is $O(n^3m^3)$, as it is possible to insert at most $n \times m$ instances to fill the X matrix.

Heuristic B is a variation of Heuristic A, since at each step it selects the node with the lowest utilization, testing the resulting performance at each addition. Since at each

step the approach has to evaluate n different configurations, the worst-case complexity is $O(n^3m^2)$. Like Heuristic A, the approach stops as soon as it fails to improve the current best performance.

Simulated annealing

Simulated Annealing [KGV83] is an effective technique for finding an acceptably good solution in a fixed amount of time, even in large search spaces. In this approach, a generic point s of the search space is called a $state$, and the neighborhood of a state s is defined as the set of states produced from s by suitable alterations.

In this implementation, a state is represented by a global allocation, and its neighborhood is the set of allocations that differ from the starting one only for one boolean element of the matrix X. This means that the neighborhood of a state is another allocation in which a single task is added (if it was not present) or removed (if it was present). The other parameters, that is the cooling factor, the maximum number of tries at fixed temperature, and the temperature threshold, are selected to improve performance.

In this work, the simulated annealing has been implemented according to two approaches, which differ for their initial state. The first approach starts from the empty allocation matrix and it is referred to as *Basic Simulated Annealing* (BSA), while the second one lies upon the result of Heuristic A and, hence, it is referred to as *Heuristic Simulated Annealing* (HSA).

6.1.4 Experimental results

This section presents a set of experimental results of the approaches proposed in Section 6.1.3. Such results are obtained by simulation on synthetic workloads adopting the power profile taken by the Microchip dsPIC[1] datasheet, interpolating the typical consumptions.

An execution scenario is characterized by the tuple (n, m, U, B), where n denotes the number of tasks, m the number of nodes in the network, U the total task set utilization, and B the total communication bandwidth required by the task set ($B = \sum_{j=1}^{n} M_j/T_j$).

Given a total utilization factor (U and B), the task set features (C_j, T_j and M_j) are computed according to a uniform distribution [BB05]. From such values, the wheel length W and task bandwidth slots (Q_j) are computed according to Equation (6.1) and Equation (6.3).

As already exposed in Section 6.1.2, the application designer has to specify the reward function γ_j for each task τ_j. Let μ_j^{sat} denote the number of running instances of τ_j beyond which the measures have no gain on accuracy, then:

$$\gamma_j = \left(1 - e^{\Delta_j}\right),$$

where

$$\Delta_j = \frac{-5(\mu_j - \mu_j^{min})}{\mu_j^{sat} - \mu_j^{min}}.$$

For all the experiments, μ_j^{sat} is set equal to the number of nodes m. The balancing parameter η between redundancy and energy saving is chosen to be 0.5.

[1] dsPIC33FJ256MC710 microcontroller

Due to the complexity of considering all the parameters introduced in the analysis, the set of experiments described in this section do not check the violation of the constraints on the minimum buffer and the minimum lifetime formalized in Equation (6.12) and Equation (6.14), respectively.

The first experiment evaluates the complexity of the algorithms as a function of the product $n \times m$, for all the approaches, setting $U = m/2$ and $B = 0.3$. Table 6.1 reports the number of performance evaluations averaged on 50 runs of each approach. The Complete Search (CS) approach represents an effective method to find the optimal solution only for small problem size, as the number of evaluations becomes too high already at $n \times m = 10 \times 6$. The number of evaluations performed by each approach reflects the complexity values reported in Section 6.1.3. Simulated annealing techniques have a higher number of evaluations since they explore the search space until the system is cooled but with a conditional stop criterion on the stability of the result. The performance difference between CS and the other approaches is under 5%.

$n \times m$	CS	Heu A	Heu B	BSA	HSA
5x4	519	161	40	14684	20603
10x6	$4.8 \cdot 10^9$	1186	168	22561	50234
15x8	-	4036	432	43598	63678
20x8	-	7139	787	57663	79626
25x10	-	15101	1309	91628	126059

Table 6.1: Problem complexity.

The second experiment evaluates the performance index Φ as a function of the task set utilization U, and results are shown in Figure 6.5. All the approaches have been tested with $n = 25$, $m = 10$ and $B = 0.3$. The utilization is normalized on the number of nodes m.

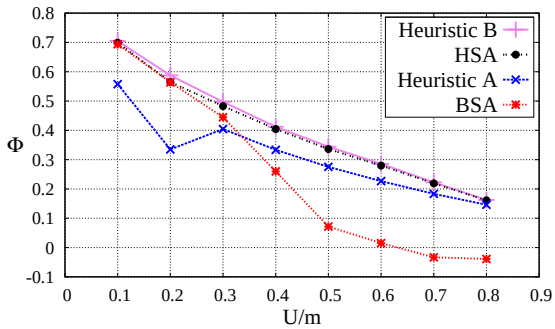


Figure 6.5: Performance analysis of the approaches.

As Figure 6.5 shows, Heuristic B outperforms all the approaches. BSA performs poorly at utilizations higher than 0.5. Heuristic A is often able to find a reasonable solution, although it can be improved by HSA, whose performance is very similar to that of Heuristic B.

The third experiment deeply investigates Heuristic B, analyzing the three components of the performance index as a function of the utilization U, normalized on the number of nodes m.

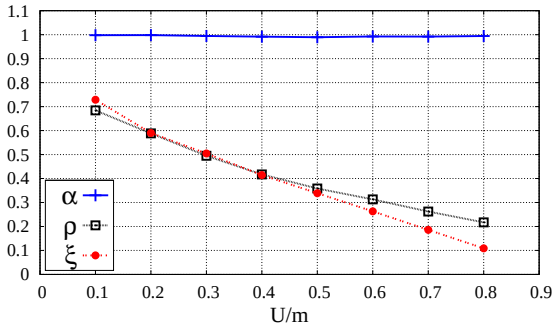


Figure 6.6: Performance components.

The results are reported in Figure 6.6, which clearly shows that the α factor is about constant, revealing that the solutions found by Heuristic B are characterized by uniform allocations. As expected, both the redundancy and energy saving indexes decrease with the growing of the normalized CPU load. This occurs because tasks with higher utilization are difficult to spread across the network without violating the schedulability constraint and causing a sensible increase of power consumption.

The fourth and last experiment analyzes Heuristic B as a function of both n and m, for a bandwidth utilization $B = 0.3$. The values of n range in $\{5, 10, 15, 20, 25\}$, while the values of m range in $\{4, 6, 8, 10, 12\}$. In this experiment, the utilization U is always set to $m/2$.

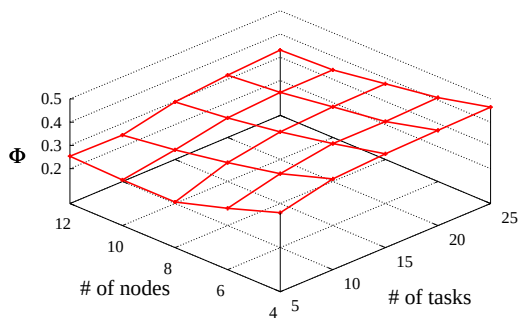


Figure 6.7: Performance analysis of Heuristic B when B=0.3.

As reported in Figure 6.7, the performance obtained by Heuristic B improves as the number of tasks increases. This is due to the higher fragmentation of the task set, that results in a wider feasible solution space. Instead, the performance index decreases as the number of nodes grows. Since the utilization U is always set to the half of m, the tasks result in a higher utilization. As for the third experiment, the approach is able to spread only a few tasks instances across the network paying a higher power consumption.

Chapter 7

Energy-aware scheduling with renewable energy

Energy harvesting systems are gaining increasing importance in the embedded systems domain, as they provide an effective solution to bridge the gap between the energy supply and demand by using solar panels and piezoelectric units.

With energy harvesting, in theory it becomes possible to design and build *energy-neutral* systems [HZK$^+$06]: systems that manage their energy consumption activities in such a way that they can perpetually sustain their operation, subject to the hardware faults/longevity only. In an energy-neutral system, over any time interval $[0, t]$, the consumed energy does not exceed the available energy, which is the harvested energy augmented by the initial energy reserves.

Researchers have recently started to investigate the impact of adding energy harvesting dimension to the existing frameworks. In general, energy-harvesting algorithms apply *task procrastination* as the conditions warrant: for instance, due to the current low energy level, or as a way to proactively prevent a *future* energy shortage. Based on this distinction, we can broadly divide the existing algorithms into *energy-greedy* and *computation-greedy* classes depending on the actions they take when the energy level is low. Under that condition, *energy-greedy* algorithms exploit the available slack in the system by procrastinating tasks and charging the battery as much as possible. In contrast, the *computation-greedy* algorithms give priority to execute the pending workload, and charge the battery only for the shortest time which guarantees the execution of the next computational unit.

For fixed-priority real-time embedded systems which are more common in practice, the two well-known algorithms are PFP_{st} (also called $EDeg$) [CMM11] and PFP_{asap} [ACM13a], that represent energy-greedy and computation-greedy algorithms, respectively. In particular, PFP_{asap} is shown to be optimal: any task set that can be feasibly scheduled by any other fixed-priority energy-harvesting scheduling algorithm can be also scheduled by PFP_{asap}. On the other hand, the same paper shows that in terms of the preemption numbers and other run-time overhead metrics, PFP_{st} has a clear advantage, while its average feasibility performance lags behind PFP_{asap} by a small margin.

Our work is partly inspired by the observation that, despite their theoretical importance, the existing algorithms assume power models that do not fully comply with existing processors: for instance, it is assumed that the CPU can be switched to/from a low-

power (sleep) state instantaneously and without any energy overhead. Moreover, there is only one sleep state and its power consumption is negligible. Contemporary processors have typically multiple low-power states, each with different power/transition overhead characteristics, such as *standby, idle,* and *deep sleep* states. Another objective is to develop a simple, proactive and highly-predictable framework that seamlessly integrate the energy-harvesting capabilities into existing and widely known fixed-priority systems.

This work proposes a novel and highly-predictable energy management algorithm for fixed-priority real-time systems with renewable energy, called *Periodic Charging Scheme (PCS)*. The main idea consists of planning in advance for periodically alternating charging and discharging phases to avoid battery failures. Specifically, the task execution is suspended periodically and for a pre-determined duration, to allow the system to re-charge the battery. At design time, the algorithm computes the duration of the phases, taking into account the characteristics of tasks and the embedded platform, as well as the break-even times of the existing low-power states. At runtime, the algorithm opportunistically extends the duration of the charging states whenever possible, to further increase the energy level. Moreover, we also provide an enhancement to increase the spare bandwidth that can be used by non real-time aperiodic tasks, if included in the workload, without affecting the overall feasibility.

We perform extensive simulations to compare the performance of PCS against the state-of-the-art techniques. We show that when realistic power parameters are considered, PCS outperforms other techniques in terms of *feasibility ratio*, which is the percentage of the task sets that are feasibly scheduled with the given energy profiles. We also evaluate several other performance indicators, such as the number of preemptions and the length of the average sleep intervals.

Section 7.1 introduces the power and workload models, while Section 7.2 gives the details of the proposed approach. Finally, Section 7.3 compares our algorithm with the state-of-the-art techniques, experimentally.

7.1 System model

7.1.1 Power model

As depicted in Figure 7.1, we assume a system with energy harvesting capability. The *harvester unit* is in charge of scavenging energy from the environment and storing in the *energy storage unit* (which may be a battery or supercapacitor). The energy available to the embedded system at time t is denoted by $E(t)$. This energy level is bounded by C, which is called the *battery capacity*. The energy is harvested at the rate of $P_r(t)$. As common in energy harvesting research ([HZK+06, LSK10, ZSA11]), we assume that the operation interval is divided into equal length *time slots* (or, *epochs*): in each time slot (of duration T_r) the energy harvesting/replenishment rate can be assumed to be constant. For example, several papers assumed T_r ranging from 15 to 60 minutes.

The power consumption of the processor in the *active* state (denoted as σ_0) is given by a constant value P_{CPU} that accounts for both dynamic and leakage dissipation. The overall power consumption due to the entire set of remaining system components (e.g., I/O devices/peripherals) is denoted by P_{dev}. The total power P_{σ_0} consumed by the system in the active state is the sum of processor power and total power consumption of the specific subsets of peripherals in use by the running task; thus, at any time, $P_{CPU} \leq P_{\sigma_0} \leq P_{CPU} + P_{dev}$. As in [CMM11, ACM13a], we do not assume DVFS

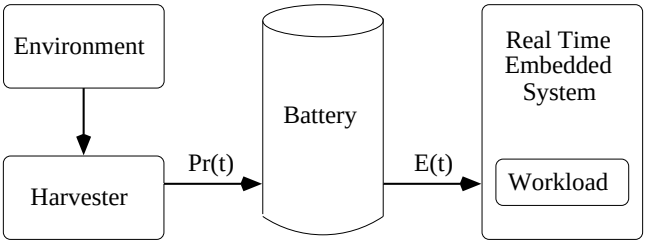


Figure 7.1: Components involved in the energy flow.

capability in the system; i.e., task execution takes place at a constant frequency level.

A set $\Phi = \{\sigma_1, \ldots, \sigma_m\}$ of low-power states features the platform as defined in Section 2.1.

Given the above notation, if the system remains in state σ_k (which may be the *active* state σ_0 as well as one of the *low-power* states σ_i $(i \geq 1)$) from time t_1 to t_2, then the battery energy level at time t_2 is expressed by:

$$E(t_2) = \min\left(C, \quad E(t_1) + \int_{t_1}^{t_2} (P_r(t) - P_{\sigma_k})\, dt\right).$$

7.1.2 Task model

The workload Γ consists of a set of n real-time sporadic independent tasks, $\{\tau_1, \tau_2 \ldots \tau_n\}$, with implicit deadlines. In addition to the parameters in Chapter 2, each task τ_i is characterized by an average power consumption P_i, including both the processor and the peripherals dissipation ($P_{CPU} \leq P_i \leq P_{CPU} + P_{dev}$). Note that two tasks having the same WCET may consume different energy if they use different peripherals. H denotes the *hyperperiod* of the task set, computed as the least common multiple of all the periods: $H = lcm(T_1, \ldots, T_n)$.

Finally, priorities are assigned according to *Rate Monotonic* and tasks are indexed in decreasing priority order, so that τ_1 has the highest priority.

7.2 Proposed approach

The proposed approach is based on a periodic scheme which alternates between active and inactive phases of the processor: the first one is in charge of executing the pending workload, while the second one replenishes energy until the next active phase.

The inactive phase is implemented by adding a new hypothetical periodic task (τ_s) that puts the processor in a low-power state for an interval C_s in every period T_s, in order to charge the battery continuously, without any interruption by other tasks. To this aim, the highest priority in the system is assigned to τ_s; implying that whenever it is ready, the system will be put in a low-power state and continuous re-charging will be enforced in a predictable and periodic fashion. Moreover, its "execution time" C_s is chosen in such a way that the system will be able to exploit the deepest possible low-power state offered by the platform, by considering the break-even times of the existing states, *while still guaranteeing the deadlines of real-time tasks*. In other words, our periodic charging scheme (PCS) provides both a predictable harvesting mechanism

and an ability to comply with the requirements of the low-power states of the processor, in terms of the overhead amortization.

According to this framework, the problem can be reformulated as finding a valid pair of C_s and T_s which avoid deadline misses and energy failures while executing τ_s at the highest-priority level. Note that the assignment of the highest priority to τ_s is critical to enforce its "non-preemptive" execution, to enable the system to enter a low-power state effectively.

In systems with renewable energy, the concept of feasibility is extended to consider also *battery (or, energy) failures* ([CMM11, ACM13a]). Specifically, in addition to guaranteeing task completions no later than their respective deadlines, in order to ensure feasibility, the algorithm must also guarantee that the battery level never drops below a certain threshold E_{low}: $\forall t, E(t) > E_{low}$. Without loss of generality, we consider the case of $E_{low} = 0$; for higher thresholds, the battery capacity can be downsized accordingly and the problem can be re-stated as an instance with $E_{low} = 0$. Note that, if an energy failure happens, it may not immediately lead to a deadline miss as the required time to charge the battery may not violate real-time constraints. Our adopted definition is stricter than this interpretation: our proactive approach treats any energy underflow as a failure condition, which may indeed introduce unpredictability in real-time embedded system design.

Compared to the existing energy-greedy and computation-greedy energy harvesting algorithms, PCS is conceptually much simpler and easier to implement with low online complexity. Moreover, thanks to its design principles, it explicitly considers the time/energy overheads involved in the processor state transitions, through the explicit analysis of the break-even times. In contrast, the energy-greedy algorithms (e.g., PFP_{st} [CMM11]) involve online computation of the existing slack to re-charge the battery, which is, in general, of pseudo-polynomial complexity. Similarly, the computation-greedy algorithms (e.g., the theoretically optimal PFP_{ASAP}) result in very frequent invocation and processor state transitions with prohibitive costs on real systems that have non-zero transition overheads.

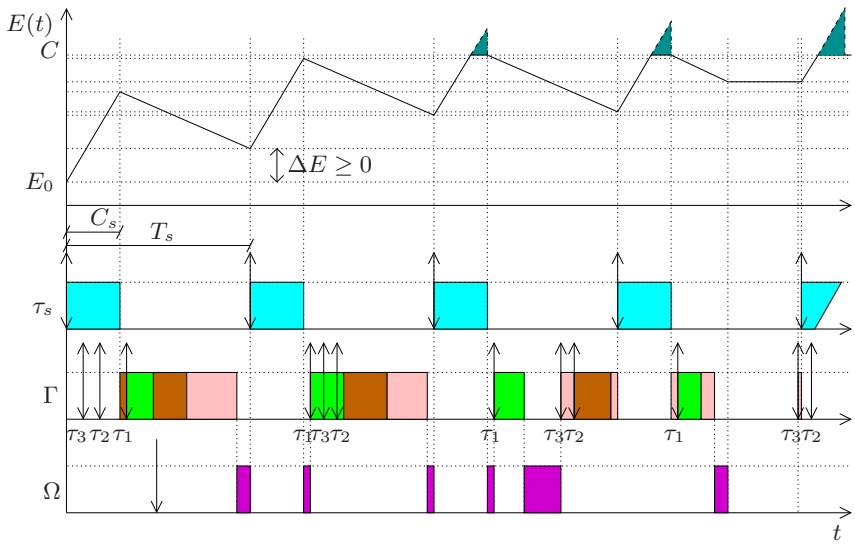


Figure 7.2: Example of algorithm execution.

An example of our approach is illustrated in Figure 7.2, showing how the battery level varies while executing the instances of τ_s and the workload. For the sake of simplicity, the replenishment function has been assumed constant and all tasks consume the same power. Since τ_s runs at the highest priority (in order to guarantee a non-preemptive execution), any time an instance of τ_s is released, the actual running job is preempted, the processor enters into a low-power state, and the battery is replenished. Then, the workload execution is resumed when τ_s instance completes its execution, running for at most $T_s - C_s$ time units before next instance arrives.

We first present the details of the proposed algorithm, then, a sufficient condition is provided to test the system feasibility at design time. Finally, for workloads that may include non real-time components, an online enhancement is introduced to improve the responsiveness of such tasks, without affecting the feasibility of the real-time workload.

7.2.1 Algorithm

This section gives the details of the proposed algorithm: *Periodic Charging Scheme (PCS)*. Specifically, at design time, the algorithm computes the period T_s and charging time C_s which lets τ_s execute periodically in a non-preemptive fashion. Then, at runtime, the algorithm opportunistically compacts idle intervals and τ_s execution, to further extend battery phases and exploit deeper low-power states.

First, let us consider the design-time step. The shortest period among the real-time tasks is assigned to T_s in order to let τ_s have the highest priority and run in a non-preemptive way. The computational time C_s is assigned according to the sensitivity analysis proposed by Bini et al. [BDNB08], which computes the highest spare utilization that τ_s can have, without causing deadline misses among the lower priority tasks. In this case, lower priority tasks correspond to the entire original task set Γ. Although the sensitivity analysis considers fully-preemptive tasks, the non-preemptive execution of τ_s is automatically guaranteed by its highest priority, without invalidating the analysis. Moreover, the specific low-power state σ_k to which the system switches during the τ_s's execution is chosen as the deepest sleep state whose break-even time is shorter than or equal to C_s. The corresponding pseudocode is presented in Algorithm 5.

Algorithm 5 PCS: Design-Time Algorithm

1: **function** PCS_AT_DESIGN_TIME(Γ)
2: $\quad T_s = \min_{\tau_i \in \Gamma} T_i$
3: $\quad C_s = \Delta C_s$ /* From sensitivity analysis [BDNB08] */
4: $\quad \Gamma \leftarrow \Gamma \cup \{\tau_s\}$
5: $\quad k = \max_{\sigma_i \in \Phi \wedge B_{\sigma_i} \leq C_s} i$
6: **end function**

On the other hand, the runtime component of PCS is executed whenever the ready queue becomes empty. Specifically, when the processor is idle, the algorithm first computes earliest possible next arrival time of any periodic task ($next\_arrival$), which can be easily computed given the minimum inter-arrival time information of the tasks. Then, it re-adjusts the next arrival time of τ_s to coincide with $next\_arrival$. In this way, the system enters an *extended* charging phase from the current time until $next\_arrival + C_s$, potentially enabling the exploitation of even deeper low-power states simultaneously.

To prove that the schedulability is not affected by re-adjusting the next invocation time of τ_s arrival, let us assume a generic task set whose feasibility is statically guaranteed. Recall that τ_s has the highest priority in the system. According to the well-known fixed-priority schedulability analysis techniques, the response time of any task is maximized when its job arrives simultaneously with the jobs of higher-priority tasks [BDNB08]. Since the task set is deemed feasible at the static phase, the response time of any task does not exceed its deadline even in that *critical instant*, by definition. Hence, by aligning the next invocation time of τ_s with the $next\_arrival$, other deadlines cannot be compromised. Then, forcing τ_s to arrive at the same time lets us obtain a configuration equivalent to the critical instant, whose feasibility is already assumed in the static analysis.

The pseudocode in Algorithm 6 gives the details of the runtime component of PCS, which computes the actual charge length (T_{charge}), adjusts τ_s's next invocation time $a_{s,j}$ and selects the deepest low-power state to use during that specific charge step.

Algorithm 6 PCS: Runtime Algorithm

1: **function** PCS_AT_RUNTIME (t) ▷ t: CPU becomes idle
2: $t_1 = next\_arrival$
3: $T_{charge} = C_s + (t_1 - t)$
4: $a_{s,j} = t_1$
5: $k' = \max\limits_{\sigma_i \in \Phi \wedge B_{\sigma_i} \leq T_{charge}} i$
6: **end function**

An example is reported in Figure 7.3, representing the schedules without and with the runtime component of PCS. Specifically, when the runtime component is enabled, it is invoked at t (when the CPU becomes idle) and, computing the next arrival time in Γ as t_1, τ_s's execution and the idle interval are compacted to form a single longer interval (of duration $C_s + (t_1 - t)$). With longer intervals, the algorithm gains the ability to potentially exploit deeper low-power states.

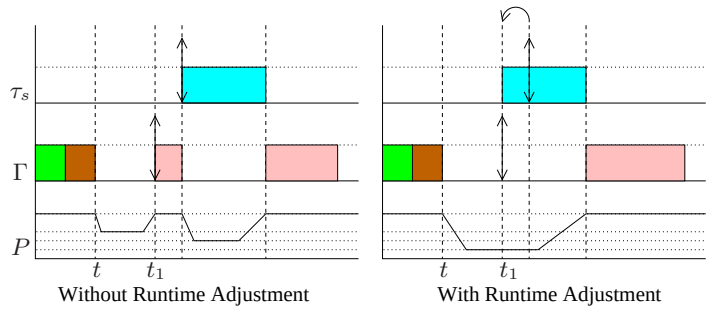


Figure 7.3: Schedule examples without and with the PCS runtime component.

At runtime, the introduced complexity for the scheduler is negligible as the algorithm only requires to schedule the additional task τ_s. The complexity of the static (design-time) component is pseudo-polynomial with respect to the number of tasks due to the sensitivity analysis. Finally, the runtime component of PCS has also low complexity: assuming the earliest next arrival time can be evaluated in constant time,

then the overall complexity is linear with respect to the number of low-power states, which is $O(m)$.

7.2.2 A sufficient condition for schedulability

In PCS, the feasibility in terms of timing constraints is explicitly guaranteed through the sensitivity analysis. On the other hand, providing a *simple* necessary and sufficient condition to check whether a given system configuration, with a certain initial energy budget and harvesting profile is feasible or not, is not trivial.

Nevertheless, a sufficient condition to guarantee execution without energy failures can be derived, for design-time (offline) analysis. The condition is based on guaranteeing that, even in the worst-case scenario, the difference between the harvested energy and the consumed energy during one period T_s of τ_s is not negative. If this holds, due to the periodic nature of τ_s's invocations, the energy level of the system will never decrease in the long run, guaranteeing feasibility.

Specifically, the difference in the energy levels at the beginning of two consecutive invocations of τ_s is given by:

$$\Delta E = (P_r - P_{\sigma_k}) \cdot C_s - (P_{act} - P_r) \cdot (T_s - C_s) \geq 0, \tag{7.1}$$

where P_{act} is the maximum task power consumption in the active state $\left(P_{act} = \max_{\tau_i} P_i\right)$ and σ_k is the low-power state selected by the offline phase of PCS.

An intuitive example is shown in Figure 7.4, illustrating how the battery level E varies while executing τ_s and real-time tasks. The first term in Eq. (7.1), $(P_r - P_{\sigma_k}) \cdot C_s$, gives the net energy gain during the charging phase, while the second term $(P_{act} - P_r) \cdot (T_s - C_s)$ corresponds to the energy loss during the discharging phase.

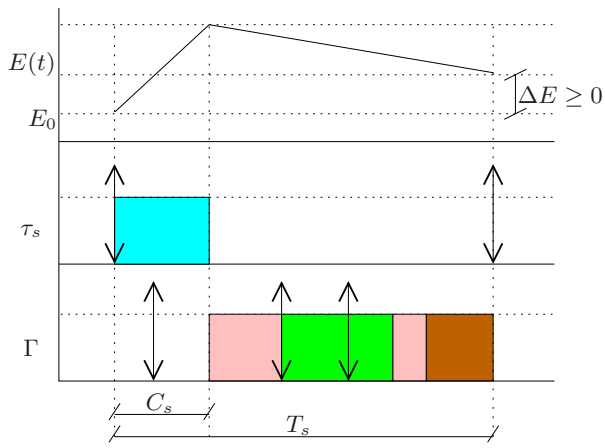


Figure 7.4: Energy level changes during one period of τ_s

Eq. (7.1) can be reformulated with respect to C_s, as:

$$C_s \geq \frac{P_{act} - P_r}{P_{act} - P_{\sigma_k}} T_s, \tag{7.2}$$

Note that this condition is pessimistic, because it is assumed that the system is always in active state (σ_0), executing the task with the maximum power consumption

characteristics, when τ_s is not running (i.e., during the discharge phase of length $T_s - C_s$). However, it provides a simple formula that can be checked in constant time, regardless of the initial energy level. In the rest of the paper, we refer to this version of PCS which checks feasibility at design time by using Equation (7.2), as PCS^*.

7.2.3 Enhancing the algorithm for mixed workloads

In some cases, thanks to a favorable scenario, the energy stored in the battery may reach the capacity, leading to a waste of energy. Although battery overflows do not represent a problem for neither real-time nor energy constraints, the scheduler may optimize the use of resources, such as energy and CPU time.

In fact, such an optimization may be quite useful for *mixed workloads* that contain both sporadic real-time and aperiodic non-real-time (NRT) tasks. For mixed workloads, the traditional objective is to meet the hard deadlines of the real-time tasks, while improving the *responsiveness* (i.e., average response time) of NRT tasks [But04]. Hence, in our settings, some instances of τ_s may be skipped, making available its allocated computation time to NRT tasks. Figure 7.2, at the bottom schedule, illustrates the execution of an NRT task Ω during the idle intervals of the PCS schedule. If the third job of τ_s is discarded, the response time of Ω can be shortened without causing any energy failure.

However, skipping too many instances of τ_s may hurt feasibility in the long term; in particular, as the harvesting rate P_r changes at the end of each epoch T_r, typically of length 15-30 minutes, one should still try to maximize the battery energy level as much as possible by the end of the current epoch. Consequently, our proposed enhancement is based on *skipping* one instance of τ_s out of $j+1$ consecutive instances ($j \geq 1$), while ensuring maximization of the battery level by the end of epoch. Specifically, by denoting the initial energy level at the beginning of the epoch as E_0, the net energy harvested until the end of the current epoch has to be no less than the available capacity $(C - E_0)$ in the battery:

$$N \cdot \Delta E^* \geq C - E_0 \geq 0, \tag{7.3}$$

Above, N is the number of skipped instances during the current epoch $\left(N = \left\lfloor \frac{T_r}{(j+1)\cdot T_s} \right\rfloor\right)$ and ΔE^* is the difference between harvested and consumed energy in a time interval of length $(j+1)\cdot T_s$:

$$\Delta E^* = j \cdot (P_r - P_{\sigma_k}) \cdot C_s - (P_{act} - P_r) \cdot (j \cdot T_s - j \cdot C_s + T_s) \geq 0. \tag{7.4}$$

Since $T_r >> T_s$, we can approximate N as $\frac{T_r}{(j+1)\cdot T_s}$. From Eqs (7.3) and (7.4), we can derive a lower bound for j:

$$j \geq \frac{(P_{act} - P_r) \cdot T_s + \frac{C-E_0}{T_r} T_s}{(P_{act} - P_{\sigma_k}) \cdot C_s - (P_{act} - P_r) \cdot T_s - \frac{C-E_0}{T_r} T_s}, \tag{7.5}$$

The value of j must be set as the smallest integer that satisfies Eq. (7.5) – a higher value of j may decrease the responsiveness of the NRT workload, while wasting energy that cannot be stored in the battery. This enhanced version of PCS, denoted by PCS^{NRT}, is invoked whenever an epoch starts and has constant-time complexity ($O(1)$), as only Eq (7.5) needs to be solved.

7.3 Experimental results

In this section, we provide the results of simulation experiments we carried out to evaluate the performance of our proposed algorithms under different system parameters.

We considered an embedded system equipped with an NXP LPC1768 [NXP] processor (ARM Cortex M3 [ARM]), powered by a battery with capacity $C = 500mAh$ and two solar panels, each providing a maximum of $500mW$. The power consumption of the processor in active state, without considering the peripherals, is $P_{CPU} \approx 690mW$. When all peripherals are activated, the overall power consumption is around $1W$, giving $P_{dev} = 310mW$. Two low-power states are considered: *idle* (σ_1) and *sleep* (σ_2). Their power consumption and break-even times are $P_{\sigma_1} = 490mW$, $B_{\sigma_1} \approx 0ms$, $P_{\sigma_2} = 290mW$ and $B_{\sigma_2} = 15ms$. Observe that, although the *sleep* state consumes least power, its break-even time is not negligible.

The synthetic task sets used in the tests are composed of 10 tasks randomly generated using the UUniFast algorithm [BB05], where each period T_i is uniformly distributed in the range of $[40, 500]ms$. In our simulations, we generated 4000 task sets (200 for each utilization value under consideration). The power consumption P_i of each job of task τ_i is computed as:

$$P_i = P_{CPU} + x_i \cdot P_{dev},$$

where $0 < x_i \leq 1.0$ is a real number generated randomly. By choosing different x_i values, tasks can consume different amount of power per time unit of execution.

We report the results of our experiments in three parts. The first set evaluates the effectiveness of the proposed algorithms in terms of the ratio of the task sets that are scheduled in feasible manner (called the *feasibility ratio*), with respect to both timing and energy constraints. The second set of experiments assess several online metrics, such as average sleep interval length and preemption count, and the last set analyzes the spare CPU bandwidth that is made available to potential non-real-time tasks.

7.3.1 Feasibility ratio

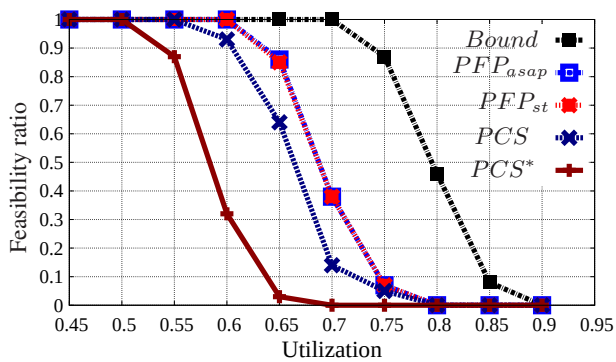
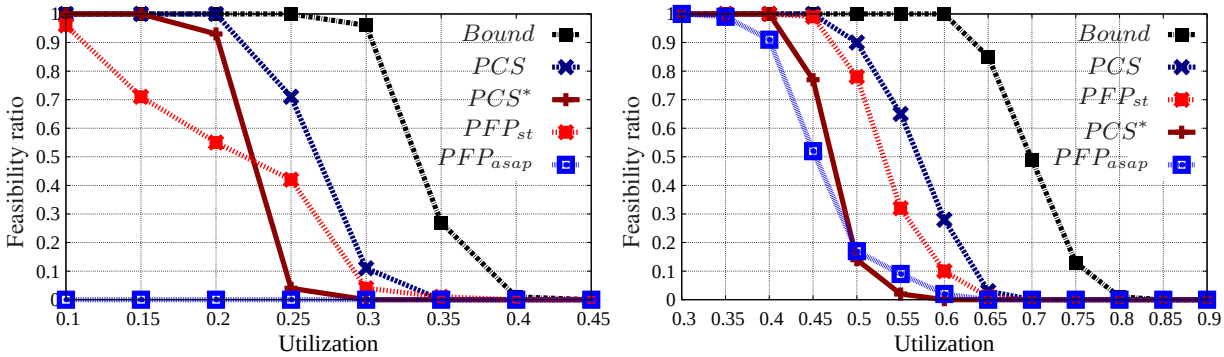


Figure 7.5: Feasibility ratio vs. Utilization ($B_{\sigma_1} = 0$, $P_{\sigma_1} = 0W$, $P_r = 70\%$).

To evaluate the feasibility ratio under different system configurations, we implemented the following algorithms in our discrete-event simulator:

- PFP_{asap}: the computation-greedy algorithm whose optimality for fixed-priority systems with renewable energy, but only under negligible state transition overheads assumption, was formally proven in [ACM13a];
- PFP_{st} (also called $EDeg$, from [CMM11]): the energy-greedy algorithm whose feasibility performance was shown to lag slightly behind PFP_{asap} in [ACM13a];
- PCS^* that evaluates only the sufficient condition given by Eq. (7.2) at design time, to guarantee the feasibility;
- PCS – the proposed algorithm;
- *Bound* that represents a theoretical limit on the feasibility performance of any scheduling algorithm.



(a) $P_r = 50\%$. (b) $P_r = 70\%$.

Figure 7.6: Feasibility ratio vs. Utilization for non-negligible transition overhead.

The executions of PFP_{asap}, PFP_{st} and PCS are simulated, assuming an initial energy level of $E_0 = 0$ (the worst-case scenario) and checking for any deadline misses or battery failures during the hyperperiod H. To implement *Bound*, we adopted a methodology similar to the one suggested by Pagani and Chen [PC13], by transforming the problem into another one where tasks have identical release times and identical deadlines (equal to the hyperperiod H of the original task set) while keeping task utilizations the same. Specifically, the battery capacity limit is ignored, and the longest possible charging interval within the hyperperiod is considered. In these ideal settings, the entire workload with utilization U can be procrastinated for $(1 - U) \cdot H$ time units and the system is feasible if and only if the energy harvested in $(1 - U) \cdot H$ is higher than or equal to the energy consumed in $U \cdot H$ time units of execution.

Since PFP_{asap} and PFP_{st} were developed assuming negligible state transition overheads, they have been updated to incorporate a simple mechanism to deal with those overheads at run-time. Specifically, PFP_{asap}, which procrastinates only to harvest the energy necessary to execute the next computational unit, chooses the deepest sleep state whose break-even time is not longer than the required time to harvest the missing energy amount. Similarly, PFP_{st}, which exploits the whole available slack to charge the battery when it becomes empty, selects the deepest low-power state whose break-even time is shorter than or equal to the target procrastination delay. In addition, both algorithms are enhanced by putting the processor to the deepest low-power state which lets the system be fully operational by the next job arrival, when the CPU is idle.

In Figure 7.5, we first report the results for a system with "ideal" settings, that is, the one with a negligible power dissipation and zero break-even time associated with the sleep state (i.e., $B_{\sigma_2} = 0ms$ and $P_{\sigma_2} = 0W$). The harvesting rate P_r is set to 70% of the maximum system power consumption ($P_{CPU} + P_{dev}$).

As expected, in this scenario with no overheads, PFP_{asap}'s optimality is demonstrated: it yields a feasibility ratio higher than PFP_{st} and PCS. Also, in accordance with what is experimentally shown in [ACM13a], PFP_{st} is a close second – in fact, its performance almost coincides with that of PFP_{asap}. PCS comes next, showing that putting periodically the processor in sleep state is not the best approach on systems with zero transition overhead and zero sleep power.

However, when a realistic set of low-power states is considered, the picture changes entirely. The results are reported in Figure 7.6(a) and Figure 7.6(b), for $P_r = 50\%$ and 70% of the maximum power consumption ($P_{CPU} + P_{dev}$), respectively.

Our approach outperforms PFP_{st} and PFP_{asap} as it periodically guarantees replenishment phases which last longer than the sleep state's break-even time, overcoming the limitations due to short idle intervals. For instance, when $P_r = 70\%$, PFP_{asap}'s performance degrades when $U = 0.35$, while PCS successfully schedules all the task sets up to $U = 0.45$. The performance of PFP_{asap} drops because it is able to exploit only shallow low-power states. The difference between PCS^* and PCS is entirely due to the pessimistic nature of the offline test. Finally, although PFP_{st}'s performance is close to PCS, its online complexity is pseudo-polynomial whereas our simple algorithm has a linear complexity at runtime.

Figure 7.7 presents the impact of the relative harvesting rate ($P_r/(P_{CPU} + P_{dev})$) on the feasibility ratio when $U = 0.4$. In other words, this analysis shows the minimum required harvesting rate that guarantees schedulability. Again, PCS offers the best performance (besides *Bound*, which gives the theoretical limit): it guarantees the feasibility for the lowest harvesting power.

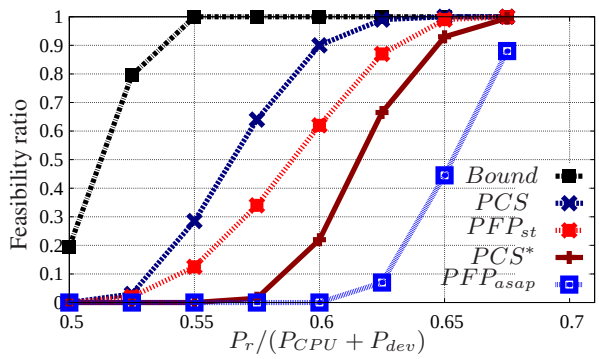
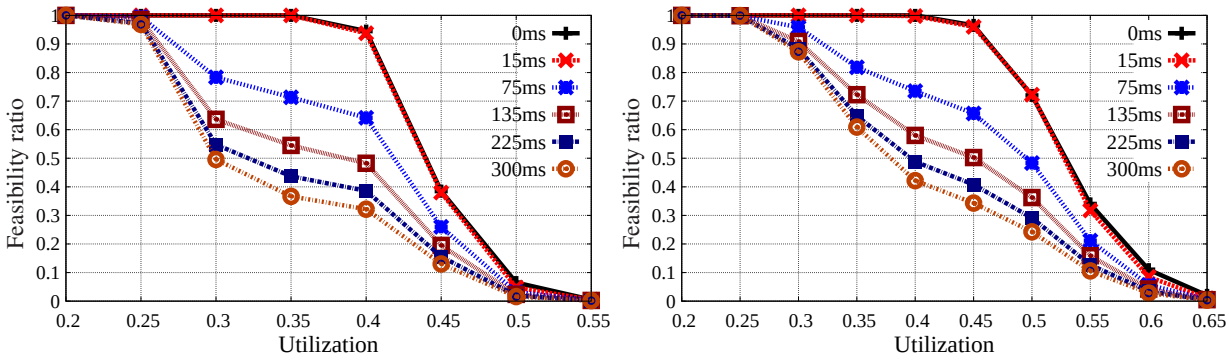


Figure 7.7: Feasibility ratio vs. Harvesting rate ($U = 0.4$).

Next, we analyze how the performance changes as a function of the different break-even times associated with the *sleep* state, in Figure 7.8(a) and Figure 7.8(b), for PCS^* and PCS, respectively. The results are obtained for $P_r = 65\%$, and the minimum period value of $40ms$. As expected, the results show that the longer the break-even time, the lower the feasibility ratio. However, PCS is less affected than PCS^* by the changes in the break-even time: this is because, it opportunistically manages to compact inactive intervals at runtime.



(a) Under PCS^*.

(b) Under PCS.

Figure 7.8: Feasibility ratio vs. Utilization under different break-even times for $P_r = 65\%$.

7.3.2 Online metrics

This subsection considers some online metrics that are relevant to characterize the algorithms' performance at runtime:

- *average time spent in sleep state:* this metric shows the capability to exploit the deepest low-power state for as long as possible. In generally, the longer the sleep intervals, the higher the harvested energy and the higher the probability to execute the workload without battery failures;
- *number of preemptions*: this metric shows the total number of preemptions experienced by each algorithm. Obviously, an algorithm with prohibitive number of preemptions is not desirable, due to the associated overhead;
- *average battery energy level:* this metric highlights the success of the algorithms in effectively harvesting energy, while executing the workload.

The comparisons in this section involve only PCS and PFP_{st}, due to the fact that the results in [ACM13a], that are in line with ours, clearly show that PFP_{st} outperforms PFP_{asap} by a significant margin when considering these online metrics.

First, we analyze the average length of the sleep intervals for PFP_{st} and PCS in Figure 7.9. We observe that PCS guarantees longer sleep intervals on the average; however their lengths tend to decrease with increasing utilization (the time assigned to τ_s becomes shorter). Conversely, the average sleep interval length for PFP_{st} first remains constant with the utilization (due to the feasibility constraint and reaching the battery capacity during recharging), before dropping. It is worth noting that for PFP_{st}, the total length of the idle intervals is comparable to that of the sleep intervals. Note that some values are not reported for certain data points whenever the corresponding algorithm does not generate a feasible solution for that utilization value.

When the absolute time spent in low-power states (idle + sleep) is considered instead of the average duration, PCS and PFP_{st} have similar trends. Specifically, the difference is around only 5% of the entire hyperperiod during which PFP_{st} puts the system in the idle state rather than the sleep state, due to the break-even time.

Next, in Figure 7.10, we show the number of preemptions of PCS, normalized with respect to PFP_{st}'s. PCS has the side effect of introducing a higher preemption

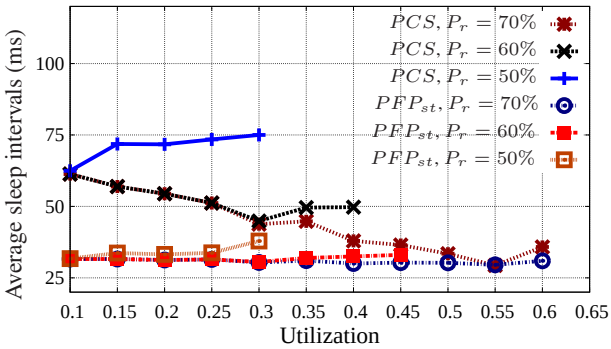


Figure 7.9: Average sleep interval vs. Utilization under PCS and PFP_{st}.

count than PFP_{st} as the system executes the additional charging task τ_s with the period set to the minimum period value. However, except for very low utilization values, the increase is only around 50% compared to PFP_{st} during the hyperperiod. The preemption ratio is not reported for utilization values where PFP_{st} or PCS does not generate feasible values.

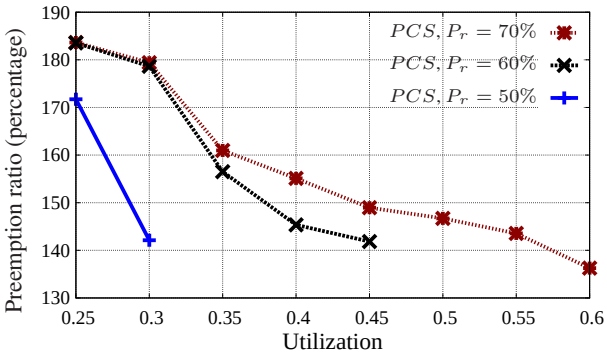


Figure 7.10: Ratio of preemption number in PCS to that in PFP_{st} (percentage).

Finally, in Figure 7.11, we analyze the average battery level, normalized to the battery capacity, within a hyperperiod. We observe that both PCS and PFP_{st} guarantee a similar average battery level, even though, charging principles and timing are different. In general, the higher the utilization, the lower the average battery level as the available time to charge the battery is shorter. In addition, the higher the harvesting rate P_r, the higher the average battery level as the charging process is more effective. Despite similar performance trends, we note that the online component of PFP_{st} has pseudo-polynomial time complexity (due to the computation of the maximum slack at runtime), while PCS has linear complexity.

7.3.3 Effective spare CPU bandwidth

Finally, we undertake an experimental analysis of PCS^{NRT} which targets allocating as much CPU bandwidth as possible to non-real-time (NRT) tasks in order to improve their responsiveness. Recall that under PCS^{NRT}, effectively additional CPU band-

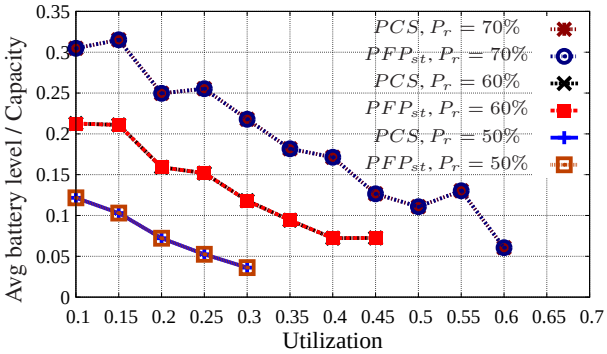
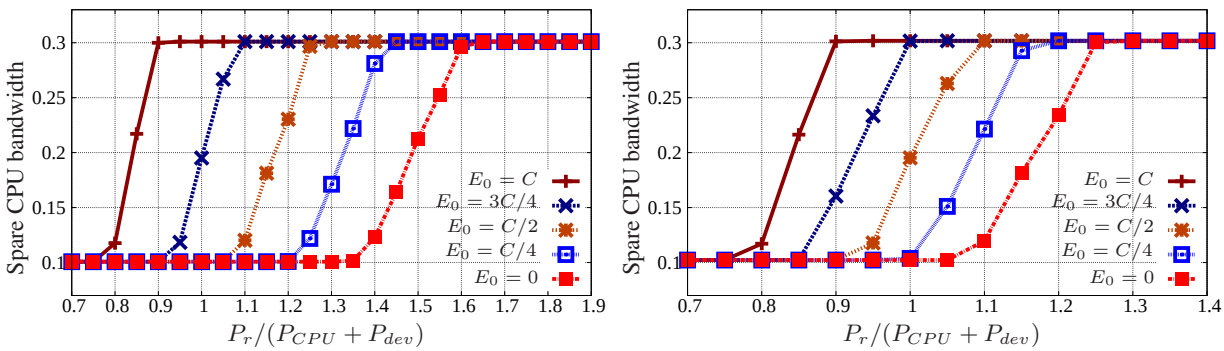


Figure 7.11: Average battery level vs. Utilization under PCS and PFP_{st}.

width is reserved by skipping an instance of τ_s out of $j + 1$ consecutive instances, where j is computed statically at the beginning of each epoch, to preserve feasibility in terms of deadlines and energy constraints.

In these experiments, the utilization of the real-time task set is set to 0.5, potentially leaving 50% of the CPU time to non-real-time tasks, regardless of τ_s's configuration. Figure 7.12(a) shows the effective spare CPU bandwidth that is left for the NRT workload, by considering also τ_s's invocations, as a function of the harvesting rates and initial battery level E_0. In this first simulation, the battery capacity is $C = 500mAh$ and the average τ_s utilization is 0.4. When the online enhancement is not able to skip any τ_s instance (mainly due to low initial battery level E_0, or scarce harvested power P_r), the CPU bandwidth allocated to NRT tasks is around 10%, which is equivalent to the static slack $(0.5 - 0.4)$. As soon as more favorable execution scenarios occur (either thanks to the higher initial battery level E_0 or higher harvesting rate P_r), several of τ_s's jobs can be skipped without hurting the feasibility. Specifically, in the best case, the effective spare CPU bandwidth goes from 10% to 30%, meaning that the actual utilization of τ_s is 0.2 (i.e., an instance is skipped out of 2).

In Figure 7.12(b), the same analysis is repeated for a system with lower battery capacity, $C = 250mAh$. Since the maximum amount of energy that can be stored is lower, the upper bound on the effective spare CPU bandwidth is now reached sooner.



(a) $C = 500mAh$.

(b) $C = 250mAh$.

Figure 7.12: Feasibility ratio vs. Utilization under different break-even times for $P_r = 65\%$.

Chapter 8

Conclusions

This PhD thesis considered the energy-saving issue in real-time systems in which the execution correctness does not depend only on the result values, but also on when such results are produced. Most attention was focused on embedded systems which are specific-purpose computers in charge of achieving dedicated functions, while exploiting limited resources (such as memory, computational capability and power supplier).

The analysis started considering single-core systems, which are still widely used in the embedded domain thanks to their high reliability and predictability. The first contribution exploited the limited preemption task model to further reduce the overall energy dissipation with respect to the actual state of art. Another work handled the impact of additional constraints (due to the communication bandwidth) by proposing a solution which synchronizes task execution and message transmission, putting the system in sleep state for longer intervals. In addition, several algorithms with low online complexity were implemented in a real system to evaluate their real effectiveness. Finally, the previous work was extended by profiling a bunch of widely used platforms in the embedded systems domain and then, using those models to simulate the most popular energy-aware real-time scheduling algorithms, showing their behaviors in practice.

Multi-core systems have also been considered, mostly focusing on high performance computing. More precisely, two opposite partitioning heuristics were compared: the first approach aims at spreading the workload on all the available cores (lowering the utilization on each core), while the second strategy compacts the workload on few cores (increasing their utilization and switching off the others). In contrast with the actual belief, the second approach provides a lower energy consumption on a real system. Such a result is particularly interesting as the scheduler and load balancer in actual operating systems implement the first strategy which guarantees lower response times.

The energy issue has also been analyzed in distributed systems in which a new trade-off between dissipation and fault tolerance (implemented through task redundancy) is introduced. Several partitioning strategies are compared to investigate the bliss point among all the constraints: real-time requirements, bandwidth utilization, lifetime, required redundancy and energy dissipation.

Finally, the energy harvesting problem has been studied, considering external power suppliers whose contribution is time dependent. More precisely, a high predictable algorithm is introduced which, combining task and environmental parameters, guarantees the schedulability for a higher number of task sets with respect to the actual state of art.

Bibliography

[AA04] Tarek A. AlEnawy and Hakan Aydin. On energy-constrained real-time scheduling. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2004.

[AA05] Tarek A. AlEnawy and Hakan Aydin. Energy-constrained scheduling for weakly-hard real-time systems. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2005.

[ACM13a] Yasmina Abdeddaim, Younes Chandarli, and Damien Masson. The optimality of PFP_{asap} algorithm for fixed-priority energy-harvesting real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[ACM13b] Yasmina Abdeddaïm, Younès Chandarli, and Damien Masson. Toward an optimal fixed-priority algorithm for energy-harvesting real-time systems. In *Proceedings of the work in progress session of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

[ADM11] S. Altmeyer, R.I. Davis, and C. Maiza. Pre-emption cost aware response time analysis for fixed priority pre-emptive systems. Technical report, University of York, 2011.

[ADZ06] Hakan Aydin, Vinay Devadas, and Dakai Zhu. System-level energy management for periodic real-time tasks. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2006.

[AMD] Amd's web site. http://www.amd.com/.

[AMMMA01] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2001.

[AMMMA04] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *Transactions on Computers*, 53(5), 2004.

[AP11] Muhammad Ali Awan and Stefan M. Petters. Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[AP13] M.A. Awan and S.M. Petters. Energy-aware partitioning of tasks onto a heterogeneous multi-core platform. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, 2013.

[ARM] Arm web site. http://www.arm.com/.

[AY03] Hakan Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[AY04] H. Aydin and Qi Yang. Energy - responsiveness tradeoffs for real-time systems with mixed workload. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[BAB14] Mario Bambagini, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for uni-processor real-time systems: a survey. submitted to ACM Transaction on Embedded Computing, 2014.

[Bar03] Sanjoy K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1), 2003.

[BB05] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2), 2005.

[BB10] M. Bertogna and S. Baruah. Limited preemption EDF scheduling of sporadic task systems. *Transactions on Industrial Informatics*, 2010.

[BBB12] Mario Bambagini, Giorgio Buttazzo, and Marko Bertogna. Energy saving exploiting the limited preemption task model. In *Real-Time Scheduling Open Problems Seminar (RTSOPS)*, 2012.

[BBB13] Mario Bambagini, Giorgio Buttazzo, and Marko Bertogna. Energy-aware scheduling for tasks with mixed energy requirements. In *Real-Time Scheduling Open Problems Seminar (RTSOPS)*, 2013.

[BBDM00] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration Systems*, 8(3), 2000.

[BBL09] Enrico Bini, Giorgio Buttazzo, and Giuseppe Lipari. Minimizing cpu energy in real-time systems with discrete speed management. *Transactions on Embedded Computing Systems*, 8(4), 2009.

[BBLB03] Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2003.

[BBM+10] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2010.

[BBMB13a] Mario Bambagini, Marko Bertogna, Mauro Marinoni, and Giorgio Buttazzo. An energy-aware algorithm exploiting limited preemptive scheduling under fixed priorities. In *Proceedings of the IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013.

[BBMB13b] Mario Bambagini, Marko Bertogna, Mauro Marinoni, and Giorgio Buttazzo. On the impact of runtime overhead on energy-aware scheduling. In *Workshop on Power, Energy, and Temperature Aware Realtime Systems (PETARS)*, 2013.

[BBY11a] Marko Bertogna, Giorgio Buttazzo, and Gang Yao. Improving feasibility of fixed priority tasks using non-preemptive regions. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2011.

[BBY11b] Marko Bertogna, Giorgio C. Buttazzo, and Gang Yao. Improving feasibility of fixed priority tasks using non-preemptive regions. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2011.

[BBY13] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems: a survey. *Transactions on Industrial Informatics*, 9(1), 2013.

[BDNB08] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems*, 39(1-3), 2008.

[BLBL13] Mario Bambagini, Juri Lelli, Giorgio Buttazzo, and Giuseppe Lipari. On the energy-aware partitioning of real-time tasks on homogeneous multi-processor systems. In *Proceedings of the IEEE International Conference on Energy Aware Computing (ICEAC)*, 2013.

[BLV09] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1-3), 2009.

[BMB14] Mario Bambagini, Bertogna Marko, and Giorgio Buttazzo. On the effectiveness of energy-aware real-time scheduling algorithms on single-core platforms. In *Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, 2014.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 1990.

[BPMB11] Mario Bambagini, Francesco Prosperi, Mauro Marinoni, and Giorgio Buttazzo. Energy management for tiny real-time kernels. In *Proceedings of the IEEE International Conference on Energy Aware Computing (ICEAC)*, 2011.

[BR03] B. Brock and K. Rajamani. Dynamic power management for embedded systems. In *Proceedings of the IEEE International Conference on Systems-on-Chip (SOC)*, 2003.

[Bur94] Alan Burns. Preemptive priority based scheduling: An appropriate engineering approach. *Advances in Real-Time Systems*, 1994.

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems*. Springer-Verlag, 2004.

[CAM12] Y. Chandarli, Y. Abdeddaim, and D. Masson. The fixed priority scheduling problem for energy harvesting real-time systems. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[CG06] Hui Cheng and Steve Goddard. Eeds nr: An online energy-efficient i/o device scheduling algorithm for hard real-time systems with non-preemptible resources. In *Proceedings of the Euromicro Conference on Real-Time Systems*, 2006.

[CH14] Aaron Carroll and Gernot Heiser. Unifying dvfs and offlining in mobile multicores. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, RTAS, 2014.

[CK05] Jian-Jia Chen and Tei-Wei Kuo. Voltage scaling scheduling for periodic real-time tasks in reward maximization. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2005.

[CK06] Jian-Jia Chen and Tei-Wei Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. *SIGPLAN Not.*, 41(7), 2006.

[CK07] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[CMM11] Maryline Chetto, Damien Masson, and Serge Midonnet. Fixed priority scheduling strategies for ambient energy-harvesting embedded systems. In *Proceedings of the IEEE/ACM International Conference on Green Computing and Communications (GreenCom)*, 2011.

[COR] Coremark web site. http://www.eembc.org/coremark/.

[CQ13] Maryline Chetto and Audrey Queudet. Clairvoyance and online scheduling in real-time energy harvesting systems. *Real-Time Systems*, 2013.

[CSB95] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power cmos digital design. *Solid State Circuits*, 1995.

[CWT09] Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposioum (RTAS)*, 2009.

[DA08] Vinay Devadas and Hakan Aydin. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, 2008.

[DD09] Isabel Dietrich and Falko Dressler. On the lifetime of wireless sensor networks. *ACM Trans. Sen. Netw.*, 5(1), 2009.

[DEL] Dell's web site. http://www.dell.com/.

[dLJ06] Pepijn de Langen and Ben Juurlink. Leakage-aware multiprocessor scheduling for low power. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, 2006.

[DW95] Robert Davis and A. Welling. Dual priority scheduling. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 1995.

[EGCC11] Hussein El Ghor, Maryline Chetto, and Rafic Hage Chehade. A real-time scheduling framework for embedded systems with environmental energy harvesting. *Computers and Electrical Engineering*, 2011.

[Gat07] Bill Gates. Information technology: A robot in every home. *j-SCI-AMER*, 296(1), 2007.

[Gor13] A. Gore. *The Future: Six Drivers of Global Change*. Random House Publishing Group, 2013.

[GPP10] Mohammad Ghasemazar, Ehsan Pakbaznia, and Massoud Pedram. Minimizing energy consumption of a chip multiprocessor through simultaneous core consolidation and dvfs. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, ISCAS, 2010.

[GSL07] Min-Sik Gong, Yeong Rak Seong, and Cheol-Hoon Lee. On-line dynamic voltage scaling on processor with discrete frequency and voltage levels. In *Proceedings of the IEEE International Conference on Convergence Information Technology (ICCIT)*, 2007.

[HCK06] Chia-Mei Hung, Jian-Jia Chen, and Tei-Wei Kuo. Energy-efficient real-time task scheduling for a dvs system with a non-dvs processing element. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2006.

[HZK+06] J. Hsu, S. Zahedi, A. Kansal, M. Srivastava, and V. Raghunathan. Adaptive duty cycling for energy harvesting systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 2006.

[INT] Intel's web site. http://www.intel.com/.

[ISG07] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *Transactions on Algorithms*, 3(4), 2007.

[IY98] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 1998.

[JG04] Ravindra Jejurikar and Rajesh Gupta. Procrastination scheduling in fixed priority real-time systems. In *Proceedings of the ACM International Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2004.

[JG05a] Ravindra Jejurikar and Rajesh Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the ACM International Conference on Design Automation Conference (DAC)*, 2005.

[JG05b] Ravindra Jejurikar and Rajesh Gupta. Energy aware non-preemptive scheduling for hard real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[JPG04] Ravindra Jejurikar, Cristiano Pereira, and Rajesh K. Gupta. Leakage aware dynamic voltage scaling for real time embedded systems. In *Proceedings of the IEEE International Conference on Design Automation Conference (DAC)*, 2004.

[KDMB12] H. Kooti, Nga Dang, D. Mishra, and E. Bozorgzadeh. Energy budget management for energy harvesting embedded systems. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983.

[Kim06] Taewhan Kim. Application-driven low-power techniques using dynamic voltage scaling. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2006.

[KKLR11] Arvind Kandhalu, Junsung Kim, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar. Energy-aware partitioned fixed-priority scheduling for chip multiprocessors. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.

[KKM04] Woonseok Kim, Jihong Kim, and Sang Lyul Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[KSH02] Roy Kaushik, Mukhopadhyay Saibal, and Mahmoodi-Meimand Hamid. Leakage current in deep-submicron cmos circuits. *Circuits, Systems, and Computers*, 11(6), 2002.

[LIN] Linux kernel's web site. http://www.kernel.org.

[LKL07] Jaewoo Lee, Kern Koh, and Chang-Gun Lee. Multi-speed dvs algorithms for periodic tasks with non-preemptible sections. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 1973.

[LRK03] Y.-H. Lee, K.P. Reddy, and C.M. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[LS04] Cheol-Hoon Lee and Kang G. Shin. On-line dynamic voltage scaling for hard real-time systems using the edf algorithm. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2004.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 1989.

[LSK10] Ren-Shiou Liu, P. Sinha, and C.E. Koksal. Joint energy management and resource allocation in rechargeable sensor networks. In *Proceedings of the IEEE INFOCOM*, 2010.

[LSP08] Martin Lawitzky, David C. Snowdon, and Stefan M. Petters. Integrating real-time and power management in a real system. In *Proceedings of the IEEE International Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2008.

[MBP+11] Mauro Marinoni, Mario Bambagini, Francesco Prosperi, Francesco Esposito, Gianluca Franchino, Luca Santinelli, and Giorgio Buttazzo. Platform-aware bandwidth-oriented energy management algorithm for real-time embedded systems. In *Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, 2011.

[MBTB06] C. Moser, D. Brunelli, L. Thiele, and L. Benini. Real-time scheduling with regenerative energy. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems (ECRTS)*, 2006.

[MCT10] C. Moser, Jian-Jia Chen, and L. Thiele. Dynamic power management in environmentally powered systems. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[MHQ07] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. Transition-overhead-aware voltage scheduling for fixed-priority real-time systems. *Transactions on Design Automation of Electronic Systems*, 12(2), 2007.

[MIC] Microsoft's web site. http://www.microsoft.com/.

[Mit14] Sparsh Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *ArXiv e-prints*, January 2014.

[MS01] T. Martin and D. Siewiorek. Non-ideal battery and main memory effects on cpu speed-setting for low power. *Very Large Scale Integration Systems*, 9(1), 2001.

[NC10] S.G. Narendra and A.P. Chandrakasan. *Leakage in Nanometer CMOS Technologies*. Integrated Circuits and Systems. Springer, 2010.

[NRM+06] Alon Naveh, Efraim Rotem, Avi Mendelson, Simcha Gochman, Rajshree Chabukswar, Karthik Krishnan, and Arun Kumar. Power and thermal management in the intel coretm duo processor. *Intel Technology Journal*, 10(2), 2006.

[NXP] Nxp web site. http://www.nxp.com/.

[PBB+12] Francesco Prosperi, Mario Bambagini, Giorgio Buttazzo, Mauro Marinoni, and Gianluca Franchino. Energy-aware algorithms for tasks and bandwidth co-allocation under real-time and redundancy constraints. In *Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, 2012.

[PC13] Santiago Pagani and Jian-Jia Chen. Energy efficiency analysis for the single frequency approximation (sfa) scheme. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013.

[PLM+12] Vinicius Petrucci, Orlando Loques, Daniel Mosse, Rami Melhem, Neven Abou Gazala, and Sameh Gobriel. Thread assignment optimization with real-time performance and memory bandwidth guarantees for energy-efficient heterogeneous multi-core systems. In *Proceedings of the 17th IEEE International Conference on Real-Time and Embedded Technology and Application Symposium*, RTAS, 2012.

[PS01] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *Operating Systems*, 35, 2001.

[QNHM04] Gang Quan, Linwei Niu, Xiaobo Sharon Hu, and Bren Mochocki. Fixed priority scheduling for reducing overall energy on variable voltage processors. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2004.

[RCM96] I. Ripoll, A. Crespo, and Aloysius K. Mok. Improvement in feasibility testing for real-time systems. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 1996.

[REC] Residential energy consumption survey by the u.s. energy information administration (eia). http://www.eia.gov/consumption/residential/.

[RLZR10] A. Rowe, K. Lakshmanan, Haifeng Zhu, and R. Rajkumar. Rate-harmonized scheduling and its applicability to energy management. *Transactions on Industrial Informatics*, 6(3), 2010.

[RV03] Daler Rakhmatov and Sarma Vrudhula. Energy management for battery-powered embedded systems. *Transactions on Embedded Computing Systems*, 2003.

[SAMR03] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: frequency-aware static timing analysis. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[SCT10] A. Schranzhofer, Jian-Jian Chen, and L. Thiele. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *IEEE Transactions on Industrial Informatics*, 6(4), Nov 2010.

[SJJ13] Pagani Santiago and Chen Jian-Jia. Single frequency approximation scheme for energy efficiency on a multi-core voltage island. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA, 2013.

[SKL01] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy, hard real-time applications. *Design & Test*, 18(2), 2001.

[SL08] Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *Transactions on Embedded Computer Systems*, 7(3), 2008.

[SMP+10] Luca Santinelli, Mauro Marinoni, Francesco Prosperi, Francesco Esposito, Gianluca Franchino, and Giorgio Buttazzo. Energy-aware packet and task co-scheduling for embedded systems. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, 2010.

[SPG02] D Soudris, C Piguet, and C Goutis. *Designing CMOS Circuits for Low Power*. European Low-Power Initiative for Electronic System Design. Springer, 2002.

[Spu96] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical report, INRIA Rocquencourt, 1996.

[SR08] Saowanee Saewong and Raj Rajkumar. Coexistence of real-time and interactive & batch tasks in dvs systems. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[SR12] Sonal Saha and Binoy Ravindran. An experimental evaluation of real-time dvfs scheduling algorithms. In *Proceedings of the IEEE International Conference on Systems and Storage (SYSTOR)*, 2012.

[TOS] Toshiba's web site. http://www.toshiba.com.

[WB08] Shengquan Wang and Riccardo Bettati. Reactive speed control in temperature-constrained real-time systems. *Real-Time Systems*, 39(1-3), 2008.

[WBL] World bank. http://www.worldbank.org/.

[XMM05] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy in real-time embedded systems. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, 2005.

[XMM07] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *Transactions on Computer Systems*, 25(4), 2007.

[XXMM04] Ruibin Xu, Chenhai Xi, Rami Melhem, and Daniel Moss. Practical pace for embedded systems. In *Proceedings of the ACM international Conference on Embedded Software (EMSOFT)*, 2004.

[YBB09] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.

[YCK05] Chuan-Yue Yang, Jian-Jia Chen, and Tei-Wei Kuo. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, 2005.

[YCK07] Chuan-Yue Yang, Jian-Jia Chen, and Tei-Wei Kuo. Preemption control for energy-efficient task scheduling in systems with a dvs processor and non-dvs devices. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[YDS95] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 374–382. IEEE Computer Society, 1995.

[YPH+09] Jun Yi, Christian Poellabauer, Xiaobo Sharon Hu, Jeff Simmer, and Liqiang Zhang. Energy-conscious co-scheduling of tasks and packets in wireless real-time environments. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009.

[ZA09a] Baoxian Zhao and H. Aydin. Minimizing expected energy consumption through optimal integration of dvs and dpm. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2009.

[ZA09b] Dakai Zhu and Hakan Aydin. Reliability-aware energy management for periodic real-time tasks. *Transactions on Computers*, 58(10), 2009.

[ZAZ12] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. Energy management under general task-level reliability constraints. In *Proceedings of the IEEE International Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012.

[ZB09] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with edf scheduling. *Transactions on Computers*, 58, 2009.

[ZC04] Fan Zhang and Samuel T. Chanson. Blocking-aware processor voltage scheduling for real-time tasks. *Transactions on Embedded Computing Systems*, 3(2), 2004.

[ZM05] Yifan Zhu and Frank Mueller. Feedback edf scheduling of real-time tasks exploiting dynamic voltage scaling. *Real-Time Systems*, 31, 2005.

[ZSA11] Bo Zhang, Robert Simon, and Hakan Aydin. Maximum utility rate allocation for energy harvesting wireless sensor networks. In *Proceedings of the ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM)*, 2011.

INSTITUTE
OF COMMUNICATION,
INFORMATION
AND PERCEPTION
TECHNOLOGIES

Scuola Superiore
Sant'Anna