Università di Pisa

Ph.D. Thesis

Dynamic Voltage Scaling for Energy-Constrained Real-Time Systems

Claudio Scordino
scordino@di.unipi.it

Supervisor
Prof. Giuseppe Lipari
ReTiS Lab
Scuola Superiore Sant'Anna

Supervisor
Prof. Susanna Pelagatti
Dipartimento di Informatica
Università di Pisa

October 12, 2007

Dipartimento di Informatica
Largo B. Pontecorvo, 3
56127 Pisa Italy

Abstract

The problem of reducing energy consumption is dominating the design of several real-time systems, from battery-operated embedded devices to large clusters and server farms. These systems are typically designed to handle peak loads in order to guarantee the timely execution of those computational activities that must meet predefined timing constraints. However, peak load conditions rarely happen in practice, and systems are underutilized most of the time.

Modern microprocessors allow to balance computational speed versus energy consumption by dynamically changing operating voltage and frequency. This technique is called Dynamic Voltage Scaling (DVS). On real-time systems, however, if this technique is not used properly, some important task may miss its timing constraints. The goal of the operating system scheduler becomes, thus, to select not only the task to be scheduled, but also the processor operating frequency in order to reduce the energy consumption while meeting real-time constraints.

This thesis is the result of three years of research on the Resource Reservation and Energy-Aware Scheduling topics. The main contributions of this thesis can be summarized as follows. We study the problem of energy minimization from an analytical point of view and we propose a novel result in the real-time literature which integrates the concept of probabilistic execution time within the framework of energy minimization. In particular, we find the optimal value of the instant of frequency transition and of the speed assignment in a two-speed scheme where probabilistic information about task execution times is known. Unlike similar results presented in the literature so far, the optimal values are found using a very general model for the processor that accounts for the idle power and for both the time and the energy overheads due to voltage/frequency transition.

This thesis also includes a study about the Resource Reservation technique [104]. In particular, we investigate some anomalies in the schedule generated by the CBS [6] and GRUB [76, 75, 77] algorithms and we propose a novel algorithm, called HGRUB [11], which maintains the same features of CBS and GRUB but it is not affected by the problems described.

Finally, we develop a novel energy-aware scheduling algorithm called GRUB-PA which, unlike most algorithms proposed in the literature, allows to reduce energy consumption on

real-time systems consisting of *any* kind of task — i.e., hard and soft, periodic, sporadic and even aperiodic tasks. The effectiveness of the algorithm is validated through the formal proof of its main properties and through a series of comparisons with the state of the art of energy-aware scheduling algorithms using an Open Source simulator.

Last but not least, we describe a working implementation of the GRUB-PA algorithm in a real test-bed running the Linux operating system and we present a series of experiments to show that the algorithm actually reduce the energy consumption of the system.

Contents

List of Figures

List of Tables

Introduction

The number of embedded systems operated by battery is increasing in several application domains [70], from Personal Digital Assistants (PDAs) to autonomous robots, laptop computers, smart phones and sensor networks [35]. The problem of reducing the energy consumed by these systems has become a key design issue, since they can only operate on the limited battery supply. Many of these systems, in fact, are powered by rechargeable batteries and the goal is to extend as much as possible the autonomy of the system. Battery lifetime is a critical design parameter for such devices, directly affecting system size, weight and cost. Battery technology is improving rather slowly and cannot keep up with the pace of modern digital systems. Thus, reducing the energy consumed by these devices is the only way to prolong their lifetime.

In recent years, as the demand for computing resources has rapidly increased, even real-time servers and clusters are facing energy constraints [71, 27]. In fact, the growth of computational speed in current digital systems is mostly obtained by reducing the size of the transistors and increasing the clock frequency of the main processor. Since the power consumption is related to the operating frequency, the net effect is a growth of the energy demand and (as a side effect) of the heat generated [99, 63, 81]. Conventional computers are currently air cooled, and manufacturers are facing the problem of building powerful systems without introducing additional techniques such as liquid cooling [71, 24]. Cooling is a complex phenomenon that cannot be modelled accurately by a simple model [62, 114]. Heat dissipation directly affects the packaging and cooling solutions for integrated circuits. With power densities increasing due to increasing performance demands and tighter packing, proper cooling becomes even more challenging.

Clusters with high peak power need complex and expensive cooling infrastructures to ensure the proper operation of the servers. Not surprisingly, a significant portion of the energy consumed is due to the cooling devices, which may consume up to 50 percent of the total energy in small commercial servers [71, 90]. Thus, electricity cost represents a significant fraction of the operation cost of data centers [24, 27]. For example, a 10kW rack consumes about 10MWh a month (including cooling), which is at least 10 percent of the operation cost [20], with this trend likely to increase in the future. Clearly, good energy management is becoming important for all servers. To better understand the impact of

techniques for reducing power consumption in high-end servers, consider the cost savings that can be obtained by reducing the energy consumed in large web server farms, in terms of air conditioning and cooling systems: reducing the energy consumed by the computing components would impact on the energy consumed by the cooling devices and, in the end, on the total cost of the system.

In order for all these systems to be active for long periods of time, energy consumption should be reduced to an absolute minimum through energy-aware techniques. For this reason, the current generation of microprocessors [93, 131, 58, 57, 59, 60, 55] allow the operating system to dynamically vary voltage and operating frequency to balance computational speed versus energy consumption. This technique is called Dynamic Voltage Scaling (DVS) [99] and is used by many energy-aware scheduling policies already proposed in the literature [112, 98, 18, 101, 141].

In real-time systems and time-sensitive applications, however, if this frequency change is not done properly, some important task may miss its timing constraints. The problem is even more difficult considering that in practice most systems consist of a mixture of critical (i.e., *hard*) and less critical (i.e., *soft*) real-time tasks. For these reasons, developers typically design these systems so that they provide the highest computational power in any circumstance, in order to guarantee the timely execution of those computational activities that must meet predefined timing constraints. However, peak load conditions rarely happen in practice, and system resources are underutilized most of the time. For example, server loads often vary significantly depending on the time of the day or other external factors. Researchers at IBM showed that the average processor use of real servers is between 10 and 50 percent of their peak capacity [27]. Thus, much of the server capacity remains unused during normal operations. These issues are even more critical in embedded clusters [138], typically untethered devices, in which peak power has an important impact on the size and the cost of the system. Examples include satellite systems or other mobile devices with multiple computing platforms, such as the Mars Rover and robotics platforms. Some studies have observed that the actual execution times of tasks in real-world embedded systems can vary up to 87 percent with respect to their measured worst case execution times [134].

We believe that the advantages of DVS can be exploited even in real-time systems, through a careful identification of the conditions under which we can safely slow down the processor speed without missing any predefined timing constraint. This way, the reduction of the power consumption does not affect the timely execution of important computational activities. In particular, an *energy-aware scheduling algorithm* can exploit DVS by selecting, at each instant, both the task to be scheduled and the processor's operating frequency.

This thesis is the result of three years of research on the Resource Reservation and

Energy-Aware Scheduling topics. We focus on algorithms for *uniprocessor* systems, since the lack of optimal scheduling algorithms for real-time multiprocessors makes the problem of creating energy-aware algorithms with high efficiency very difficult.

The main contributions of this thesis can be summarized as follows. We start by studying the problem of energy minimization from an analytical point of view. We propose a novel result in the real-time literature which integrates the concept of probabilistic execution time within the framework of energy minimization. In particular, we find the optimal value of the instant of frequency transition (i.e., *transition point*) and of the speed assignment in a two-speed scheme where probabilistic information about task execution times is known. Through this study we show that saving energy while still meeting real-time constraints is actually possible. Unlike similar results presented in the literature so far, the optimal values are found using a very general model for the processor that accounts for the idle power and for both the time and the energy overheads due to voltage/frequency transition. Our result represents a net improvement over the sub-optimal mechanism used in the DVS-EDF algorithm proposed by Zhu et al. [43, 141].

This thesis also includes a study about the Resource Reservation technique [104]. In particular, we investigate some anomalies in the schedule generated by the CBS [6] and GRUB [76, 75, 77] algorithms and we propose a novel algorithm, called HGRUB [11], which maintains the same features of CBS and GRUB but it is not affected by the problems described.

Then, starting from the GRUB algorithm proposed by Lipari and Baruah [76, 75, 77], we develop a novel energy-aware scheduling algorithm called GRUB-PA which, unlike most algorithms proposed in the literature, allows to reduce energy consumption on real-time systems consisting of *any* kind of task — i.e., hard and soft, periodic, sporadic and even aperiodic tasks. With this algorithm we show that saving energy while meeting real-time constraints is possible even in presence of a mixture of hard and soft real-time tasks. The effectiveness of the algorithm is validated through the formal proof of its main properties and through a series of comparisons with the state of the art of energy-aware scheduling algorithms using the *RTSim* 0.3 [96, 4] scheduling simulator.

Last but not least, we describe a working implementation of the GRUB-PA algorithm in a real test-bed running the Linux operating system and we present a series of experiments to show that the algorithm actually reduce the energy consumption of the system.

The main results of this thesis have been already published at several conferences [111, 112, 110, 107, 78, 11] and on the *IEEE Transactions on Computers* journal [113], and have been accepted for publication on a special issue of the *International Journal of Embedded Systems* (IJES) journal [26].

This thesis is organized as follows. In Chapter 1, we introduce definitions and characteristics concerning real-time systems as well as the scheduling model that will be used

throughout the rest of this thesis. The interested reader can refer to Appendix A for an overview of some working implementations of real-time operating systems (RTOSs) based on Linux. In Chapter 2, we describe some architectural aspects concerning the DVS technique for CMOS circuits. Then, we propose a taxonomy of energy-aware scheduling algorithms and we provide an overview of the state of the art of the algorithms proposed in the real-time literature. In Chapter 3, we study the problem of energy minimization from an analytical point of view, finding the optimal values of transition point and speed assignments when probabilistic information about task execution times is known. The optimal values are found using a very general model for the processor that accounts for idle power and for both the time and the energy overheads due to voltage/frequency transition. In Chapter 4, we recall some concepts about the Resource Reservation technique [104] and we propose the GRUB and HGRUB algorithms that effectively solve some drawbacks of the original Constant Bandwidth Server (CBS) algorithm [6]. These algorithms will then be used as basis for our energy-aware scheduling algorithm GRUB-PA, presented in Chapter 5, which reduces energy consumption on real-time systems consisting of hard and soft, periodic, sporadic and aperiodic tasks. Finally, in Chapter 6 we state our conclusions, providing an overview of the ongoing and future work.

Acknowledgments

First of all, I would like to thank Jesus Christ (my life and my peace) for making all of this possible.

Thanks with all my love to Giulia, my wife, for her love and her patience during the trips back and forth from the United States.

My special thanks are for my two supervisors, namely prof. Giuseppe Lipari and prof. Susanna Pelagatti, for their precious help and their important suggestions. Working with you was really a honor!

Thanks to Enrico Bini, co-author of Chapter 3 and master of the LaTeX language.

Many thanks to prof. Daniel Mossé, Alexandre P. Ferreira, Kirsten Ream and all the people at the University of Pittsburgh. Studying at Pitt has been a wonderful experience and I hope to have the chance of meet all of you again and soon.

Thanks to prof. Hakan Aydin for his help in the development of the code of the DRA algorithm in the *RTSim* environment [16, 4].

Thanks also to my colleague Paolo Gai who gave me the exciting opportunity of working at *Evidence*.

Last but not least, many thanks to Luca Abeni and Michael Trimarchi for their precious teaching about Linux kernel internals.

To Giulia, my princess and my wife.

Chapter 1

Real-Time Systems

This chapter introduces basic terminology, definitions and notation concerning real-time systems as well as the scheduling model that will be used throughout the rest of this thesis. The interested reader can refer to Appendix A for an overview of some existing implementations of real-time systems based on Linux.

1.1 Introduction to Real-Time Systems

Real-time systems are *"systems in which the correctness depends not only on the logical result of the computation, but also on the time at which the results are produced"* [125, 116]. Thus, a system can be defined real-time if *"it produces the results within a finite and predictable interval of time"*, which is not necessarily the fastest time possible.

The system controlling the speed of a train is an example of real-time system: once an obstacle has been detected, the action of activating the brakes must be performed within a maximum delay, otherwise the train will crash on the obstacle. Keeping the previous example in mind, a real-time system can be more precisely defined as follows:

Definition 1 *[32] A real-time system is a computing system in which computational activities must be performed within predefined timing constraints.*

Typically, a real-time system is a controlling system managing and coordinating the activities of some controlled environment (see Figure 1.1). The real-time system acquires information about the environment through some sensors [35], and controls the environment using actuators. The adjective "real" means that the clock of the environment and the clock of the real-time system are synchronized. The environment creates some events, and the real-time control system must respond to these events *within a finite and predictable period of time*. The interaction is bidirectional, and is characterized by timing correctness constraints: the control — i.e., the reaction to internal or external events — must be done within finite and pre-established delays. Since a real-time system must

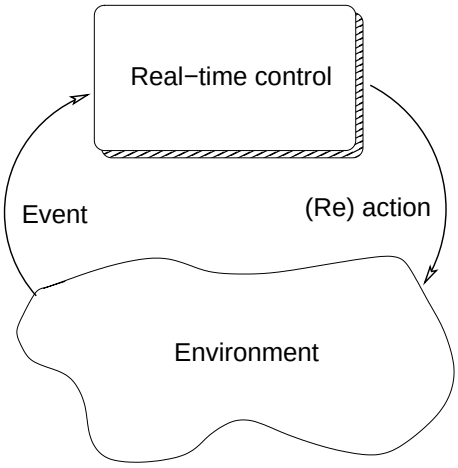

Figure 1.1: Real-time control system.

provide guarantees about its response times, it must have a predictable and deterministic timing behaviour.

Notice that the concept of "real-time" is not synonymous of "fast": having the response of the system within the timing requirements is sufficient. The objective of fast computing is to minimize the average response time of a given set of tasks. The objective of real-time computing, instead, is to meet the individual timing requirements of each task. Rather than being fast (which is a relative term), the most important goal in real-time system design is *predictability* — that is, the functional and timing behaviour of the system must be as deterministic as necessary to satisfy system specifications. This determinism allows to provide timing guarantees about the evolution of the environment. The fastness of the system (i.e., a low latency) remains in any case a desirable feature, since it allows to respond in a short time to events that need immediate attention [116] — fast computing is helpful in meeting stringent timing specifications, but it does not guarantee any predictability.

In the last decades, real-time systems have started to be used in different areas of everyday life. Nowadays, real-time capabilities can be found in several electronic devices, from simple PDAs for audio and video streaming to complex critical systems for the control of nuclear power plants. Examples of use of a real-time system include the control of laboratory experiments, car engines, nuclear power plants, chemical stations, flight systems, space shuttle and aircraft avionics, robotics and rocket systems [126, 32, 116]. Real-time systems are also widely used in many areas of telecommunications and multimedia, to guarantee timely execution of applications and proper Quality of Service (QoS) to end-users.

1.2 Design Issues

Although in the last decades, a strong mathematical theory has been successfully developed to model and formalize the behaviour of a real-time system [126, 32, 79], sometimes the development of such systems is still done in empirical ways, using heuristic assumptions. In other cases, the real-time system is sized according to the worst case scenario, resulting in a partial utilization of the available resources (e.g., the processor).

Some designers consider that a fast enough system is always able to respond in time. If an infinitely fast computer was available, we would not have any real-time problem, because the system would be capable of immediate response to any event, regardless of the current workload. However, even if advances in hardware technology will likely exploit faster processors to improve system throughput, this does *not* mean that timing constraints will be automatically met. Moreover, we must consider that the greater is the computational power provided by the hardware, the greater is the amount of resources needed. The amount of available computational power, in fact, affects many different aspects of the system, like the cost, the size, the energy consumption, the heat and noise produced and the fault robustness. For instance, the growth of computational power in current microprocessors is mostly obtained by reducing the size of the transistors in order to increase the clock frequency. This leads to both greater power consumption and greater heat production. Heat dissipation directly affects packaging and cooling solutions for integrated circuits, which, in turn, affect the size and the cost of the whole system.

This problem is critical especially in the field of *"embedded"* systems. These are special-purpose computers part of larger systems or machines that may not be of electronic kind (think, for instance, to the embedded system controlling an automotive engine). Typically, an embedded system is expected to work without human intervention, and it is housed on a single microprocessor board with the programs stored in ROM. In these systems, there is the need of reducing the resources used by the system as much as possible — brute force techniques do not scale to meet the requirements of guaranteeing real-time constraints on embedded devices. For instance, the development of operating systems for embedded devices starting from those for higher level architectures has often collided with the limited number of available resources. In an embedded system, the trade-off between the provided computational power and the amount of resources needed is very important. In particular, parameters like cost, size and *energy consumption* are decisive factors for the actual usability of such systems. Notice that the cost and the size are often conflicting goals, since the development of smaller devices is typically more expensive. Moreover, in embedded real-time systems there is the intrinsic trade-off between the importance to pursue such aims, and the need to have enough computational power to guarantee the real-time constraints.

Hence, it is very important to improve the efficiency by restricting the amount of re-

sources needed to the minimum possible value. During the last decades several formal approaches have been proposed to formalize the behaviour of a real-time system exactly [31]. However, they have been seldomly used in the design of real systems.

Currently, real-time system design is mostly ad hoc. This does not mean, however, that a scientific approach is not possible: most good science grew out of attempts to solve practical problems. For instance, the first flight of the space shuttle was delayed, at considerable cost, because of a timing bug that arose from a transient processor overload during system initialization [125]. The development of a scientific basis for verifying that a design is free of such bugs is clearly necessary.

1.3 Real-Time Operating Systems

Typically, a real-time system is implemented as a set of concurrent tasks that are executed on a Real-Time Operating System (RTOS). Each task represents a computational activity that needs to be performed according to a set of constraints. The objective of the RTOS is, thus, to manage and control the assignment of the system resources (e.g., the microprocessor) to the tasks in order to meet such constraints.

Definition 2 *[3] A real-time operating system (RTOS) is an operating system capable of guaranteeing the timing requirements of the tasks under its control through some hypothesis about their behaviour and a model of the external environment.*

"RTOS" is a generic term for a set of operating systems providing some kind of support for real-time applications. There is a wide range of RTOSs, from the small and simple system that fits in few kilobytes of memory and can run on simple processors, to the high-end RTOS providing full graphical user interface and requiring several megabytes of RAM and powerful processors (with MMU, protected mode, etc.).

A real-time system should be flexible enough to react to a highly dynamic and adaptive environment, but at the same time it should be able to avoid resource conflicts so that timing constraints can be predictably met. The environment may cause any unpredictable combination of events to occur, but the real-time system should be carefully built in order to predict the possibility of meeting timing constraints at *any* time during execution.

Desirable features of a RTOSs are the co-existence of normal and real-time tasks, low latency, and fault tolerant mechanisms [6, 48, 9, 8, 32]. Appendix A explains how these goals can be achieved in Linux in order to create a real-time operating system.

1.4 Real-Time Tasks

Real-time scheduling involves the allocation of system resources to tasks in such a way that certain predefined timing requirements are met. Scheduling has been probably the

most widely researched topic in the real-time literature [126].

Definition 3 *[126] A real-time task is an executable entity of work which is characterized, at least, by a worst case execution time and a timing constraint. It denotes a generic scheduling entity, that can correspond to either a thread or a process on a real operating system.*

Depending on the type of application, different timing constraints (like the jitter on the initial or the finishing time of execution) can be defined. A typical constraint is the *deadline*, which is the instant of time the task's execution is required to be completed. A real-time task must complete before its deadline, otherwise the results could be produced too late to be useful. The deadline is the only timing constraint that we will consider throughout this thesis.

Typically, real-time tasks have a cyclic structure in which they execute some code and then they block waiting for a timer or for a particular event. For this reason, real-time tasks can be well modelled by thinking each task as consisting of a sequential stream of *"jobs"*.

Definition 4 *[126] A job is an instance of a real-time task.*

The job is the unit of work, scheduled and executed by the operating system. We say that a job arrives when the task unblocks (i.e., when the task becomes ready for execution), and that the job ends when the task blocks. Jobs of the same task must be executed sequentially — i.e., the concurrent execution of jobs of the same task is not possible.

Typically, tasks have variable execution times between different jobs. The execution time, in fact, depends on several factors like input data, current state of cache and processor pipeline, etc. As an example, consider the two branches of a typical `if-then-else` statement, which may have very different computation times.

There are many real-time applications in which the worst-case scenario happens very rarely, but its duration is much longer than the average case. Some studies have observed that the actual execution times of tasks in real-world embedded systems can vary up to 87 percent with respect to their measured worst case execution times [134]. An algorithm with highly variable computation time is the decoding of a MPEG frame, where the execution time depends on the data contained in the frames.

The problem of variable execution times is typically handled in the real-time literature by defining a parameter of the task called Worst Case Execution Time (WCET), which represents the maximum possible value that the computation times of task's jobs can assume. Another parameter, called Worst Case Execution Cycles (WCEC), represents the

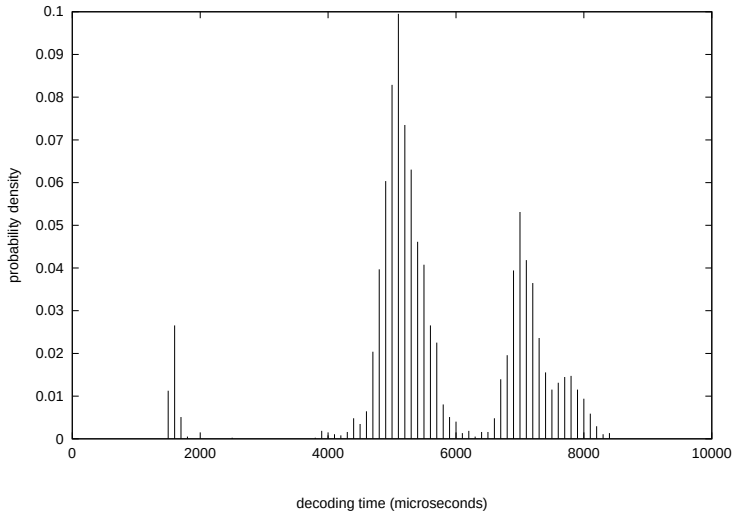

Figure 1.2: Distribution of execution times of decoding the *Star Wars* movie.

maximum possible processor cycles required by the task instances, and is typically used when dealing with processors having dynamic speed.

Computing the exact value of the WCET of a task consists on computing the execution time of all the possible paths that the program may follow, then selecting the path with the maximum execution time. This computation is possible only in principle. In practice, the problem is intractable even for deterministic programs: the presence of caches, pipelines, DMA and branch prediction makes the problem of computing a constant execution time very difficult. In fact, although these mechanisms reduce the average execution time of a task, they make much more difficult the estimation of worst case execution times [32]. Moreover, the worst case path may never happen in practice (because not possible). Thus, the value estimated for the WCET in practice is affected by large errors (typically, more than 20 percent [32]). Therefore, an interesting property of the scheduling discipline is the ability of reclaiming resources used by tasks that execute less than their worst case requirements. As we will see in Chapter 5, our scheduling algorithm, GRUB-PA, has such interesting feature.

A more complete and precise information about the behaviour of a dynamic computational activity like a real-time task is the probability density function (p.d.f.) derived by experimental data. Figure 1.2 shows the probability density function of the execution time of the job decoding a frame of the *Star Wars* movie [32]. The x axis shows the decoding time (expressed in microseconds) whereas the y axis shows the probability density function. In Chapter 3, we will study the problem of reducing energy consumption in real-time systems where probabilistic information about execution times is known.

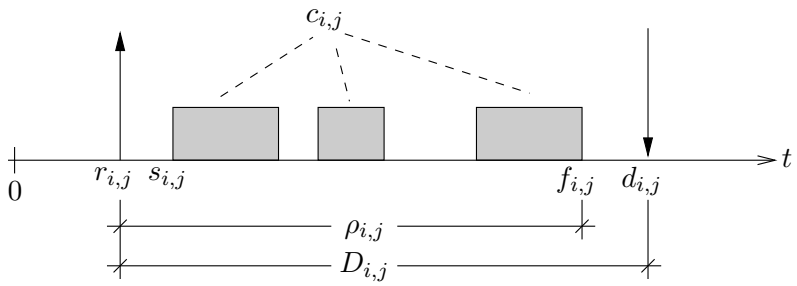

Figure 1.3: A GANNT chart describing the parameters of a generic job $\tau_{i,k}$.

1.5 Scheduling Model

We consider the processor as the only resource shared by a set of real-time tasks, reducing the scheduling problem to the choice of a possible assignment of the tasks to the processor. Furthermore, we restrict our attention to preemptive uniprocessor systems, with the assumption that processing is fully preemptible at any point. Thus, at any instant of time, the running task is the highest priority active task in the system. If a low priority task is in execution and a higher priority task becomes ready for execution, the former is *preempted* and the processor is given to the new arrival. This assumption is realistic on modern operating systems like Linux which, with the advent of the 2.6 kernel series, has become a fully preemptive OS [82] (refer to Appendix A.4.1 for more details).

We consider a system comprised of n real-time tasks τ_1, τ_2,..., τ_n. We refer to this set of tasks as the *task set* $\tau = \{\tau_1, ..., \tau_n\}$. Each task τ_i is a sequential stream of *jobs* (or instances) $\tau_{i,1}$, $\tau_{i,2}$, $\tau_{i,3}$, ..., where $\tau_{i,j}$ becomes ready for execution ("arrives") at time $r_{i,j}$ ($r_{i,j} \leq r_{i,j+1} \quad \forall i, j$), and requires a computation time equal to $c_{i,j}$ units of time. Real-time tasks can be well described through a GANNT chart (see Figure 1.3) having an horizontal time axis for each task. The assignment of jobs to the processor is represented by filled rectangular boxes drawn along the axes. Typically, capital letters are used to represent absolute values in time, whereas small letters are used for relative values.

In particular, a job $\tau_{i,k}$ is characterized by the following parameters:

Definition 5 (Release time) *The* release (or activation) time $r_{i,k}$ *is the instant of time at which the job becomes ready for execution (because it has been activated by some event or condition).*

In the GANNT chart the release time is typically represented by an upward arrow (see Figure 1.3).

Definition 6 (Start time) *The* start time $s_{i,k}$ *is the time at which the job starts its execution for the first time (i.e., the first time the processor is assigned to the job).*

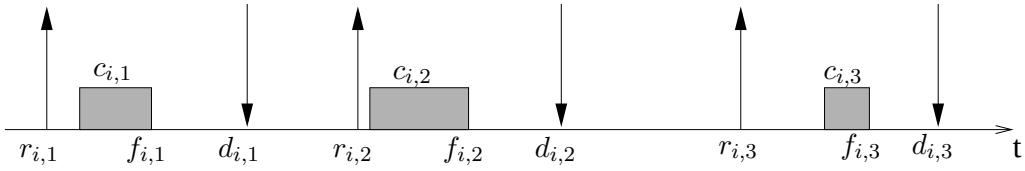

Figure 1.4: A GANNT chart describing a sequence of jobs $\tau_{i,k}$ belonging to the same task τ_i.

Definition 7 (Finishing time) *The* finishing (or completion) time $f_{i,k}$ *is the time at which the job actually completes its execution.*

Definition 8 (Relative deadline) *The* relative deadline $D_{i,k}$ *is the interval of time job's execution is required to be completed with respect to its release time. Typically, this is a constant value equal for all the jobs belonging to the same task, and it is called* task relative deadline D_i.

Definition 9 (Absolute deadline) *The* absolute deadline $d_{i,k}$ *is the absolute instant of time by which job's execution must complete. The deadline is successfully met if and only if* $f_{i,k} \leq d_{i,k}$*, otherwise it is missed. The job deadline* $d_{i,j}$ *is computed based on the relative deadline:* $d_{i,j} = r_{i,j} + D_{i,j}$.

In the GANNT chart the deadline is typically represented by a downward arrow (see Figure 1.3).

Definition 10 (Computation time) *The* computation time $c_{i,k}$ *is the time required by the processor to complete job's execution without interruption.*

Definition 11 (Response time) *The* response time $\rho_{i,k}$ *is the time elapsed from the job release time to the finishing time (*$\rho_{i,k} = f_{i,k} - r_{i,k}$*).*

We assume that, for each task, the jobs are executed in FIFO order — i.e., $\tau_{i,j+1}$ can start execution only after $\tau_{i,j}$ has completed. A GANNT chart describing a sequence of jobs belonging to the same task is shown in Figure 1.4.

A task τ_i is characterized by the following parameters.

Definition 12 (Worst case execution time) *The* worst case execution time *(WCET) of task* τ_i *is the worst (i.e., maximum) computation time required by all its instances:*

$$C_i = \max_{k \geq 0}\{c_{i,k}\} \tag{1.1}$$

Definition 13 (Worst case response time) *The* worst case response time *of task* τ_i *is the worst (i.e., maximum) response time required by all its instances:*

$$\rho_i = \max_{k \geq 0}\{\rho_{i,k}\} \tag{1.2}$$

1.6 Task Periodicity

Real-time tasks can either be activated by a timer at predefined instants of time or by the occurrence of a specific event or condition [32]. In the former case, an important characteristic of the task is given by the regularity of its activations (i.e., the release times of its jobs) which allows to distinguish between *periodic* and *aperiodic* tasks.

Definition 14 *Periodic tasks are tasks that are activated (released) at regular intervals of time. In particular, a task τ_i is said to be* periodic *if*

$$r_{i,k+1} = r_{i,k} + T_i \qquad \forall k \geq 1 \tag{1.3}$$

where T_i is the task period. *The deadline, if not otherwise stated, corresponds to the end of the period.*

Typically, periodic tasks have a regular structure, consisting of an infinite cycle, in which the task executes a computation and then suspends itself waiting for the next periodic activation. A typical periodic task has the following structure:

```
void * PeriodicTask(void *arg)
{
     <initialization>;
     <start periodic timer, period = T>;
     while (cond) {
          <read sensors>;
          <update outputs>;
          <update state variables>;
          <wait for next activation>;
     }
}
```

A typical characteristic of a periodic task set is a parameter called *utilization*, which is defined as follows

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1.4}$$

and represents the computational workload requested by the task set to the processor, assuming that each job executes for its worst-case execution time. This value is typically used to evaluate the schedulability of the task set (see Section 1.9).

Another class of real-time tasks, very similar to periodic tasks, is called *sporadic* and waits for aperiodic events.

Definition 15 *[126] Sporadic tasks are real-time tasks that are activated irregularly with some known bounded rate. The parameter T_i denotes the* minimum *(rather than exact)*

separation between successive jobs of the same task, and is called minimum interarrival time*:*

$$r_{i,k+1} \geq r_{i,k} + T_i \qquad \forall k \geq 1 \tag{1.5}$$

A typical sporadic task has the following structure:

```
void * SporadicTask(void *)
{
     <initialization>;
     while (cond) {
          <computation>;
          <wait event>;
     }
}
```

Finally, there is a class of real-time tasks for which it is not possible to set even a minimum interarrival time between two consecutive jobs. This kind of tasks, called *aperiodic*, does not have any particular timing structure, and typically responds to events that occur rarely (e.g., a state change) or that happen with an irregular structure (e.g., burst of packets arriving from the network).

Definition 16 *[126, 32] Aperiodic tasks are real-time tasks which are activated irregularly at some unknown and possibly unbounded rate. For this kind of tasks, the release time of the job $\tau_{i,k+1}$ is greater than or equal to that of the previous job $\tau_{i,k}$:*

$$r_{i,k+1} \geq r_{i,k} \qquad \forall k \geq 1 \tag{1.6}$$

1.7 Hard and Soft Real-Time Tasks

An inherent characteristic of real-time tasks is that the specification of their requirements includes timing information. A real-time task must complete before its deadline, otherwise the results could be produced too late to be useful. In safety critical applications, for instance, a deadline miss could result in serious consequences for the system. The importance of meeting timing constraints divides tasks into two classes: *Hard* and *Soft* real-time tasks.

Hard real-time tasks are those critical activities whose deadlines can never be missed, otherwise a critical system failure can compromise the functionality of the system: failure in meeting timing constraints is a fatal fault and it is as much an error as a failure in the value domain. In particular, a task can be defined as hard real-time when *"a deadline miss has the potential to be catastrophic"* [29] — i.e., the consequences are incommensurably greater than any benefits provided in absence of failure. This kind of task is typically used to control or monitor some physical device, and a missed deadline may cause catastrophic

consequences. For this reason, hard real-time systems cannot compensate some deadline miss in the worst case with a good performance in the average case, and need an a-priori study to ensure that all deadlines will be met under *any* possible conditions.

Hard real-time systems are designed under worst-case scenarios, by making pessimistic assumptions on system behaviour and on the external environment. This approach allows system designers to perform off-line analysis to guarantee that the system will be able to achieve the desired performance in all operating conditions that have been predicted in advance. However, the consequence of such worst-case design methodology is that high predictability is achieved at the price of a very low efficiency in resource utilization. Low efficiency means more memory and more computational power which, in turn, affect system costs [31].

Hard real-time tasks are needed in a number of application domains, including automotive, air-traffic, avionics, industrial, chemical, nuclear, safety-critical and military controls. Examples of hard real-time systems operated by batteries or by solar cells are autonomous robots operating in hazardous environments, like those sent by NASA for exploring the surface of Mars.

In most large real-time systems, not all computational activities are really hard or critical. For soft real-time tasks, the timing constraints are important but not critical, and the system will still work correctly if some deadline is occasionally missed (it does not compromise the functionality of the system, but there is some kind of degradation of the performance perceived by the user). Typically, the number of missed deadlines is related to the Quality of Service (QoS) provided by the application: a deadline miss does not compromise the correctness of the system, but its QoS degrades. An example is a real-time system guaranteeing a fixed QoS to each user accessing a shared resource. Typical requirements on soft real-time tasks are:

- no more than x consecutive missed deadlines;
- no more than x missed deadlines in an interval of time ΔT;
- the *deadline miss ratio* (i.e., percentage or total missed deadlines over the total number of deadlines) not exceeding a certain threshold;

In addition, with respect to hard real-time systems, these systems often operate in more dynamic environments, where tasks can be created or canceled at run-time, or task parameters can change from one job to the other.

Typical examples of soft real-time systems are multimedia (e.g., virtual-reality, interactive computer games, home-theaters, audio and video streaming) and telecommunication applications. In the last years, we noticed an incredible growth of interest in supporting multimedia applications (e.g., audio and video streaming) on general-purpose operating

systems. These applications are characterized by implicit, but not critical, timing constraints, which must be satisfied to provide the desired QoS. A classical example of soft real-time task is an MPEG player. The typical frame rate of a video is 25 Frame Per Second (FPS). If some frame is displayed with a little delay, the user may not even be able to perceive the effect. If frames are skipped or displayed too late, however, the disturbance becomes evident. Avoiding *any* delay may involve a costly hardware. Using a soft real-time task, instead, makes possible to allow some occasional delay without affecting the quality perceived by the user.

The distinction between hard and soft real-time systems is useful for a general discussion but, in practice, many real systems consist of a mixture of hard and soft real-time tasks. The objective is to guarantee that all hard real-time tasks will always complete before their deadlines and, at the same time, to maximize the QoS provided by soft real-time tasks. Clearly, respecting the timing constraints in such hybrid systems is even more difficult. The problem of mixing hard and soft real-time tasks can be efficiently solved by using the Resource Reservation framework [84, 85, 87, 86] that will be introduced in Chapter 4. In Chapter 5, we will present a novel energy-aware algorithm called GRUB-PA able of scheduling both hard and soft, periodic, sporadic and even aperiodic tasks respecting the timing constraints of each running application.

1.8 Other Constraints

Besides timing constraints (like the deadlines), many further constraints can be defined for real-time tasks executing on a RTOS. In particular, in real RTOSs we often find the following kinds of constraints [31]:

- **Precedence constraints**. Depending on the specific application, it may be not possible to run the tasks in an arbitrary order (consider, for instance, an assembly line). Therefore, task precedence constraints need to be taken into account. A task τ_i is said to *precede* another task τ_j if τ_j can only start execution after τ_i has completed its computation. This kind of constraints can be effectively expressed through a DAG [1].

- **Constraints on shared resources**. Real-time tasks may require access to certain resources other than the processor, such as I/O devices, networking, data structures, files. When different tasks must access the same resource, it is important to keep the shared resource in a consistent state. If a task is interrupted while it is using a resource, in fact, the resource may remain in an inconsistent state. To solve this

[1] A Direct Acyclic Graph (DAG) is a pair $G = (T, P)$ where the vertices in T are tasks and the edges in P represent the precedence constraints. An edge (τ_i, τ_j) means that task τ_i must be completed before task τ_j can start execution.

problem the tasks must access the resource in *mutual exclusion*, so that at one time no more than one task has the permission to use the resource.

The implementation of mutual exclusion policies in real-time systems is difficult because it can lead to a problem known in the literature as *priority inversion* [117]. This undesired phenomenon creates unbounded delays in the real-time schedule, so that some important task may miss its deadlines.

A priority inversion happens when a high priority task waits for the lock held by a low priority task, which, in turn, has been preempted by a task with intermediate priority. Thus, the high priority task must wait the completion of both the intermediate priority task (which is blocking the low priority task) and the low priority task (which holds the lock on the shared resource). Waiting for an unbounded time, the high priority task may miss its deadline.

Priority inversion is one of the most critical problems in the development of software for real-time systems. An example of priority inversion occurred during the *Pathfinder* mission on Mars. In the operating system (VxWorks) a hidden semaphore shared between two tasks with different priorities was used to protect a pipe. Thus, the higher priority task missed its deadline whenever the task with lower priority was preempted by a task with intermediate priority [105].

The *Priority Inheritance* and *Priority Ceiling* protocols, first proposed by Sha et al. [117], solve the problem of unbounded priority inversion [136]. The idea is that the low priority task *inherits* the priority of the high priority task while holding the lock, preventing the preemption by medium priority tasks. In the Priority Ceiling protocol, for instance, a *priority ceiling* is assigned to each semaphore, which is equal to the priority of the highest priority task that *may* use the semaphore. Before any task τ_i enters a critical section, it must first obtain the lock on the semaphore S guarding the critical section. If the priority of the task τ_i is not higher than the highest priority ceiling among all semaphores currently blocked by the other tasks, then the lock on S is denied and the task τ_i blocked. When a task τ_i blocks higher priority tasks, it automatically inherits the highest priority of the tasks that it is currently blocking. When τ_i exits a critical section, it resumes the priority that it had before entering the critical section. Finally, a task τ_i not attempting to enter critical sections can only preempt tasks having lower (inherited or assigned) priority.

We have described these two kind of constraints for completeness. However, we will consider timing constraints (in particular, the deadline) as the only constraint in our scheduling model because we consider the processor as the only resource shared among our real-time tasks and we do not impose any precedence order among tasks.

1.9 Taxonomy of Scheduling Algorithms

The real-time scheduling theory addresses the problem of guaranteeing timing constraints in real-time systems.

Definition 17 *A* scheduling algorithm $\mathcal{A}$ *is an algorithm that for each instant of time* t*, selects a task to be executed on the processor among the ready tasks. The scheduling algorithm, applied to a specific task set* $\tau = \{\tau_1, ..., \tau_n\}$*, generates a* schedule $\sigma_{\mathcal{A}}(t)$*, which is a possible assignment of the processor to the jobs.*

A scheduling algorithm should be designed in such a way that the timing behaviour of the system is understandable, predictable and maintainable. Scheduling techniques can be classified according to several important characteristics [50] like:

- **Adaptation**: the ability of the scheduler to detect and adapt to any change in the application behaviour [7];

- **Predictability**: the ability to analyze the run-time behaviour by, for instance, estimating task's response time and verifying the timing constraints;

- **Complexity**: the volume of computation required to make scheduling decisions (e.g., $O(n)$, $O(nlogn)$, $O(n^2)$, and so on).

According to these properties, real-time scheduling algorithms can be distinguished into:

- **Static or dynamic priority.** A static priority, also called Fixed Priority (FP), algorithm assigns static priorities to the tasks off-line, but schedules them at run-time. The schedule itself is not fixed, but the priorities that drive the schedule are fixed [126]. Static algorithms require prior knowledge about the properties of the system, but yield little run-time overhead. *Rate Monotonic* (RM) and *Deadline Monotonic* (DM) are examples of FP algorithms for periodic tasks where the priorities are assigned according to tasks periods and to relative deadlines, respectively [116].

  Another important class of scheduling algorithms is the class of dynamic priority algorithms, where the priority of a task is computed at run-time and can change during task execution. Dynamic priority algorithms are typically more flexible than static priority ones but may suffer of some drawback, like *domino* effects in case of overload. Static priority algorithms, instead, are typically more predictable. The most important (and analyzed) dynamic priority algorithm is *Earliest Deadline First* (EDF) [79], where the highest priority is given to the task with earliest deadline — i.e., the priority of the job is inversely proportional to its absolute deadline.

- **Preemptive or non-preemptive.** In preemptive schemes, a low priority task may be suspended if a higher priority task is available for execution. Alternatively, in non-preemptive approaches, once started, each task finishes its execution without interruption from other tasks. Clearly, preemptive schemes are more flexible, but they also introduce some time overhead due to context switches. Intermediate approaches, like *deferred preemption*, exist to avoid preemption during critical time intervals.

- **Centralized or distributed.** Centralized algorithms are typically used in (uni- or multi-processor) systems with shared memory, where the communication overhead is negligible. Distributed algorithms, instead, are used in distributed systems where communications take a considerable time, which has to be considered during feasibility analysis.

1.10 Schedulability tests

The goal of system designers is to prove that all tasks (or, at least, a sufficient percentage of them) meet their deadlines.

Definition 18 *A task set τ is said to be* schedulable by the algorithm $\mathcal{A}$ *if, in the generated schedule $\sigma_{\mathcal{A}}(t)$, every job starts at or after its release time and completes before its deadline:*

$$r_{i,j} \leq f_{i,j} \leq d_{i,j} \qquad \forall i,j \tag{1.7}$$

In this case, the generated schedule is said to be feasible.

Definition 19 *A task set τ is said to be* schedulable *if there exist some algorithm that produces a feasible schedule.*

Definition 20 *[126] A scheduling algorithm $\mathcal{A}$ is* optimal *if every task set that is schedulable by another algorithm is schedulable also by algorithm $\mathcal{A}$.*

This definition of optimality is the typical one used in real-time scheduling [126, 32].

Definition 21 *A* schedulability test *for the algorithm $\mathcal{A}$ is an algorithm that, given a task set $\tau = \{\tau_1, ..., \tau_n\}$, returns* YES *if and only if the task set is schedulable by $\mathcal{A}$.*

For periodic tasks, a schedulability test is typically based on the parameter $U_{lub}^{\mathcal{A}}$ (which stands for *"utilization least upper bound"*), which is an intrinsic characteristic of the scheduling algorithm $\mathcal{A}$: if the total utilization U does not exceed the bound (i.e., $U \leq U_{lub}^{\mathcal{A}}$), then it is guaranteed that the task set is schedulable by the algorithm $\mathcal{A}$. When $U_{lub}^{\mathcal{A}} < U \leq 1$, instead, the task set may or may not be schedulable by the algorithm.

Notice that no existing algorithm can schedule task sets having $U > 1$, since it means that they are asking for more than 100 percent of processor usage.

In particular, for Rate Monotonic we have that [116]

$$U_{lub} = n(2^{1/n} - 1) \tag{1.8}$$

therefore, for large values of n, we have that

$$\lim_{n \to \infty} U_{lub} = 0.69 \tag{1.9}$$

This means that under Rate Monotonic the timing constraints are guaranteed only if the utilization is below a value which is rather far from the optimal total utilization (i.e., $U = 1$).

An important result in the real-time literature is the theorem stating that for the EDF algorithm $U_{lub} = 1$ [79]. This implies the optimality of the EDF algorithm: if a task set is schedulable using some algorithm, then it is schedulable also by EDF. In fact, if $U \leq 1$, then the task set is schedulable by EDF, otherwise the task set is not schedulable by any algorithm. In particular, notice that EDF can schedule all task sets that can be scheduled by fixed priority algorithms, but not vice versa. Energy-aware algorithms are typically based on dynamic priorities because they need to exploit system resources up to 100 percent.

1.11 Summary

In this chapter, we introduced the real-time scheduling theory. We explained why such theory is important, and why fast computing cannot guarantee the respect of timing constraints in any circumstance. Moreover, we provided the definitions, the notation and the scheduling model that will be used throughout the rest of this thesis.

In the next chapter, we will see how real-time and energy-saving objectives can be pursued at the same time through the use of energy-aware real-time scheduling algorithms.

Chapter 2

Energy-Aware Scheduling

In this chapter, we describe some architectural aspects concerning the Dynamic Voltage Scaling (DVS) technique for CMOS digital circuits. Then, we propose a taxonomy of energy-aware scheduling algorithms and we provide an overview of the state of the art of the algorithms proposed in the literature so far.

2.1 Energy Constrained Systems

The booming market share of embedded systems (like PDAs, autonomous robots, smart phones, sensor networks [35] and so on) has promoted energy efficiency as a major design goal [70]. Many of these systems, in fact, are powered by rechargeable batteries and the goal is to extend the autonomy of the system as much as possible. Battery lifetime is a critical design parameter for such devices, directly affecting system size, weight and cost. Battery technology is improving rather slowly and cannot keep up with the pace of modern digital systems.

In recent years, as the demand for computing resources has rapidly increased, even real-time servers and clusters are facing energy constraints [71, 27]. In fact, the growth of computational speed in current digital systems is mostly obtained by reducing the size of the transistors and increasing the clock frequency of the main processor. As we will show in the next sections, power consumption is related to the operating frequency. Thus, the net effect is a growth of the energy demand and (as a side effect) of the heat generated [99, 63, 81]. Clusters with high peak power need complex and expensive cooling infrastructures to ensure the proper operation of the servers and manufacturers are facing the problem of building powerful systems without introducing additional techniques such as liquid cooling [71, 24]. Moreover, cooling, and hence temperature, is a complex phenomenon that cannot be modelled accurately by a simple model [62, 114].

In order for these devices to be active for long periods of time, energy consumption should be reduced to an absolute minimum through *energy-aware techniques*. At the same

time, however, it is important to guarantee the timing constraints of the real-time applications. Many of these devices, for instance, use a soft real-time computation to ensure a proper Quality of Service (QoS) in telecommunications or in multimedia applications.

Irani et al. [62] wrote a general survey about the current research on several algorithmic problems related to power management (cooling of microprocessors included).

One of the most energy consuming resources in both embedded and high-end machines is the main microprocessor. For this reason, modern processors usually support several operating states with different levels of power consumption [93, 131, 58, 57, 59, 60, 55]. These operating states can be broadly categorized as

- **active**, in which the processor continues to operate, but possibly with reduced performance and power consumption. Processors might have a range of active states with different frequencies and power characteristics;

- **idle**, in which the processor is not operating. Idle states vary in both power consumption and latency for returning the processor to an active state.

2.2 Dynamic Voltage Scaling

2.2.1 CMOS Microprocessors

Nowadays, most digital devices are implemented using Complementary Metal Oxide Semiconductor (CMOS) circuits. Power consumption of this kind of circuits can be modelled accurately with simple equations [63, 99, 50, 70]. Like in other kind of circuits, in CMOS circuits power consumption can be splitted in two main components:

$$P_{CMOS} = P_{static} + P_{dynamic} \tag{2.1}$$

where P_{static} and $P_{dynamic}$ are the static and dynamic components of power consumption, respectively.

In the ideal case, CMOS circuits do not dissipate static power (i.e., $P_{static} = 0$) since, in steady state, there is no open path from source to ground. In reality, bias and leakage currents through the MOS transistors cause a static power consumption which is a (usually) small portion of the total power consumed by the circuit.

Dynamic power consumption in CMOS circuits is dissipated during the transient behaviour (i.e., during switches between logic levels). Every transition of a digital circuit consumes power, because every charge or discharge of the digital circuit's capacitance drains power. Dynamic power consumption is equal to

$$P_{dynamic} = \sum_{k=1}^{M} C_k \cdot f_k \cdot V_{DD}^2 \tag{2.2}$$

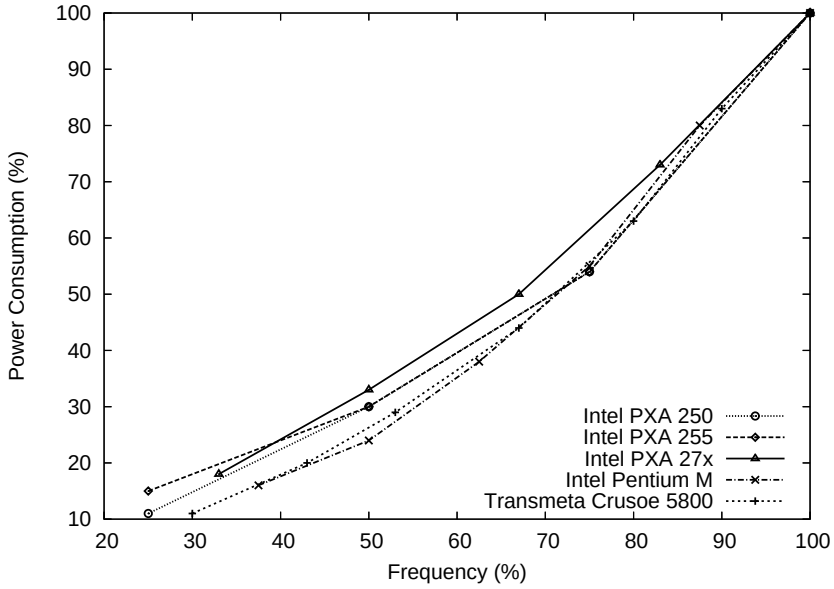

Figure 2.1: Normalized power consumption of well-known microprocessors.

where M is the number of gates in the circuit, C_k is the load capacitance of the gate g_k, f_k is the switching frequency of g_k per second, and V_{DD} is the supply voltage.

If we assume that the dynamic component is the most dominant one [70], we can associate a power consumption

$$P_{CMOS} \propto f \cdot V_{DD}^2 \tag{2.3}$$

to the clock frequency f of the microprocessor, as done in [50, 99, 63]. Although not exact, this is the most used model in the real-time literature for the evaluation and the comparison of energy-aware scheduling algorithms and, if not otherwise stated, it is the model that we will assume throughout this thesis. In Figure 2.1, we show the normalized value of the power consumption of some well-known industrial microprocessors. The resulting values have been obtained by applying Equation 2.3 to the values taken from the processors datasheets [130, 131, 58, 57, 59, 60, 56].

Notice that, although the static power is today about two orders of magnitude smaller than the total power, the typical chip's leakage power increases about five times each generation, and it is expected to significantly affect, if not dominate, the overall energy consumption in integrated circuits. Even if leakage power can be substantially reduced by cooling, this model of the power consumption may be not suitable for next-generations microprocessors [50]. For this reason, in Section 3.2.1, we will introduce a general model for the processor power consumption which considers each mode as a separate entity, and does not assume any relationship among the energy consumption of each level.

2.2.2 Processor Speed

For many applications, the microprocessor is the bottleneck of the system and the main used resource, therefore the assumption that the speed of the running task corresponds to the speed of the processor is quite natural. Thus, when considering the relationship between the microprocessor frequency and the task computation time, we can make the (worst-case) assumption that the number of processor cycles required by the task is constant (i.e., independent of the processor speed α) and that a change of the processor frequency does not affect the worst case execution cycles (WCEC) of the task. This assumption is also justified by the experiment described in Section 5.4.2, showing some experimental results on a real test-bed embedded system.

However, if we want a more accurate model of the system, we must consider that the processor is *not* the only resource involved in the computation: even very simple applications need to access some peripheral (like memory, disk, network card, etc.) through an external bus. Bus frequencies generally diverge from internal processor frequencies, and they do *not* scale at the same rate as processor does. Since bus access time often limits the performance of data-intensive applications, running the task at reduced processor frequency has a limited impact on performance. This makes the assumption of the task speed corresponding to the processor speed a worst-case model. In particular, this assumption holds only for systems where the memory latency can scale with processor frequency (mainly, systems with on-chip memory). In contrast, for a system where the memory latency does not scale with processor frequency (systems with dynamic memory and memory hierarchies), the WCEC of a task *does not* remain constant when the processor frequency scales. In these systems, in fact, there is a constant access latency for memory references, and an increase of the processor frequency increases the number of cycles required to access the memory. Of course, this effect can be relieved using good caches. However, even if we do not assume perfect caches, it is possible to extend the model accounting for the total number of cache misses for the task.

Recently, some frequency models to express WCEC bounds as parametric terms whose components are frequency-sensitive parameters have been proposed in the literature [115, 25]. In these models, cycles are interpreted in terms of the processor frequency, whereas memory accesses are expressed in terms of the memory latency overhead due to the external bus speed. Essentially, the execution time C_i of task τ_i is splitted into two components:

$$C_i^{\alpha} = \frac{C_i}{\alpha} + m_i \tag{2.4}$$

where α is the processor speed (expressed in cycles per second, and typically comprised between 0 and 1), C_i (expressed in processor cycles) scales with the clock frequency, and m_i (expressed in seconds) does not scale.

Clearly, the amount of computation time that varies with the processor frequency depends on the particular task and can be different for tasks running on the same system. This more accurate model has been described for completeness, but it will not be used in the rest of this thesis. In fact, we are not interested in an accurate model of the task speed, but rather in a comparative analysis among different energy-aware algorithms.

2.2.3 The DVS Technique

From Equation 2.3, it follows that reducing V_{DD} is the most effective way to lower the power consumption. This technique is known as *Dynamic Voltage Scaling* (DVS). Many modern processors [93, 131, 58, 57, 59, 60, 55] can dynamically lower the voltage to reduce the power consumption. Unfortunately, a reduction of the power supply voltage causes an increase of the circuit delay. In turn, the propagation delay restricts the clock frequency of the microprocessor: *the processor can operate at a lower supply voltage, but only if the clock frequency is reduced to tolerate the increased propagation delay.* Thus, in most cases, when reducing the supply voltage it is necessary to lower also the operating frequency (i.e., the microprocessor speed). As a consequence, all tasks will take more time to be executed. In real-time systems, if this frequency change is not done properly, the timing requirements of the application cannot be respected. Therefore, the advantages of the DVS technique can be exploited in real-time systems only after a careful identification of the conditions under which we can safely slow down the processor without missing any deadline (for hard real-time tasks) or missing a limited number of deadlines (for soft real-time tasks). This way, the reduction of the power consumption does not affect the timely execution of important computational activities. In particular, an *energy-aware scheduling algorithm* can exploit DVS by selecting, besides the task to be scheduled, also the processor's operating frequency at each instant of time. The problem becomes more difficult in systems with a combination of hard and soft, periodic and aperiodic real-time tasks.

Real-time systems with variable speed are still described using GANNT charts as shown in Figure 2.2. This figure is similar to Figure 1.3, with an additional vertical y axis which represents the current speed of the processor. Thus, the height of the filled rectangular box representing the assignment of the job to the processor specifies the speed at which the job itself is executed. The speed of the processor is typically represented using the α symbol [98, 43, 25]. For an explanation of the parameters shown in the figure, refer to Section 1.5.

Timeliness and energy efficiency are often seen as conflicting goals. Thus, when designing a real-time system, the first concern is usually time, leaving energy efficiency as a hopeful consequence of empiric decisions. However, some papers presented in the real-time literature [17, 18, 112, 141] have shown that both goals can be achieved at design time.

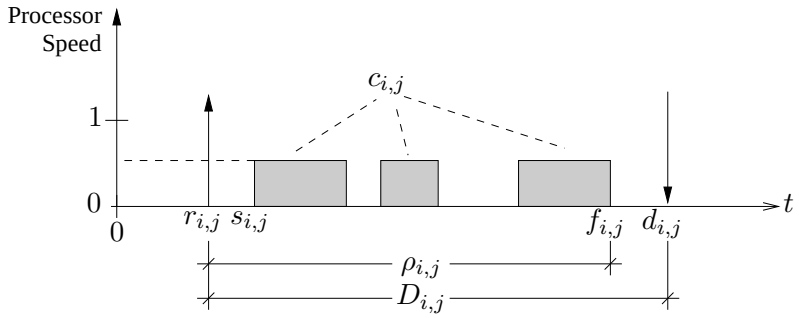

Figure 2.2: A GANNT chart describing the generic job $\tau_{i,k}$ on a DVS processor.

Notice that the power consumption scales linearly with the frequency and quadratically with the voltage — i.e., reducing frequency and voltage together reduces energy per operation quadratically, but only decreases performance linearly. An important consequence is that the minimal energy consumption is obtained by running the task at the lowest uniform speed that allows to meet the deadlines [63, 98]. In fact, the convexity of the power/speed curve implies that maintaining a constant speed $\overline{\alpha}$ is better than switching between two different speeds across $\overline{\alpha}$. We refer to this speed $\overline{\alpha}$ as the *optimal* speed. In practice, however, processors provide only a finite number of discrete speeds [93, 131, 58, 57, 59, 60, 55], and the optimal speed may not be available on a certain processor. When the optimal speed is not available, it has to be approximated with one of the existing values. To prevent any deadline miss, the processor speed is set equal to the closest discrete speed higher than the optimal value. This solution, however, causes a waste of energy, especially when the number of available speeds is small. The increase of energy consumption is called *energy quantization error*, and has been studied by Saewong and Rajkumar [109]. Ishihara and Yasuura [63] have proved that when a processor offers a limited set of speeds, using the two speeds which are immediately neighbors to the optimal speed minimizes the energy consumption. Again, this is a consequence of the convexity of the power/speed curve. Notice that DVS architectures may also have inefficient operating frequencies [109], which exist when the power/speed curve is not convex (i.e., running the task at higher frequencies reduces the energy consumption). The precise definition of inefficient frequencies will be given in Section 3.2.1. A simple online tool for inefficient frequency elimination has been provided in [132]. Removal of inefficient operating frequencies is the first step in any DVS application.

With the advent of variable speed processors, scheduling acquired the new dimension of processor speed. Classic real-time scheduling techniques can now be adapted to address both timing and energy through efficient selection of processor speed. For instance, we have measured [111] that using both a variable speed processor and a good real-time algorithm it is possible to save up to 38 percent of the *total* energy consumed by an embedded system without missing any deadline. The experiment will be described in Section 5.4.2.

2.2.4 Overheads

One issue that must be taken into careful consideration is the overhead of changing frequency. Changing frequency is not "free", as the processor needs some transitory time to adjust to the new frequency. The duration of this transitory is variable, and varies a lot from processor to processor. For example, on the Intel PXA250 [58, 57] it can go up to $500\mu sec$. Even though in many soft real-time applications this can be considered negligible, it should not be ignored. We will show how to account for this delay in Sections 3.2.1 and 5.2.4.

It is also undeniable the presence of an energy overhead at every frequency switch. This overhead depends on the particular kind of processor the algorithm is running on, and it is quite difficult to estimate and measure. In Section 3.2.1, we will introduce a general model for the processor power consumption which accounts for this overhead. In Section 5.2.4, we will also devise a technique to limit the number of switches in an interval of time, therefore limiting the maximum amount of energy spent for switching frequency.

2.3 A Taxonomy of Energy-Aware Scheduling Algorithms

Recently, many energy-aware algorithms have been proposed in the literature to exploit voltage variable processors [112, 98, 18, 101, 141]. In these algorithms, the scheduler, in addition to selecting the executing task, selects also the operating microprocessor frequency. The problem becomes more difficult in systems with a combination of hard and soft, periodic and aperiodic real-time tasks. For this reason, most algorithms focus only on one kind of task (typically, hard periodic real-time tasks), avoiding the more difficult case of the co-existence of different kinds of task in the same system. In practice, however, many real systems consist of a mixture of hard and soft real-time tasks. We will present an energy-aware algorithm capable of handling both kinds of tasks in Chapter 5.

We now present a possible taxonomy of energy-aware scheduling algorithms, which resembles the one proposed by Kim et al. [65]. Remind that this thesis focuses on energy-aware scheduling algorithms for *uniprocessor* systems. The multiprocessor case, in fact, is much more complex and includes several variants: processors may be homogeneous (e.g., SMPs) or heterogeneous (e.g., computer network), the RTOS may have task migration or may not, etc. Moreover, the lack of optimal scheduling algorithms for real-time multiprocessor systems makes the problem of creating energy-aware algorithms with high efficiency even more difficult. Our proposed taxonomy is summarized in Figure 2.3.

Energy-aware algorithms can be divided into *static* and *dynamic* techniques [110, 113], depending on the scaling decisions be taken off-line or on-line, respectively. Notice that static is synonym of off-line, whereas dynamic is synonym of on-line, respectively. In fact, in the former case, the operating system statically chooses the processor speed off-line,

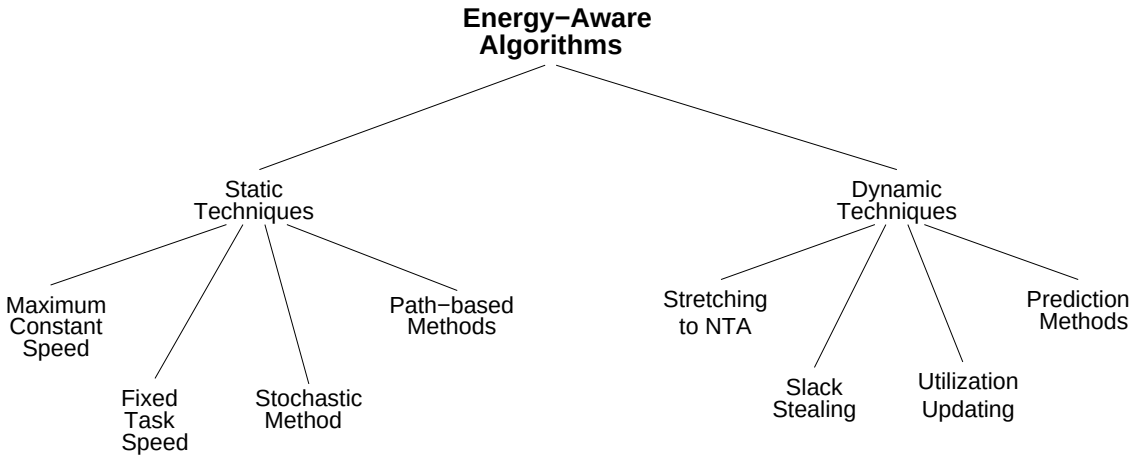

Figure 2.3: Taxonomy of energy-aware scheduling algorithms.

regardless of the run-time behaviour of tasks. In the latter case, instead, the system chooses the processor speed dynamically at run-time. This is the typical notation used in the real-time literature [25, 12, 13, 115, 18, 108].

2.3.1 Static Techniques

Sometimes the system is not fully utilized even if all real-time tasks have a computation time equal to their worst-case requirement. On EDF, for instance, this happens when the total utilization U in less than one. In this case, a *static "slack"* time is present in the system and it can be identified statically in order to reduce the energy consumption by reducing the processor speed.

Static techniques [139, 98, 109, 80, 25, 52] are typically applied to periodic tasks, and make use of off-line parameters, such as periods and worst case execution cycles (WCECs) to select the appropriate processor voltage/frequency to be used. In this techniques, the computation of the processor speed is made off-line. Thus, since the worst-case parameters may differ significantly from the actual values, these techniques save less energy than dynamic schemes. However, the static strategies are very effective when the parameters do not vary significantly.

Static techniques can be further divided into four classes, depending on *when* the processor speed is changed. These classes are: maximum constant speed, fixed task speed, stochastic methods and path-based methods.

Maximum constant speed

In this first class of methods, a single constant speed is computed off-line and assigned to execute all tasks until the task set changes. This speed is defined as the *lowest possible*

clock speed that guarantees the feasible schedule of a task set [121]. This technique is also called *"Fixed System Voltage Assignment"*.

With this approach, there is no additional overhead for voltage switching at run-time. However, since the timing constraints of all tasks must be satisfied, the clock frequency must be the worst case highest clock frequency needed by the task set, which may result in less energy saving compared to other approaches.

Fixed Task Speed

This second class of static methods is sometimes called *"Fixed Task Voltage Assignment"* and belongs to the class of Inter-Task Scaling techniques [65], meaning that the processor speed is not fixed but *statically* assigned on a task-by-task basis (i.e., based upon task's parameters) before system execution. In other words, given a set of periodic tasks, the algorithm assigns a possibly different clock frequency to each individual task. The assignments are still computed off-line, and are fixed until the task set changes. Since the task schedule is periodic, the voltage schedule obtained by this method is also periodic and can be stored in a table. With respect to fixed system voltage assignment, this technique consumes an additional overhead to scale the processor frequency to the proper value during a context switch.

Some of these methods propose to assign a different speed to each task [109]. Some others adopt a more general scheme, where the speed switching instants are more freely chosen and, typically, occur at the activation/deadline of some job [139, 80].

The major drawback of this approach, which actually prevents its use in real-world applications, derives from the tight relationship established between the task schedule and the power management scheme. If, for some reason, some task activation is lost or delayed, the entire speed assignment is affected, resulting in a potential domino effect on the other tasks in the system, and some deadline may be missed. In this sense, such a speed assignment scheme is fragile because it is affected by the misbehaviour of a task. Running always at a fixed speed is a more robust design practice, because it avoids this potential problem.

Stochastic method

This method belongs to the class of Intra-Task Scaling techniques [65] meaning that the processor speed is adjusted within an individual task boundary (i.e., within the execution of a single job).

This method is based on the idea that it is better to defer some work in the hope that the current job will have an execution time less than the worst case requirement. Thus, if the task finishes earlier than its WCET, the high speed may never be needed. The

method starts task execution at a low speed, and accelerates the processor speed during task execution. The clock speed is raised at specific time instances, until the job finishes.

Theoretically, if the probability density function of the task execution time is known in advance, the optimal speed schedule can be computed [49].

Path-based methods

These methods belong to the Intra-Task Scaling techniques as well [65]. In the path-based methods, the processor frequency is set based on a predicted reference execution path, such as the Worst Case Execution Path (WCEP). Consider the two branches of a typical `if-then-else` statement, which may have very different computation times. If the application deviates from the longest execution path, then the operating frequency can be lowered to reduce the energy consumption.

The locations for possible speed transition inside the program can be identified using static program analysis [119] or execution time profiling [69].

2.3.2 Dynamic Techniques

In most applications, the probability of a task requiring an amount of run-time equal to its WCET is very low [110]. Some studies have observed that the actual execution cycles of real-world embedded tasks may vary up to 87 percent with respect to their measured WCETs [134]. Dynamic techniques take advantage of early job completions, and have been the topic of much recent research [98, 18, 141, 109, 101, 112].

When a task executes less than its worst case requirement, a *dynamic "slack"* time is created as difference between the actual schedule and the worst case scenario. Whereas static techniques are only capable of reclaiming static slack time, dynamic algorithms can reclaim dynamic slack time as well, resulting in a greater amount of energy saved — this additional time can be exploited by the dynamic algorithm to change the schedule at run-time and further reduce the processor speed [33].

Dynamic methods can exploit information about the run-time behaviour of tasks, which may be very far from the pessimistic assumptions required at design time. For this reason, they theoretically allow to reduce energy consumption by a longer amount than static schemes.

Dynamic algorithms are typically based on *slack reclaiming*. Most algorithms use at least one of the following slack estimation techniques.

Stretching to NTA

This simple method to estimate the dynamic slack time consists in computing the Next Time Arrival (NTA), which is the release time of the next job. Suppose that at time t_o

the job $\tau_{i,j}$ is scheduled for execution. If at such time, NTA is later than $(t_o + WCET_i)$, then the execution of $\tau_{i,j}$ can be "stretched" up to NTA (i.e., the processor speed can be lowered so that the execution of $\tau_{i,j}$ in the worst case completes exactly at the NTA). This permits to reduce the processor frequency in the interval of time $[t_o, NTA]$, as shown in Figure 2.4.

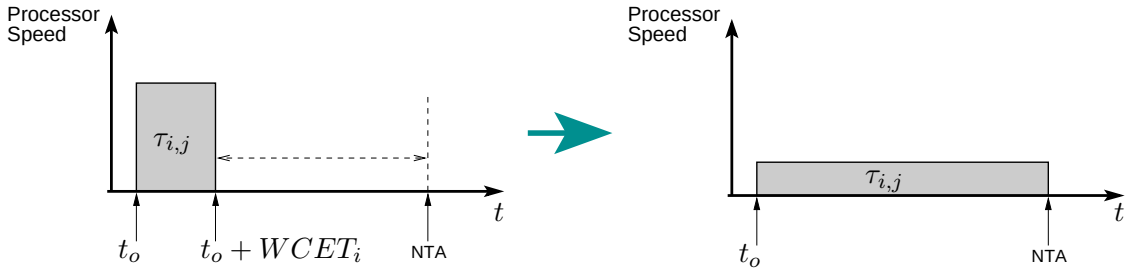

Figure 2.4: The Stretching to NTA technique.

Slack stealing

The basic idea of this method is that when a task completes its execution earlier than its worst case requirement, the following tasks can use the slack time. Typically, tasks only reclaim the slack time of higher priority tasks to avoid computationally expensive algorithms [65].

Utilization updating

This technique estimates the current required utilization at any instant of time [65]. Whenever such utilization changes, the processor frequency is changed accordingly. The main advantage of this method is the simple implementation. In fact, it does not make any assumption on task duration, and waits the task completion to know the exact execution time. Then, it changes the processor speed according to this information.

Prediction methods

If the real-time scheduler knew in advance that a task would complete execution before its worst case execution time, it would exploit this information to further reduce the power consumption. Thus, this class of techniques makes decisions in advance, before current instances of tasks have completed. Typically, these algorithms try to defer some work in the hope that the current instances will execute less than the worst case scenario. Obviously, a right prediction allows to considerably reduce the energy consumption (i.e., more than using techniques that make decisions only at completion times). However, when the predicted behaviour is distant from reality, some undesired side effect (such as

a deadline miss in soft real-time systems or an increase of the energy consumption) may occur.

Many algorithms try to predict the duration of the current task instance based on some task's characteristics such as average execution time. This decision typically depends on the behaviour of the previous instances of the task. The prediction mechanism can be very simple (like keeping the average execution time of each task) or more complex (like exploiting probabilistic information and feedback controllers [68, 67, 141, 39]). The success of this class of methods relies on predicting the task behaviour correctly. For this reason, the use of richer task set information may enhance the effectiveness of the DVS. One important performance metric of these mechanisms is given by the capacity of adapting to ever-changing workloads as fast as possible.

2.4 State of the Art

We now present a description of the current state of the art of energy-aware scheduling algorithms presented in the real-time literature so far. A comparison among some of the algorithms presented here can be found in [120, 65]. The interested reader can refer to the original papers for the full description of the algorithms.

Irani et al. [62] wrote a brief general survey about current research and open problems in power management techniques. Their work includes several research fields, like real-time systems, cooling infrastructures and power reduction to one or more system components. In particular, they formalize power management issues as algorithmic optimization problems.

2.4.1 Power Management Points

Concerning path-based methods, Mossé et al. introduced the concept of *Power Management Points* (PMPs) [92] which are pecies of code that manage information about the execution of program segments (e.g., `if-then-else` branches) to make decisions about how to change the processor speed.

AbouGhazaleh et al. [12, 13] then focused on compiler-inserted PMPs and studied the effect of different overheads on both time and energy when using such mechanism. In fact, there may be cases where the energy consumption exerted by the overhead of selecting and setting a new speed overwhelms any energy savings of a speed setting algorithm. They proposed an analytical model and a theoretical solution for choosing the optimal granularity of PMPs in a program. Their results are validated through a series of experiments with *SimpleScalar* [5], a system software infrastructure used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification.

2.4.2 The PM-Clock Algorithm

In the context of static algorithms, Saewong and Rajkumar [108] provided an algorithm to find the optimal constant speed assignment for fixed priority real-time tasks [109].

They also proposed the voltage-scaling PM-Clock algorithm for hard real-time systems using fixed-priority (i.e., Rate Monotonic or Deadline Monotonic) schedulers. The clock frequency of the processor to execute a periodic task is assigned during the admission control, and it is fixed until the task set changes. Frequencies are assigned to tasks in such a way that the frequency assigned to a task is greater than or equal to that assigned to a lower priority task. The authors proved that the algorithm is optimal in the sense that it consumes the minimum energy among all possible fixed clock frequency assignments.

Finally, they proposed a dynamic clock frequency assignment algorithm, called Dynamic PM-Clock, which minimizes energy consumption when tasks execute less than their worst case execution times. The algorithm detects the early completion of tasks instances and makes use of the additional slack time by reducing the processor speed.

2.4.3 The RTDVS Algorithms

Pillai and Shin [98] proposed three different algorithms for periodic hard real-time tasks. The first algorithm is a static (off-line) algorithm based on the Maximum Constant Speed technique (i.e., the algorithm selects the minimal constant speed which allows to meet all the deadlines for the given task set). They derived the optimal algorithm under EDF and proposed a near-optimal algorithm under Rate Monotonic. In the rest of this thesis, we will refer to this static algorithm using the name RTDVS-Static.

They also proposed two dynamic algorithms, called Cycle-Conserving and Look-Ahead, respectively, to take into account the slack time. The Cycle Conserving algorithm assumes the worst-case at release time and executes at a high frequency until the task completes, and only then it reduces operating frequency and voltage. This algorithm may need to dynamically reduce frequency on each task completion, and to increase frequency on each task release.

The last algorithm (RTDVS-Look Ahead) has a smarter implementation. It defers as much work as possible, setting the processor frequency to the minimum value which ensures that all future deadlines will be met. It may require to run at higher frequencies later to complete all the deferred work in time. However, if tasks tend to use much less than their worst-case execution times, the peak execution rates for deferred work may never be needed. All these algorithms have been proposed for both Rate Monotonic and Earliest Deadline First schedulers.

2.4.4 The DRA Algorithms

Aydin et. al. [17, 18] proposed the DRA algorithm based on EDF for reclaiming the slack time. The algorithm permits to schedule periodic tasks in a hard real-time environment, reducing the energy consumption without missing any deadline. In particular, the DRA scheme consists of a basic algorithm and of two extensions.

The basic algorithm (DRA-Standard) consists on an on-line speed adjustment mechanism which dynamically reclaims slack time not used by tasks that have completed without consuming their worst-case workload. The amount of slack time is computed through a queue of tasks (called α-queue) ordered by the earliest deadline. The queue is used to compute the earliness of tasks when they are dispatched. At any time it contains information about tasks that would be active (i.e., running or ready) at that time in the canonical schedule S^{can}. S^{can} is the static optimal schedule in which every instance presents its worst-case workload and the processor runs at the constant speed $\alpha = max\{\alpha_{min}, U_{tot}\}$, where α_{min} is the lowest available processor speed and U_{tot} is the worst case system utilization according to EDF. At time t, this queue contains information about all instances $\tau_{i,j}$ such that $r_{i,j} \leq t \leq d_{i,j}$ and whose remaining execution time is greater than 0. At dispatch time, the algorithm computes the earliness of tasks and adjusts the processor speed according to this value.

The *"One Task"* extension (DRA-OTE) further slows down the processor speed when there is only one task in the ready queue and its worst-case execution time (under the current speed) does not extend beyond the next event.

The *"Aggressive"* extension (DRA-AGGR), instead, speculatively assumes that the current and future instances of tasks will most probably present a computational demand lower than the worst case requirements. Hence, it tries to reduce the speed of the running task by deferring all the work above a certain threshold, which is set according to the average workload. The algorithm adopts an aggressive approach based on reducing the speed of the running task under certain conditions to a level which is even lower than the one suggested by the basic algorithm and the One Task extension. However, when the worst case scenario happens, the algorithm must increase the processor speed to guarantee the feasibility of future jobs.

Notice that the original papers [17, 18] contain some mistakes in the description of the algorithm. We contacted the authors highlighting such issues, thus they published a technical report to fix the errors [16].

2.4.5 Algorithms by Shin and Kim

Shin and Kim [118] proposed a static and a dynamic algorithm for real-time systems with both periodic and aperiodic tasks. The idea is to handle the aperiodic tasks through a dedicated server, and to let the algorithm select the operating speeds of both the periodic

tasks and the dedicated server.

In particular, the dynamic algorithm can use fixed priority or EDF [79] policy, while the dedicated server can be a *Deferrable Server* [73] or a *Total Bandwidth Server* [124]. Using the stretching to NTA mechanism and some existing energy-aware algorithms (including a modified version of the DRA [17, 18, 16] algorithm) they reclaim the slack time for both periodic and aperiodic tasks.

2.4.6 The DVVST Algorithm

Recently, Qadi et. al. [101] presented the DVSST algorithm that schedules sporadic hard real-time tasks reclaiming the unused bandwidth to lower the processor frequency. This algorithm is based on the Utilization Updating technique. The basic idea is to keep track of the total bandwidth used by all active sporadic tasks with a variable U: when a sporadic task is activated, U is increased by U_i (the task's utilization, $U_i = \frac{C_i}{T_i}$), and at the task's deadline the bandwidth is decreased by U_i. The processor frequency is set depending on the value of U. The algorithm is not able to reclaim the spare bandwidth that is due to tasks with variable execution time. Indeed, in the case of periodic tasks, the algorithm maintains a constant speed, regardless of tasks executing less than their WCET. As we will see in Chapter 5, our algorithm GRUB-PA, instead, explicitly reclaims the spare bandwidth of tasks that execute less than the worst case and, therefore, is able to reclaim spare time in the case of both periodic and sporadic tasks.

2.4.7 Prediction Mechanism by Kumar and Srivastava

Kumar and Srivastava [68, 67] proposed a prediction mechanism for fixed-priority scheduling of soft periodic tasks. The real-time scheduler maintains a *history table* which specifies the probability distribution of the actual execution time of each task. For each task the history table contains n slots, where the i^{th} slot specifies the number of times that the value of the execution time has been comprised in the range $(\frac{i-1}{n} \cdot WCET, \frac{i}{n} \cdot WCET)$ — i.e., the number of jobs whose execution time has been comprised in this interval.

Once a job finishes execution, the prediction mechanism computes the percentage of the WCET used by the job and then increments the corresponding slot of the history table. When dispatching a job, the scheduler predicts the actual execution time of the job as the expected value of the probability distribution. If a job misses its deadline, then the execution time is assumed to be the WCET of the task, and it is stored in the history table.

The algorithm is a soft real-time one, with a trade-off between the amount of energy saved and the number of missed deadlines.

2.4.8 The DVS-EDF Algorithm

Zhu et al. [43, 141] proposed a novel approach combining feedback control with DVS schemes targeting hard real-time systems with dynamic workload. The method integrates a DVS scheduler and a feedback controller within the EDF scheduling algorithm.

The algorithm is called DVS-EDF and divides the execution of each task's instance into two portions (see Figure 2.5). The objective is to provide the average number of cycles C_{avg} in the first portion, exploiting frequency scaling. The second part at the maximum processor speed α_{max} ensures that the deadline is met even when the instance requires a number of cycles equal to the worst case value.

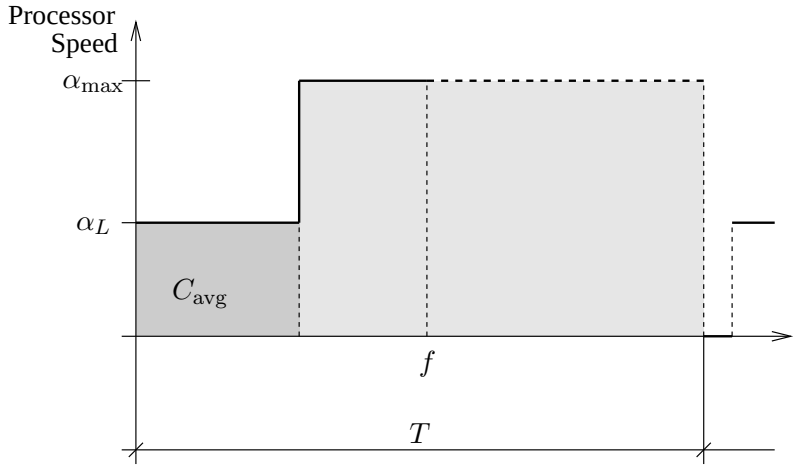

Figure 2.5: The energy management scheme of DVS-EDF.

Notice that this scheme is equivalent to apply the scheme by Ishihara et al. [63] with average time for the first part and rely on the WCET for the total time.

The mechanism is based upon a comparison between the *actual* schedule and the worst case (called *maximal*) schedule (i.e., the schedule produced by EDF when the execution time of every job is equal to the WCET). To decrease the complexity of computing the amount of slack time in the schedule, the algorithm uses the *slack passing* technique, introducing an idle task in the maximal schedule. The WCET and the period of the idle task are chosen in such a way that the total utilization of the task set is equal to 100 percent.

The control is done using a PID continuous feedback controller. A PID controller consists of three different elements — namely, proportional, integral and derivative controls. The proportional control influences the speed of the system of adapting to errors. The integral control is used to adjust the accuracy of the system through the introduction of an integrator on past errors histories. The derivative control is used to increase the stability of the system.

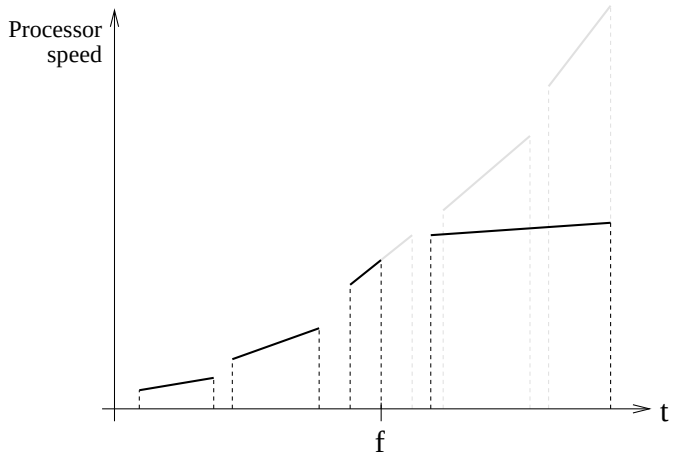

Figure 2.6: The PACE scheme.

2.4.9 PACE Algorithms

Lorch and Smith [81] proposed the Processor Acceleration to Conserve Energy (PACE) model. This is a static stochastic model that increases speed as the task progresses in continuous speed processors (see Figure 2.6). The original model proposed by Lorch and Smith is not very general: it considers only a well-defined power function (i.e., energy per cycle proportional to the speed square) and does not account for the overhead during voltage transition.

Xu et al. [137] reviewed the original model making the extension to the case of discrete speeds and general power functions. Furthermore, they took into account idle power and speed change overhead. The proposed algorithm is called Practical PACE (PPACE).

None of the above papers dealt with optimal speed transition instants, and it is not clear how an optimal sequence of transition points can be found. In the original paper the authors proposed a heuristic to select a "good" sequence of transition points, but they only justified it under the condition of continuous speed [81]. In the second paper, the problem of optimal transition points is said to be "still an open problem beyond the scope of the paper" [137].

An attempt to consider stochastic information in energy reduction problems has been made also by Gruian [49, 50]. The paper addresses energy-aware scheduling of hard real-time tasks on fixed priority (i.e., Rate Monotonic or Deadline Monotonic) schedulers, using stochastic data at both task and task-set levels. However, this study addresses the case in absence of transition overheads and with a specific power function.

2.4.10 Algorithm by Pouwelse et al.

Some work on soft real-time scheduling has been done by Pouwelse et al. [99, 100]. They presented a study of power consumption and energy-aware scheduling applied to mul-

timedia streaming. However, the technique is based on heuristics and cannot provide guarantees to hard real-time tasks.

2.5 Summary

In this chapter, we discussed how the speed of a CMOS processor can be slowed down in order to reduce its energy consumption. We explained the need of an energy-aware real-time scheduling algorithm to guarantee real-time constraints when using the DVS technique. We also proposed a taxonomy of the scheduling algorithms proposed in the literature so far, which is very similar to the one proposed by Kim et al. [65]. Finally, we described the state of the art as well as the most used techniques to reduce energy consumption in the algorithms proposed so far.

In the next chapter, we will explain how to exploit probabilistic information about task execution time to reduce energy consumption. We will find the optimal solution for the basic case of two processor speeds, extending the sub-optimal algorithm proposed by Zhu et al. [43, 141].

Chapter 3

Optimal Speed Assignment for Probabilistic Execution Times

As we have seen in Section 2.4.8, Zhu et al. [43, 141] proposed a two-speed scheme in the DVS-EDF algorithm. Their scheme is equivalent to apply the scheme by Ishihara et al. [63] with average time for the first part and rely on the WCET for the total time. The scheme is sub-optimal, and consists on providing the average number of cycles in the first portion, running the second portion at the maximum processor speed to guarantee hard real-time constraints.

In this chapter, we extend their approach, providing a novel result in the real-time literature. We study the problem of energy minimization from an analytical point of view, integrating the concept of probabilistic execution time within the framework of energy minimization. In particular, we find the optimal values of the instant of frequency transition (i.e., *transition point*) and speed assignments when probabilistic information about task execution times is known. The optimal values are found using a very general model for the processor that accounts for idle power and for both the time and the energy overheads due to voltage/frequency transition. We also show how these results can be applied to some significant cases.

The results shown in this chapter have been already presented at a conference about power-aware real-time computing [110] and have been accepted for publication on a special issue of the *International Journal of Embedded Systems* (IJES) journal [26].

3.1 Towards a probabilistic model for energy minimization

A major weakness in the approach used when designing most of the energy-aware scheduling algorithms is due to the set of assumptions, often not realistic, which are made to simplify the solution. Besides ignoring the energy consumed by the processor when it is idle, these methods often neglect also the delay due to a frequency transition, preventing

thus the application of research results to real-world systems. In some approaches [69, 88] such a delay has been considered in the processor model, but only dynamic techniques aimed at reducing the slack time have been developed. For this reason, we have formulated a general model for the processor [25, 110, 26], which takes into account both time and energy overheads due to voltage transition.

In real-time systems, any energy-aware policy acting on the processor speed must take timing constraints into account, to guarantee the timely execution of all computational activities. As we have seen in the introduction, hard real-time systems are typically designed to handle peak loads for safety reasons. However, peak load conditions rarely happen in practice, and the system resources are underutilized most of the time. For example, server loads often vary significantly depending on the time of the day or other external factors, and the average processor use is between 10 and 50 percent of the peak capacity [27, 107]. These issues are even more critical in embedded systems [138], where the actual execution times of tasks can vary up to 87 percent with respect to their measured worst case execution times [134]. This suggests that a striking energy reduction can be achieved by enriching DVS policies with a more detailed information on the required workload — i.e., the use of a richer task information may enhance the effectiveness of DVS.

Recently, the discipline of probabilistic timing analysis has significantly advanced [30, 44], and today there exist tools that can provide the probability density function (p.d.f.) of task's execution times [23]. Basically, these tools partition the task code into basic blocks, which are sequential instructions between two consecutive conditional branches. The duration of each block depends on the processor status (e.g., caches, pipeline stages, out-of-order execution, etc.) and it can be modeled by a random variable. The p.d.f. of the whole task can be extracted by combining the information of every block. This information can then be exploited to reduce energy consumption.

In this chapter, we show how to integrate the concept of probabilistic execution time within the framework of energy minimization. We show how probabilistic information about task's execution times can be exploited to reduce the energy consumed by the processor without missing timing constraints. Optimal speed assignments and transition points are found using a very general model that accounts for processor idle power and for both time and energy overheads due to frequency transitions. We also show how these results can be applied to some significant examples.

3.2 Energy management scheme

We focus on the problem of reducing the energy consumed by a hard real-time task τ_i on a variable speed processor, following a methodology similar to the one by Zhu et al. [43, 141].

Other existing energy-aware algorithms [81, 137] have been deployed starting from this simple scheme, since it constitutes a good starting point for more complex analysis.

In our model, task τ_i has period and deadline both equal to T. The number of processor cycles required by the task is modelled by a random variable whose p.d.f. is $f_C(c)$. The maximum possible number of cycles needed by τ_i is $C_{\max}$. Since the task is hard real-time, $C_{\max}$ cycles must be available in $[0, T]$ whenever the task needs them.

If the number of required cycles in $[0, T]$ was known in advance, it has been shown [63, 99] that using a constant speed during task execution would minimize the energy consumed assuming a continuous speed processor. In fact, the convexity of the power/speed curve implies that maintaining a constant speed $\overline{\alpha}$ is better than switching between two different speeds across $\overline{\alpha}$. When the processor offers a limited set of speeds, using the two speeds which are immediately neighbors to the optimal speed minimizes the energy consumption [63].

In our scheme, it is not possible to compute the optimal constant speed $\overline{\alpha}$ because the actual number of cycles required by the current instance of τ_i is *not* known in advance. In this case, a common technique adopted in the literature [18, 141, 98, 81, 137] is based upon the idea that the current instance of τ_i will request much less than its WCEC $C_{\max}$.

We apply this technique by splitting task execution into two parts, as shown in Figure 3.1. In the first part, the processor runs at a lower speed α_L to reduce the energy consumed in the average case. In the second part, instead, the processor runs at a higher speed α_H in order to provide up to $C_{\max}$ cycles even in the worst case. The idea is that, if a task tends to use much less than its WCEC, the second part, which consumes more energy, may never be needed. When the worst-case condition occurs, instead, the speed increase guarantees the completion of all the deferred work within $[0, T]$.

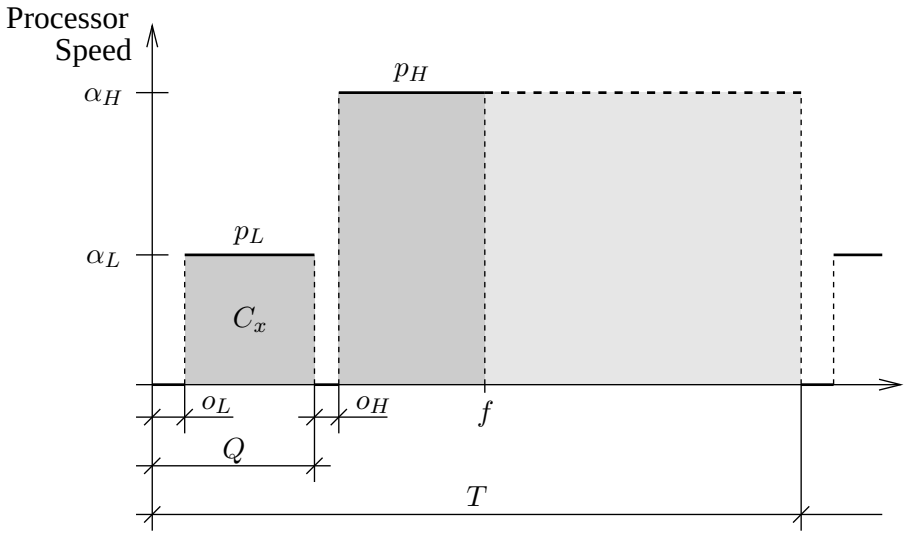

Figure 3.1: The energy management scheme used throughout Chapter 3.

The idea of deferring work has been widely used in the literature to create several energy-aware algorithms [98, 18, 141]. However, most of these techniques follow intuitive ideas (such as providing the average execution cycles in the first part [141]) to simplify the solution, and do not study the problem of optimal values for speed assignments and transition instants from an analytical point of view.

In the rest of this chapter, instead, we will show how it is possible to analytically find the best processor speeds for the first and the second parts of the scheme presented in Figure 3.1 as well as for the transition instant. We will find such optimal values using a very general model for the processor, which accounts for idle power and for both the time and the energy overheads due to voltage/frequency transition. All the simulations in this chapter have been computed using the *Matlab* environment [129].

3.2.1 Processor model

As we have explained in Section 2.2, in CMOS circuits, the energy consumption due to dynamic switching dominates the energy lost by leakage currents, and the dynamic portion of energy consumption is modelled by well known polynomial relationships [36, 53]. However, as the integration technology advances, it is expected that the leakage will significantly affect, if not dominate, the overall energy consumption in integrated circuits (ICs) [61, 103, 50, 28]. Very recently, some work addressed the issue of scheduling a real-time application while reducing the leakage power as well [102]. Moreover, an important fraction of the consumed energy depends on the memory. It has been shown [99] that, at low frequencies, the energy consumption is dominated by the memory, whereas at high frequencies it is dominated by the processor core.

All these remarks have led us to formulate a general model for the processor energy consumption [25, 110, 26]. The processor is characterized by a set $\mathcal{M} = \{\Lambda_1, \Lambda_2, \Lambda_3, \ldots\}$ of operating modes.

Each mode $\Lambda_k = (\alpha_k, p_k, o_k, e_k)$ is described by four parameters:

- α_k is the processor speed in mode Λ_k, measured as number of cycles per second;
- p_k is the power consumed in mode Λ_k when running at speed α_k, measured in Watts;
- o_k is the time overhead needed to enter mode Λ_k, measured in seconds;
- e_k is the energy overhead spent in the switch to the mode Λ_k, measured in Joule.

For the sake of simplicity, we assume that o_k and e_k only depend on mode Λ_k and neither on the mode the processor was operating before (i.e., the previous processor mode), nor on the processor status. We make this assumption in our model to keep the problem tractable analytically. In fact, we are interested in the optimal solution in a closed form (if any). Notice that AbouGhazaleh et al. [12, 13] formulated a model where the overheads

depend also on the previous processor mode. Their model, however, is used to solve a different class of energy minimization problem. Also, we consider only efficient speeds meaning that

$$\forall \Lambda_i, \Lambda_j \qquad \alpha_i \leq \alpha_j \;\Rightarrow\; \frac{p_i}{\alpha_i} \leq \frac{p_j}{\alpha_j}. \tag{3.1}$$

In fact, if this condition is not true for some Λ_i and Λ_j, then the mode Λ_j would be always more efficient than the mode Λ_i. In this case the mode Λ_i is said to be an *inefficient* mode. Notice that the "efficiency" of the processor mode is not affected by the value of the energy and time overheads, but only by the convexity of the power/speed curve (i.e., we only consider the efficiency of *running* at that mode). Inefficient operating frequencies, thus, can be easily removed from the set of available frequencies [109, 132].

Notice that this model is very general, since it is suitable for both continuous and discrete speed processors. If the processor can vary its speed continuously, then the set $\mathcal{M}$ is composed by infinite modes; on the other hand, if the processor has only discrete operating modes then the set $\mathcal{M}$ will be finite.

Finally, we suppose that the processor has one *idle operating mode* denoted by Λ_I. The processor enters idle mode Λ_I when all the computation required by the task is completed. When running in idle mode, the processor does not provide any useful computation (i.e., $\alpha_I = 0$). Notice that existing processors have several idle modes presenting different features. Taking into account only one idle mode Λ_I, however, constitutes a good starting point for considering more complex processors.

3.3 Optimal speed assignment

For our purposes, a speed assignment is optimal when it minimizes the average energy consumed. Given the p.d.f. of the task computation time $f_C(c)$, the expected energy consumption E^{avg} can be computed as $\int_{-\infty}^{+\infty} E(c) f_C(c)\,\mathrm{d}c$, where $E(c)$ denotes the energy consumed when the task executes for c cycles. The optimal values of the parameters occur where the partial derivatives of E^{avg} are equal to zero.

Table 3.1 contains the glossary and the notations used throughout this chapter.

3.3.1 Average Energy Consumption

In this section, we compute the average energy consumption, based on the probabilistic information of the task execution time. The optimal values for speed assignments and transition instants will be computed in Section 3.3.2 starting from this result.

The energy consumed by the processor can be split into two separate contributions: the *active energy* E_A, consumed when executing the task τ_i, and the *idle energy* E_I, consumed when the task has terminated and the processor has entered mode Λ_I. Since

Symbol	Explanation
Λ_k	the k^{th} processor operating mode
α_k	speed when running in Λ_k mode
p_k	power consumption when running in Λ_k mode
o_k	the time overhead to enter the Λ_k mode
e_k	the energy overhead to enter the Λ_k mode
Λ_L	the low speed operating mode
Λ_H	the high speed op. mode, entered after the speed switch
Λ_I	the idle op. mode, entered when the task has finished
T	the task period and deadline
$C_{\max}$	the worst-case execution cycles
C_{avg}	the average execution cycles
C_x	amount of cycles provided before the mode switch from Λ_L to Λ_H
Q	instant when switching from Λ_L to Λ_H
$f_C(c)$	the probability density function (p.d.f.) of the execution cycles
$F_C(c)$	the cumulative distribution function (c.d.f.) of the exec. cycles
$G_C(c)$	$= \int_0^c f_C(x)\mathrm{d}x$. A property of $G_C(c)$ is that $G_C(C_{\max}) = C_{\text{avg}}$
$\gamma(x)$	$= G_C(x) + x(1 - F_C(x))$, auxiliary function used to compact the expressions
$E_A(c)$	active energy (consumed in modes Λ_L, Λ_H) when executing c cycles
$E_I(c)$	idle energy (consumed in mode Λ_I) when executing c cycles
E_A^{avg}	average active energy
E_I^{avg}	average idle energy
E^{avg}	average total (active+idle) energy

Table 3.1: Glossary and notations used throughout Chapter 3.

these two terms must be added, they can be considered separately. First, let us compute the value of the active energy.

Let α_L and α_H be the lower and the higher processor speeds, respectively. The period of the scheme is T. The number of processor cycles required by the task τ_i in each period is modelled by a random variable whose p.d.f. is $f_C(c)$, and the maximum number of cycles is $C_{\max}$. This amount of cycles must be guaranteed in each period because the task is subject to a hard real-time constraint. Our goal is to find the optimal values for the two speed levels α_L and α_H and the instant of time Q when to switch.

Let C_x be the number of cycles provided while running at α_L, as shown in Figure 3.1. Let also be c the actual number of cycles required by the current instance of τ_i, and f the finishing time of the task. We distinguish two different cases:

1. if $c \leq C_x$, then the task terminates before the speed switch, and we expect $f \leq Q$;
2. otherwise, if $c > C_x$, then we need to run at speed α_H to provide the required cycles and we expect $f > Q + o_H$.

We consider the two cases separately.

In the first case ($c \leq C_x$), the finishing time is

$$f = o_L + \frac{c}{\alpha_L} \tag{3.2}$$

and the active energy consumed in one period T is

$$E_A = e_L + p_L\,(f - o_L) = e_L + \frac{p_L}{\alpha_L}\,c. \tag{3.3}$$

On the other hand, when $C_x < c \leq C_{\max}$, we have

$$f = Q + o_H + \frac{c - C_x}{\alpha_H} \tag{3.4}$$

and the energy consumption is

$$E_A = e_L + \frac{p_L}{\alpha_L}\,C_x + e_H + \frac{p_H}{\alpha_H}\,(c - C_x). \tag{3.5}$$

The overall behaviour of the energy consumption as function of the number of cycles c is reported in Figure 3.2. Notice that, due to the assumption of Equation 3.1, the slope $\frac{p_H}{\alpha_H}$ is greater than $\frac{p_L}{\alpha_L}$.

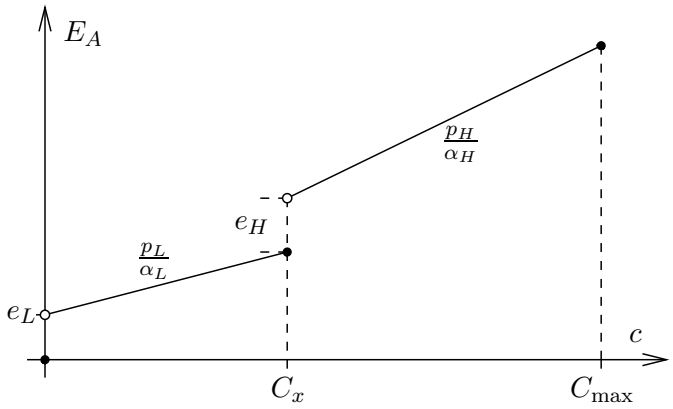

Figure 3.2: The active energy E_A vs. the computation time c

Equations 3.3 and 3.5 provide the active energy E_A consumed when the number of cycles is c. Since the number of cycles is a random variable with p.d.f. $f_C(c)$, then the energy consumed is a random variable as well. Our goal is to minimize the expectation E_A^{avg} of the random variable E_A, whose value is

$$\begin{aligned}
E_A^{\text{avg}} &= \int_0^{C_x} E_A\, f_C(c)\,\mathrm{d}c + \int_{C_x}^{C_{\max}} E_A\, f_C(c)\,\mathrm{d}c \\
&= \int_0^{C_x} \left(e_L + \frac{p_L}{\alpha_L}\,c\right) f_C(c)\,\mathrm{d}c + \int_{C_x}^{C_{\max}} \left(e_L + \frac{p_L}{\alpha_L}C_x + e_H + \frac{p_H}{\alpha_H}\,(c - C_x)\right) f_C(c)\,\mathrm{d}c \\
&= e_L + \frac{p_L}{\alpha_L}G_C(C_x) + \left(e_H - \left(\frac{p_H}{\alpha_H} - \frac{p_L}{\alpha_L}\right)C_x\right)(1 - F_C(C_x)) + \frac{p_H}{\alpha_H}\,(C_{\text{avg}} - G_C(C_x)) \\
&= e_L + e_H(1 - F_C(C_x)) + \frac{p_H}{\alpha_H}\,C_{\text{avg}} - \left(\frac{p_H}{\alpha_H} - \frac{p_L}{\alpha_L}\right)(G_C(C_x) + C_x(1 - F_C(C_x)))
\end{aligned}$$

where we set

$$F_C(x) = \int_0^x f_C(c)\,\mathrm{d}c \qquad G_C(x) = \int_0^x c\, f_C(c)\,\mathrm{d}c.$$

For compactness, if we also set

$$\gamma(x) = G_C(x) + x(1 - F_C(x)), \tag{3.6}$$

the average active energy E_A^{avg} consumed in a period can be written as

$$E_A^{\text{avg}} = e_L + e_H(1 - F_C(C_x)) + \frac{p_L}{\alpha_L}\gamma(C_x) + \frac{p_H}{\alpha_H}(C_{\text{avg}} - \gamma(C_x)) \tag{3.7}$$

Notice that $G_C(C_{\max})$ is equal to C_{avg}, by definition. For this reason, we always have $0 \le \gamma(x) \le C_{\text{avg}}$ for all x.

Accounting for the idle power Equation 3.7 takes into account only the energy consumed when the processor is running the task. We now evaluate the contribution E_I to the energy consumed by the processor after the task has terminated. This contribution is

$$E_I = e_I + p_I(T - f - o_I)$$

where e_I and o_I are the energy and time overheads to enter the idle mode, respectively, and f is the finishing time of the task.

From the previous Equations 3.2 and 3.4, which established the relationship between the required number of cycles and the finishing time f, we can express E_I as function of the number of cycles c. Hence, we have

$$E_I = \begin{cases} e_I + p_I(T - o_L - \frac{c}{\alpha_L} - o_I) & \text{if } c \le C_x \\ e_I + p_I(T - o_L - \frac{C_x}{\alpha_L} - o_H - \frac{c - C_x}{\alpha_H} - o_I) & \text{if } c > C_x \end{cases} \tag{3.8}$$

Similarly as done for the active power consumption, we calculate the average consumption of idle energy by integrating Equation 3.8 with the p.d.f. $f_C(c)$ of the number

of cycles.

$$\begin{aligned}
E_I^{\text{avg}} =& e_I + p_I \left(T - o_L - o_I - \int_0^{C_x} \frac{c}{\alpha_L} f_C(c)\,\mathrm{d}c - \int_{C_x}^{C_{\max}} \left(\frac{C_x}{\alpha_L} + o_H + \frac{c - C_x}{\alpha_H} \right) f_C(c)\,\mathrm{d}c \right) \\
=& e_I + p_I \left(T - o_L - o_I - \int_0^{C_{\max}} \left(\frac{C_x}{\alpha_L} + o_H + \frac{c - C_x}{\alpha_H} \right) f_C(c)\,\mathrm{d}c \right. \\
& \left. + \int_0^{C_x} \left(\frac{C_x}{\alpha_L} + o_H + \frac{c - C_x}{\alpha_H} - \frac{c}{\alpha_L} \right) f_C(c)\,\mathrm{d}c \right) \\
=& e_I + p_I \left(T - o_L - o_I - \frac{C_x}{\alpha_L} - o_H - \frac{C_{\text{avg}} - C_x}{\alpha_H} + \left(\frac{C_x}{\alpha_L} - \frac{C_x}{\alpha_H} + o_H \right) F_C(C_x) \right. \\
& \left. + \left(\frac{1}{\alpha_H} - \frac{1}{\alpha_L} \right) G_C(C_x) \right) \\
=& e_I + p_I \left(T - o_L - o_I - o_H(1 - F_C(C_x)) - \frac{C_{\text{avg}}}{\alpha_H} - \left(\frac{1}{\alpha_L} - \frac{1}{\alpha_H} \right) \gamma(C_x) \right)
\end{aligned}$$

where we used the definition of $\gamma(x)$ given in Equation 3.6. We can write E_I^{avg} in a more compact form as

$$E_I^{\text{avg}} = e_I + p_I \left(T - o_L - o_I - o_H(1 - F_C(C_x)) - \frac{C_{\text{avg}} - \gamma(C_x)}{\alpha_H} - \frac{\gamma(C_x)}{\alpha_L} \right) \tag{3.9}$$

Finally, if we add Equation 3.7 and Equation 3.9 we find that

$$\begin{aligned}
E^{\text{avg}} = e_L + e_I + p_I(T - o_L - o_I) + (e_H - p_I o_H)(1 - F_C(C_x)) \\
+ \frac{p_L - p_I}{\alpha_L} \gamma(C_x) + \frac{p_H - p_I}{\alpha_H} (C_{\text{avg}} - \gamma(C_x))
\end{aligned} \tag{3.10}$$

Equation 3.10, which extends Equation 3.7 to the case of idle power, is a new result in the literature and an original contribution of this thesis. It expresses the average energy consumption as function of the probability density of the task execution cycles, taking also into account the idle power and the overheads.

Now an essential remark is in order. Equation 3.10 can be obtained from Equation 3.7 by the following substitution

$$\begin{aligned}
\tilde{e}_L &= e_L + e_I + p_I(T - o_L - o_I) & \tilde{e}_H &= e_H - p_I o_H \\
\tilde{p}_L &= p_L - p_I & \tilde{p}_H &= p_H - p_I
\end{aligned}$$

Hence, we can say that, in our model, *the idle power can be taken into account by a simple adjustment of the operating modes.* For this reason, unless specified differently, in the rest of the paper we will consider only Equation 3.7.

3.3.2 Finding the minimum energy consumption

Equation 3.7 is valid for processors with discrete as well as continuous operating modes. Taking into account discrete operating modes requires the evaluation of the energy consumed by all the mode pairs. The problem then becomes to find the optimal pair with the lowest total energy consumption. The complexity of this evaluation process is $O(m^2)$, where m is the number of available efficient operating modes.

For processors with continuous operating modes, instead, it is possible to *analytically* find the optimal values for speed assignments and transition instant. We know that no existing processor can vary its frequency with continuity. In fact, all processors that support DVS provide a set of operating modes, each one characterized by a value of frequency and voltage [131, 57, 59, 60, 55]. However, many significant contributions in the literature [18, 98, 141, 112] assume a continuous speed because if the processor speed levels are very close each other, then this approximation is very close to reality. Obviously, if the optimal speed is not available, it has to be approximated with the closest discrete speed higher than the optimal one. In this case, there is an increase of energy consumption, called *energy quantization error*, that has been studied by Saewong and Rajkumar [109].

In the continuous model, we assume that

- all operating modes Λ_k require the same time overhead o and energy overhead e. Formally, we have that $\forall k \; e_k = e, \; o_k = o$;

- the speed α varies within $[0, \alpha_{\max}]$, where $\alpha_{\max}$ is the maximum speed allowed by the processor;

- the power consumption at speed α is modelled by the function $p(\alpha)$. Typically, the power function $p(\alpha)$ is a polynomial [36]. However, as stated earlier, it is expected that the power/speed relationship may differ from the ideal polynomial function [50]. For this reason we model this relationship by a generic function $p(\alpha)$.

Differently from the case of discrete operating modes, in the case of continuous operating modes we can find the conditions of minimum energy consumption (i.e., optimal transition instant and speed levels) starting from Equation 3.7.

The speeds (α_L, α_H) can be expressed as function of C_x and Q, as follows

$$\alpha_L = \frac{C_x}{Q - o} \qquad \alpha_H = \frac{C_{\max} - C_x}{T - Q - o}. \tag{3.11}$$

These equalities follow directly from the adopted energy management scheme as shown in Figure 3.1. In the remainder of this chapter, we will develop the energy management scheme using C_x and Q as our free variables.

It is very insightful to plot the quantity E^{avg} on a plane (C_x, Q). Figure 3.3 shows the 3-D surface of E^{avg}, when we assume that the execution cycles are uniformly distributed in $[2, 10]$, the period T is 2 and the power function is $p(\alpha) = k\,\alpha^3$.

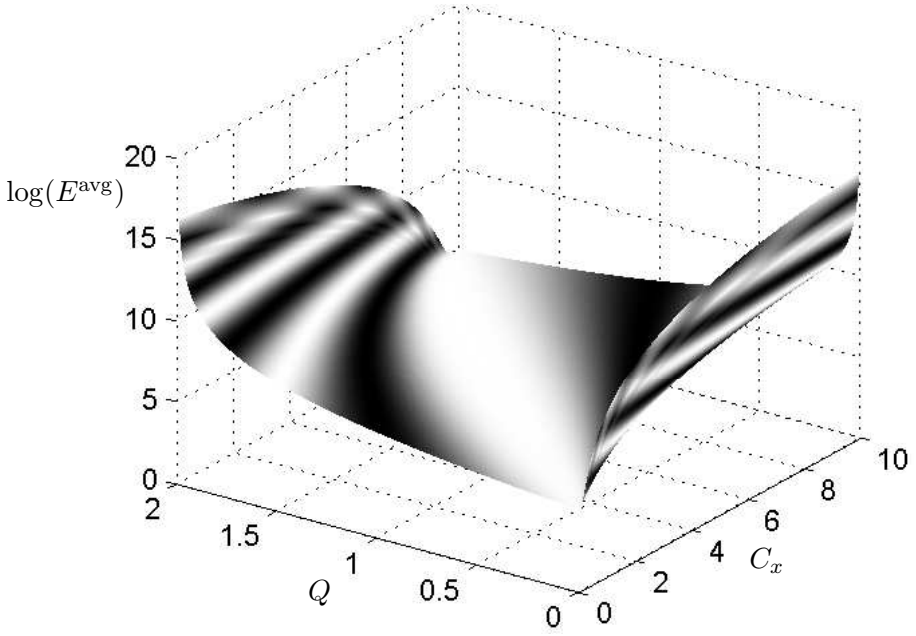

Figure 3.3: E^{avg} for uniform execution times.

Another example is presented in Figure 3.4, which shows the level curves of the quantity E^{avg} as function of C_x and of Q. In the plot, we assumed an exponential p.d.f. with average value $C_{\text{avg}} = 0.2929$, a period T equal to 1 and a power function $p(\alpha) = k\,\alpha^3$.

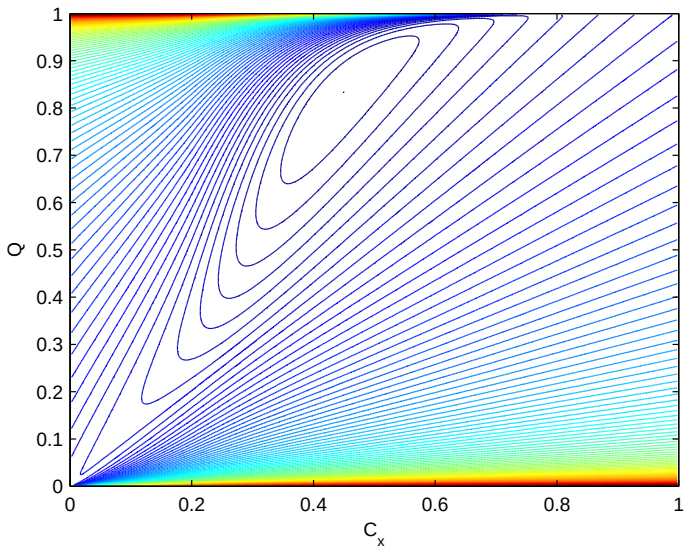

Figure 3.4: Level curves of E^{avg} for exponential p.d.f.

The minimum occurs at the center of the white region for a value of C_x greater than C_{avg}.

The minimum of Equation 3.7 can be found analytically by calculating the partial derivative of E^{avg} with respect to the variables (C_x, Q). Let $p'_H = \frac{\mathrm{d}p}{\mathrm{d}\alpha}(\alpha_H)$ and $p'_L = \frac{\mathrm{d}p}{\mathrm{d}\alpha}(\alpha_L)$. We have:

$$\begin{aligned}\frac{\partial E^{\text{avg}}}{\partial C_x} = &- e\, f_C(C_x) - \left(p'_H - \frac{p_H}{\alpha_H}\right)\frac{C_{\text{avg}} - \gamma(C_x)}{C_{\max} - C_x} - \frac{p_H}{\alpha_H}\gamma'(C_x) \\ &+ \left(p'_L - \frac{p_L}{\alpha_L}\right)\frac{\gamma(C_x)}{C_x} + \frac{p_L}{\alpha_L}\gamma'(C_x)\end{aligned} \tag{3.12}$$

where we used the property that, from Equation 3.11, it follows:

$$\begin{aligned}{\alpha_L}' = \frac{\partial \alpha_L}{\partial C_x} = \frac{1}{Q-o} = \frac{\alpha_L}{C_x} \quad &\Longrightarrow \frac{{\alpha_L}'}{\alpha_L} = \frac{1}{C_x} \\ {\alpha_H}' = \frac{\partial \alpha_H}{\partial C_x} = -\frac{1}{T-Q-o} = -\frac{\alpha_H}{C_{\max} - C_x} \quad &\Longrightarrow \frac{{\alpha_H}'}{\alpha_H} = -\frac{1}{C_{\max} - C_x}\end{aligned}$$

An equivalent property can also be found for the differentiation with respect to Q. In fact, we have

$$\begin{aligned}\frac{\partial \alpha_L}{\partial Q} = -\frac{C_x}{Q^2} = -\frac{{\alpha_L}^2}{C_x} \quad &\Longrightarrow \frac{{\alpha_L}'}{{\alpha_L}^2} = -\frac{1}{C_x} \\ \frac{\partial \alpha_H}{\partial Q} = \frac{C_{\max} - C_x}{(T-Q-o)^2} = \frac{{\alpha_H}^2}{C_{\max} - C_x} \quad &\Longrightarrow \frac{{\alpha_H}'}{{\alpha_H}^2} = \frac{1}{C_{\max} - C_x}\end{aligned} \tag{3.13}$$

Now we complete the analysis of the function E^{avg} by computing $\frac{\partial E^{\text{avg}}}{\partial Q}$, which can be greatly simplified thanks to Equation 3.13. Notice that, differently from Equation 3.12, in the next equation ${\alpha_L}'$ and ${\alpha_H}'$ denote $\frac{\partial \alpha_L}{\partial Q}$ and $\frac{\partial \alpha_H}{\partial Q}$, respectively.

$$\frac{\partial E^{\text{avg}}}{\partial Q} = (p'_H \alpha_H - p_H)\frac{C_{\text{avg}} - \gamma(C_x)}{C_{\max} - C_x} - (p'_L \alpha_L - p_L)\frac{\gamma(C_x)}{C_x} \tag{3.14}$$

Equations 3.12 and 3.14 are the components of the gradient ∇E^{avg}. From functional analysis, we know that the minimum satisfies the condition $\nabla E^{\text{avg}} = 0$. Once the optimal (C_x, Q) is found, then the constraint $\alpha_H \leq \alpha_{\max}$ must be checked. In fact, if it is violated, it means that the global minimum would result in a too high value of α_H. In this case, from the Kuhn-Tucker conditions, we know that the minimum occurs when $\alpha_H = \alpha_{\max}$, which means that

$$\frac{C_{\max} - C_x}{T-Q-o} = \alpha_{\max} \;\Rightarrow\; \alpha_L = \frac{C_x\, \alpha_{\max}}{\alpha_{\max}(T-2o) - C_{\max} + C_x} \tag{3.15}$$

From Equation 3.7, substituting α_H with $\alpha_{\max}$ and α_L with the expression of Equation 3.15, we find E^{avg} as function of the unique variable C_x. The minimal energy solution is found by applying classical techniques of functional analysis of one-variable functions.

3.4 Examples

After the main equations for the general case are found, we show how they can be applied to find the optimal (C_x, Q) in some significant examples.

When considering continuous speed levels, a common assumption is that the relationship between the power consumption p and speed α is

$$p(\alpha) = k\,\alpha^n$$

for some k, n. The typical value of n is 3. However, we keep the general form as long as the math is tractable. In these hypothesis, the gradient can be simplified as follows:

$$\begin{cases} \dfrac{\partial E^{\text{avg}}}{\partial C_x} = -\,e\,f_C(C_x) - k\left(\left((n-1)\dfrac{C_{\text{avg}}-\gamma(C_x)}{C_{\max}-C_x}+\gamma'(C_x)\right){\alpha_H}^{n-1}\right. \\ \qquad\qquad \left. -(n-1)\left(\dfrac{\gamma(C_x)}{C_x}+\gamma'(C_x)\right){\alpha_L}^{n-1}\right) \\ \dfrac{\partial E^{\text{avg}}}{\partial Q} = k\,(n-1)\left({\alpha_H}^n\dfrac{C_{\text{avg}}-\gamma(C_x)}{C_{\max}-C_x} - {\alpha_L}^n\dfrac{\gamma(C_x)}{C_x}\right) \end{cases}$$

In order to find the conditions of minimum energy, we have to set both the gradient components equal to zero. The math is greatly simplified by assuming no overhead ($e = 0$ and $o = 0$). If we do so, by setting $\nabla E^{\text{avg}} = 0$, we finally find that the pair (C_x, Q) minimizing the average energy E^{avg} must satisfy Equations 3.16.

$$\begin{cases} \dfrac{(n-1)\,\gamma(C_x) + C_x\gamma'(C_x)}{(n-1)\,(C_{\text{avg}}-\gamma(C_x)) + (C_{\max}-C_x)\gamma'(C_x)}\left(\dfrac{C_{\max}}{C_x}-1\right)\left(\dfrac{C_{\text{avg}}}{\gamma(C_x)}-1\right) = \dfrac{T}{Q}-1 \\ \left(\dfrac{C_{\max}}{C_x}-1\right)^{n-1}\left(\dfrac{C_{\text{avg}}}{\gamma(C_x)}-1\right) = \left(\dfrac{T}{Q}-1\right)^n \end{cases} \tag{3.16}$$

For their importance, in the rest of the chapter we will refer to the Equations 3.16 with the name of *minimum stochastic energy equations*. Once we know n and the probability density $f_C(c)$, Equations 3.16 can be solved to obtain the pair (C_x, Q) that minimizes the energy consumption.

3.4.1 Uniform Density

Let us now assume a uniform density between $C_{\min}$ and $C_{\max}$. It means that

$$f_C(c) = \begin{cases} \frac{1}{C_{\max} - C_{\min}} & \text{if } C_{\min} \leq c \leq C_{\max} \\ 0 & \text{otherwise} \end{cases}$$

and also, when $C_{\min} \leq c \leq C_{\max}$,

$$F_C(c) = \frac{c - C_{\min}}{C_{\max} - C_{\min}} \qquad G_C(c) = \frac{c^2 - {C_{\min}}^2}{2(C_{\max} - C_{\min})}.$$

The function $\gamma(c)$, defined in Equation 3.6, and its derivative are, respectively,

$$\gamma(c) = \frac{-c^2 + 2c\,C_{\max} - {C_{\min}}^2}{2(C_{\max} - C_{\min})} \qquad \gamma'(c) = \frac{C_{\max} - c}{C_{\max} - C_{\min}}$$

In this case, the minimum energy can be simply found by properly substituting $\gamma(C_x)$ and $\gamma'(C_x)$ in the minimum stochastic energy equations (3.16). To simplify and compact them, it is very convenient to normalize the cycles C_x and $C_{\min}$ with respect to $C_{\max}$. Hence, we set $x = \frac{C_x}{C_{\max}}$ and $a = \frac{C_{\min}}{C_{\max}}$.

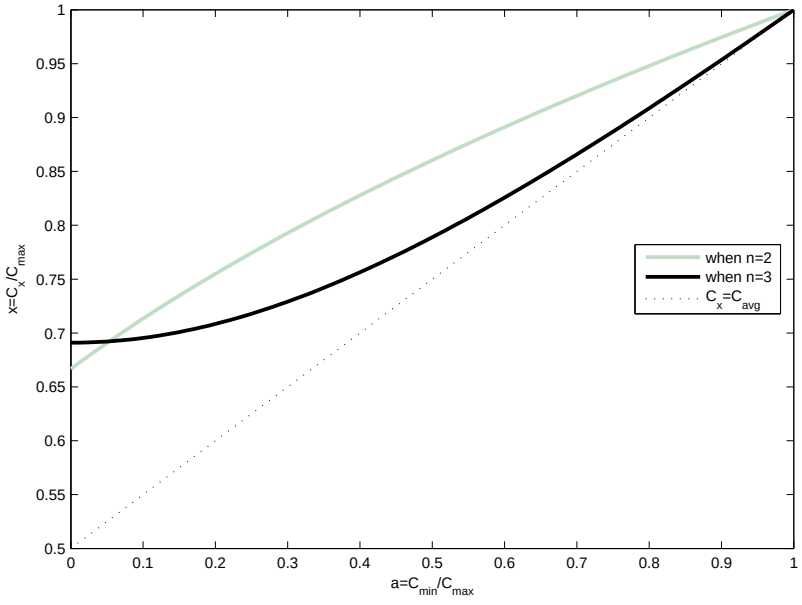

Figure 3.5: The optimal number of cycles in case of uniform density and polynomial power function.

When $n = 2$, after a long algebraic manipulation, we find that the minimum point is

the solution of the polynomial

$$3\,x^2 - (2 + a^2)\,x^2 + a^2\,x + a^4 = 0$$

Since the admissible solution x must be within $[a, 1]$, due to the fact that $C_{\min} \leq C_x \leq C_{\max}$, we find that the only admissible solution is

$$x_{\text{opt}} = \frac{1 + \sqrt{1 + 3\,a^2}}{3}. \tag{3.17}$$

In the same way, if we assume a more realistic power function with $n = 3$, the polynomial to be solved becomes

$$4\,x^5 - (10 + 4\,a^2)\,x^4 + (5 + 12\,a^2)\,x^3 - (5\,a^2 + 2\,a^4)\,x^2 - a^4\,x + a^6$$

whose only admissible solution in $[a, 1]$ is

$$x_{\text{opt}} = \frac{5 - \sqrt{5} + \sqrt{2}\sqrt{5(3 - \sqrt{5}) - 8(1 - \sqrt{5})\,a^2}}{8} \tag{3.18}$$

From both Equations 3.17 and 3.18 it is possible to see that, under the conditions of this example, *the optimal value is always greater than C_{avg}*. Figure 3.5 shows the optimal value in both cases. On the x axis we showed the $\frac{C_{\min}}{C_{\max}}$ ratio, whereas the y axis represents the optimal number of cycles C_x provided before the processor mode switch (normalized with respect to $C_{\max}$).

Notice that the closer is $C_{\min}$ to $C_{\max}$, the closer is C_x to C_{avg} (therefore the improvement over the solution proposed by Zhu et al. [141] becomes smaller). In particular, when $C_{\min} = C_{\max}$, the task has a computation time $C_x = C_{\min} = C_{\max}$. In this case, the two schemes are equivalent because the task has a deterministic behaviour and the actual number of cycles required by the task is known.

For any other value of $C_{\min}$, however, our solution allows to save more energy than the sub-optimal solution proposed by Zhu and Mueller [141] (i.e., setting C_x equal to C_{avg}): providing C_{avg} cycles at speed α_L, in fact, would increase the average energy consumed in the period with respect to using our solution.

The actual number of cycles (i.e., the value of the $\frac{C_{\min}}{C_{\max}}$ ratio) depends by the particular task and there are not typical values. However, some studies have observed that the actual execution times C_x of tasks in real-world embedded systems can vary up to 87 percent with respect to their measured worst case execution times $C_{\max}$ [134].

3.4.2 Exponential Density

The probability density considered in the previous section is very simple and it allows to compute the exact pair (C_x, Q) that minimizes the average energy consumption. We consider now a more sophisticated density $f_C(c)$ that better captures the characteristics of real execution times. Without loss of generality, we normalize the number of cycles by $C_{\max}$ so that the possible values of cycles are in $[0, 1]$. As done before we set $a = \frac{C_{\min}}{C_{\max}}$.

We consider the following exponential p.d.f.:

$$f_C(c) = \begin{cases} \frac{1}{K} e^{\beta c}(1-c)(c-a) & \text{if } c \in [a, 1] \\ 0 & \text{otherwise} \end{cases} \tag{3.19}$$

where K is a proper constant such that $\int_a^1 f_C(c)\mathrm{d}c = 1$. The presence of β allows to alter the symmetry of the density. In fact, for negative values of β the density shifts to the left, meaning that values closer to $C_{\min}$ are more likely to happen. On the other hand, positive values of β means that execution cycles closer to $C_{\max}$ occur more frequently. Figure 3.6 shows the shape of some possible functions.

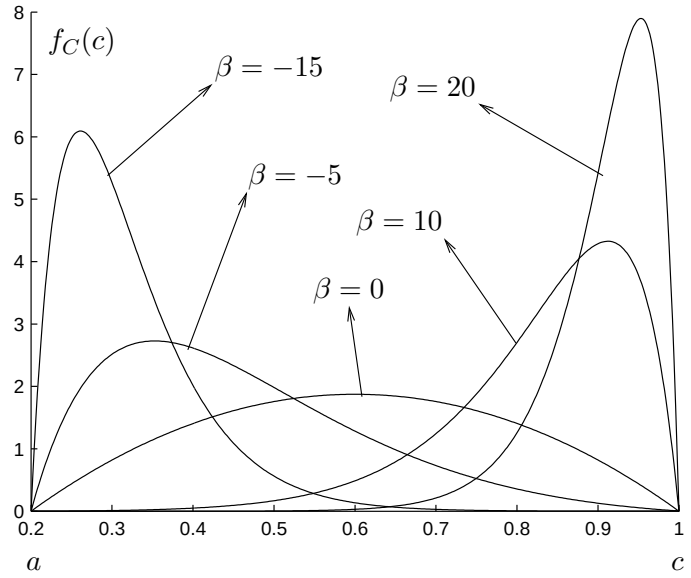

Figure 3.6: Exponential probability density functions.

Unfortunately, when dealing with exponential densities, the minimal energy (C_x, Q) pair can only be found by numerical approximation. We investigated the effect of the p.d.f. asymmetry onto the solution. The result is quite interesting. In Figure 3.7 we plot the ratios $\frac{C_x}{C_{\text{avg}}}$ and $\frac{Q}{T}$, assuming $a = \frac{C_{\min}}{C_{\max}} = 0.2$. The line above (i.e., $\frac{C_x}{C_{\text{avg}}}$) represents the percentage of cycles executed in the first part with respect to the average number of cycles. The line below (i.e, $\frac{Q}{T}$) represents the mode switch instant Q with respect to the task period T.

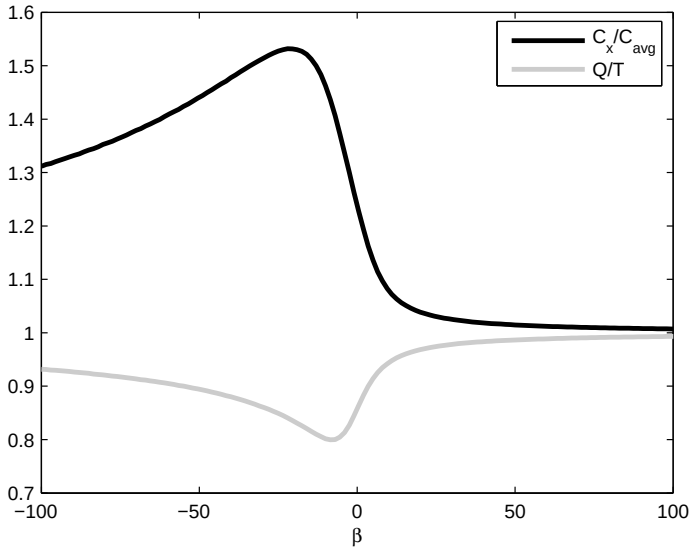

Figure 3.7: The optimal (C_x, Q) pair as function of the symmetry for exponential p.d.f.

A first result, also noticed for uniform density, is that the optimal C_x is always greater than C_{avg}. This fact is evidenced by the black curve which is always above 1. We also highlight that for big positive values of β (meaning that values closer to $C_{\max}$ are more likely to occur), C_x tends to C_{avg}.

3.4.3 Impact of the overheads

In this experiment, we evaluate the impact of the energy overhead e and the time overhead o on the energy consumed. For this purpose, we fix the p.d.f. $f_C(c)$ equal to the exponential density with $\beta = -50$ (we already explained the meaning of this parameter in Section 3.4.2), and the power function $p(\alpha) = \alpha^3$. Then, we vary the time overhead o and the energy overhead e and, for each value, we compute the average energy consumption E^{avg}.

The results are plotted in Figures 3.8 and 3.9. As expected, increasing the overheads results in an increase of the energy consumed.

The particular value $\beta = -50$ is just an example, and it does not affect the shape of the plot. We can show, in fact, that different values of β lead to similar (i.e., increasing) curves. The shapes are not identical because the value of β affects the value of C_{avg} which, in turn, affects the value of E^{avg}.

3.4.4 Idle power

As remarked at the end of Section 3.3.1, the energy consumed during the idle operating mode Λ_I can be taken into account by adjusting properly the parameters of the two modes

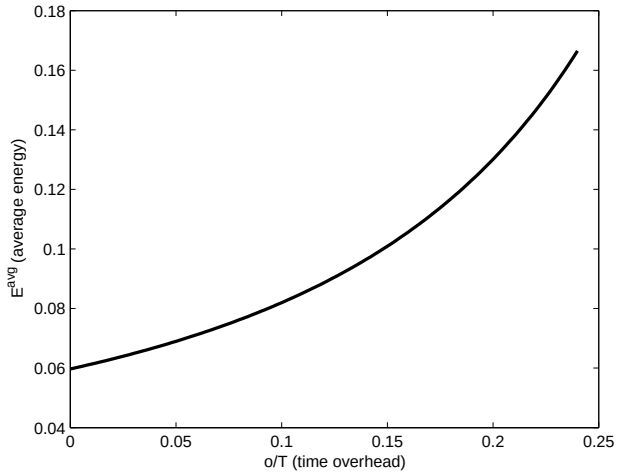

Figure 3.8: Impact of the time overhead o on the average energy consumption using an exponential p.d.f. with $\beta = -50$ and a cubic power function.

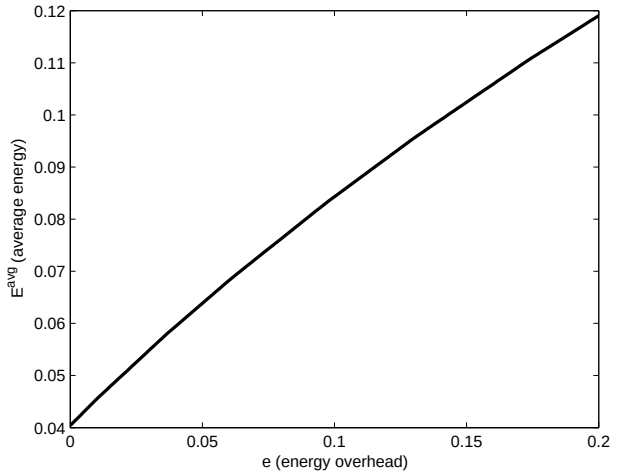

Figure 3.9: Impact of the energy overhead e on the average energy consumption using an exponential p.d.f. with $\beta = -50$ and a cubic power function.

Λ_L and Λ_H.

In this section we evaluate the impact of the power p_I consumed during the idle mode on the optimal values of the speed switch. In the experiments, the p.d.f. is set equal to the exponential density and the power function is assumed to be cubic. In Figures 3.10, 3.11 and 3.12 we plot the results.

The increase of the average finishing time f_{avg} shown in Figure 3.10 is justified by the fact that the processor consumes energy during the idle mode Λ_I as well. To not waste energy, it is more convenient to stay in active mode for a longer time by deferring the

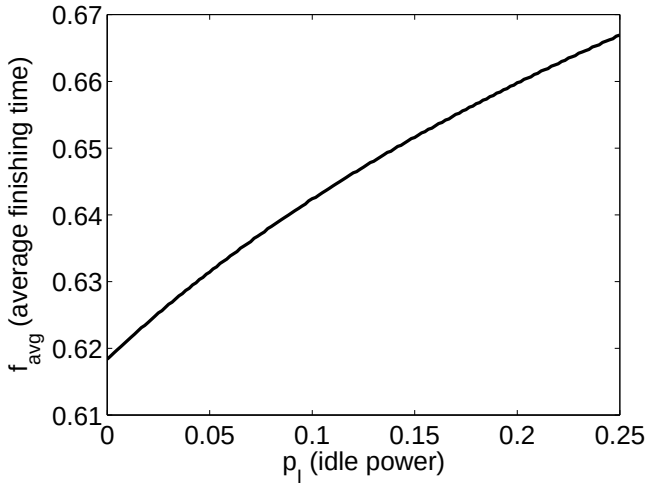

Figure 3.10: Impact of the idle power on the finishing time using an exponential p.d.f. and a cubic power function.

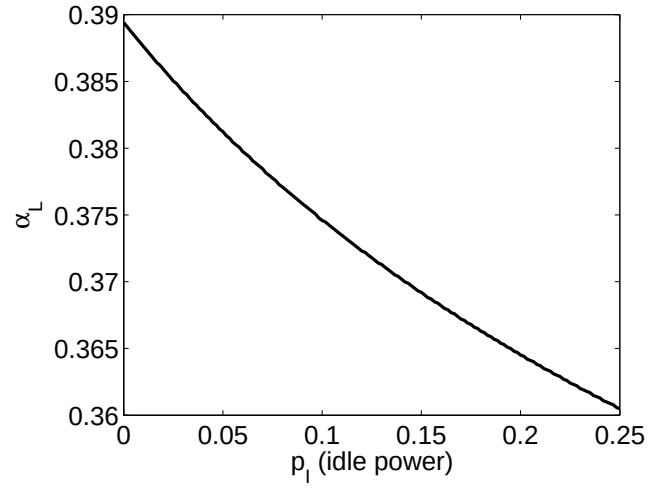

Figure 3.11: Impact of the idle power on the speed α_L using an exponential p.d.f. and a cubic power function.

average finishing time f_{avg}. Thus, the higher is the value of P_I, the higher is the average finishing time f_{avg} (i.e., the shape is monotonically non-decreasing). This is confirmed by our experimental simulations using the Matlab environment [129].

Clearly, as the average finishing time f_{avg} increases, it is possible to lower the speeds α_L and α_H, as confirmed by Figures 3.11 and 3.12.

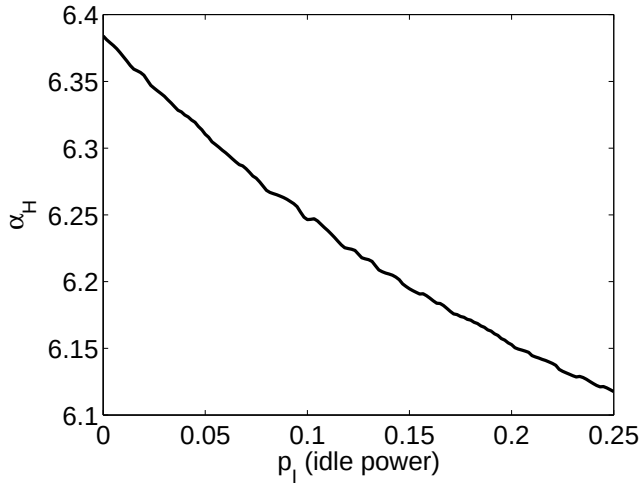

Figure 3.12: Impact of the idle power on the speed α_H using an exponential p.d.f. and a cubic power function.

3.5 Summary

In this chapter we showed that reducing energy consumption while still meeting real-time constraints is actually possible. In particular, deferring the work is a very effective technique when the actual number of cycles is unknown in advance.

We proposed a very general model for the processor that accounts for the idle power and for both the time and the energy overheads due to frequency transitions. We also computed the optimal values for the speed assignments and the transition instant within our energy management scheme. Our scheme is an improvement of the sub-optimal mechanism used in the DVS-EDF algorithm proposed by Zhu et al. [43, 141] (i.e., applying the scheme by Ishihara et al. [63] with average time for the first part and rely on the WCET for the total time).

We also studied how the overhead and the idle power affect the optimal values. Very interestingly, we showed that, in our model, the power consumed during the idle operating mode can be easily taken into account by adjusting the active operating modes of the processor.

Finally, we showed how our results can be applied to some significant cases (namely, uniform and exponential densities).

In the next chapter we will introduce the Resource Reservation technique, that will be used as basis for the development of our energy-aware scheduling algorithm, GRUB-PA, presented in Chapter 5.

Chapter 4

Resource Reservations

Since the GRUB-PA algorithm presented in Chapter 5 is based on the Resource Reservation framework (in particular, on the GRUB algorithm), we now recall some concepts about this technique and we briefly describe the original GRUB algorithm. The interested reader can refer to the original papers [76, 75, 77] for a more detailed presentation of the original algorithm.

During this chapter, we will also investigate some anomalies in the schedule generated by the CBS [6] and GRUB [76, 75, 77] algorithms and we will propose a novel algorithm, called HGRUB [11], which maintains the same features of CBS and GRUB but it is not affected by the problems described.

Throughout this chapter, we will assume that the processor speed is set to the maximum and is not changed. Then, in the next chapter, we will show how it is possible to extend the GRUB algorithm to exploit DVS.

4.1 An introduction to Resource Reservations

We are interested in developing an energy-aware scheduling algorithm that can handle both hard and soft, and periodic, sporadic and aperiodic tasks. The problem of mixing hard and soft real-time tasks can be efficiently solved by using the *Resource Reservation framework* [104, 39]. The basic idea behind the Resource Reservation technique consists on *reserving* a fraction of the processor time to time-sensitive applications through real-time scheduling. The mechanism works as follows. Each real-time task τ_i is assigned an abstract entity called *server* characterized by a *maximum budget* Q_i and a *period* P_i, the interpretation being that the task can use the processor for at least Q_i units of time every period P_i. In particular, the server *current budget* q_i is a value comprised between zero and Q_i which measures the amount of remaining processor time that the task τ_i can use during the current period P_i. Notice that the server period P_i may differ from the task period T_i depending on the needs of the system designer — the server acts as a periodic

task having $WCET = Q_i$ and period P_i, regardless of the task served by the server, that can be periodic, sporadic or aperiodic.

The goal of the system-wide scheduler is then to schedule run-time resources in such a way that each server is guaranteed a certain (quantifiable) level of service, with the exact guarantee depending upon the server parameters. That is, the server parameters represent its *contract* with the system, and the system global scheduler is obliged to fulfil its part of the contract by providing the level of service contracted [76].

Many server algorithms for both fixed priority (e.g. Polling Server, Deferrable Server, Sporadic Server) and dynamic priority schedulers (e.g., Constant Bandwidth Server, Constant Utilization Server, IRIS) [72, 122, 124, 6] have been proposed in the literature. Typically these algorithms differ in the way the budget is replenished and priorities and deadlines are updated. When the tasks execute less than expected, the remaining *slack time* can be used to reduce the response time of soft aperiodic tasks. Techniques for using this slack time are usually referred to as *reclamation techniques*. Examples of such reclamation techniques for resource reservation under dynamic scheduling have been already proposed [76, 33].

4.1.1 Temporal isolation

Multiprogrammed computer systems are expected to execute several tasks concurrently. When some (or all) of these tasks correspond to real-time applications, it is important that the underlying scheduling policy has the following features:

1. Each individual task should be *guaranteed* a certain level of service, and

2. There should be effective *isolation* among tasks — a misbehaving task should not be able to cause an unacceptable degradation in performance in other (well-behaving) tasks.

All resource reservation algorithms provide the *temporal isolation property*: the temporal behaviour of each task (i.e., its ability to meet its deadlines) is isolated from the rest of the system and it is not affected by the behaviour of the other tasks. If a task misbehaves and requires a large execution time, it cannot monopolize the processor. Thanks to the temporal isolation property, each task executes as it were on a slower dedicated processor, so that it is possible to provide real-time guarantees on a per-task basis. Such property is particularly useful when mixing hard and soft real-time tasks on the same system.

More formally, a Resource Reservation RSV_i on a resource $\mathcal{R}$ for a task τ_i is described by the tuple (Q_i, P_i), meaning that the task is *reserved* the resource $\mathcal{R}$ for a time Q_i every period P_i. Q_i is also called *maximum budget* of the reservation, and P_i is called *reservation period*. As stated in Chapter 1, we are considering the processor as the only resource $\mathcal{R}$

allocated to the real-time tasks. Furthermore, we restrict our attention to preemptive uniprocessor systems.

4.1.2 Qualitative comparison with Proportional share

As introduced in the previous section, the *utilization of the reservation* $U_i = \frac{Q_i}{P_i}$ represents the percentage of resource that we want to reserve to the task τ_i, whereas the reservation's period P_i represents the *temporal granularity* of the reservation.

Resource reservations present similarities with other classes of scheduling algorithms, like *proportional share* algorithms (EEVDF [127], WF^{2}Q [21], etc.) or p-fair [51]. Such algorithms usually fix a *scheduling quantum* for the whole system, and then assign each task a *weight* w_i. Each task receives a percentage of service proportional to

$$\frac{w_i}{\sum_{j \in Active} w_j}. \tag{4.1}$$

There are many differences between resource reservations and these scheduling techniques. The goal of proportional share algorithms is to emulate a fluid allocation system, like the Generalized Processor Sharing (GPS) [97], as close as possible. More formally, these algorithms bound the *lag* — i.e., the maximum difference in the service provided by the actual system and the ideal fluid system. The objective of resource reservations, instead, is to provide real-time execution (i.e., completion before deadline) to hard and soft real-time tasks.

Another objective of proportional share systems is to fairly distribute the bandwidth to all tasks in proportion to their weight. In other words, the allocation specification is always a *relative* parameter, and not absolute. Instead, fairness is not a concern for Resource Reservation algorithms.

In proportional share systems, each task is assigned a single parameter, the weight, whereas the scheduling quantum is a *global* system parameter. Therefore, in such systems, the *temporal granularity* is a system-wide property. Resource Reservations, instead, assign to each task two parameters, Q_i and P_i, respectively. In particular, it is possible to assign a different temporal granularity to each application, depending on its needs.

Typically, the additional degree of freedom of real-time systems allows to provide more guarantees about tasks timing constraints while decreasing the number of preemptions [11].

Assigning the values Q_i and P_i to each task in the system is not easy, of course. However, it is important to highlight that this thesis focuses on the scheduling algorithm, and not on the policies for dynamically assigning and modifying the scheduling and the reservation parameters. Many papers deal with heuristic algorithms and feedback control schemes for adapting the scheduling parameters to the need of the application [10, 95, 141, 39, 7]. We prefer to keep the scheduling algorithm separate from such policies because we

firmly believe that the problem of selecting the server parameters is a higher level problem that depends on the characteristics of the application.

4.1.3 The CBS class of schedulers

The first example of a Resource Reservation algorithm was the CPU Capacity Reserve proposed by Mercer and Tokuda [85]. According to this algorithm, each task is assigned a budget q_i that is decreased during task execution. The processor is scheduled according to Fixed Priority and a task is allowed to execute only when its budget is greater than zero. The budget is periodically recharged to Q_i every P_i units of time. This is basically the behaviour of the *Deferrable Server* (DS) algorithm [128], where each task is served by a dedicated server. Unfortunately, aperiodic activations and deactivations can break the reservation behaviour, as shown in [9].

The Constant Bandwidth Server (CBS) algorithm [6] is an algorithm belonging to the class of *aperiodic servers with dynamic priorities*. It derives inspiration from the service mechanisms proposed in the *Dynamic Sporadic Server* (DSS) [124, 47] and the *Total Bandwidth Server* (TBS) [123, 124], and uses dynamic priorities to correctly cope with dynamic aperiodic arrivals.

A limit of some aperiodic servers with dynamic priorities is that they rely on the knowledge of execution times of served aperiodic tasks. In some cases, though, the execution time of a task is unknown, or extremely variable from an instance to another (consider, for example, a MPEG player). In these cases, the use of a hard real-time system to manage this kind of applications would be unsuitable for two reasons:

- First, the worst case execution time (WCET) of the job could be much higher than its average execution time. Since the guarantees for hard real-time tasks are given on the basis of the WCET (and not on the basis of the average execution time), this kind of applications could cause an enormous waste of resources. In fact, the system is sized according to the WCET of each real-time task, and this leads to a very partial utilization of the performance that it could offer.

- Second, it is difficult to provide an exact evaluation of the WCET. The fact that the real-time guarantees depend on the evaluation of the WCET of each job, makes the hard real-time system weak respect to some mistake in this evaluation. If a job does not respect the evaluated execution time, another task could miss its deadline.

In the CBS algorithm, each task τ_i is scheduled through a server, which is characterized by the parameters Q_i and P_i described above and by two variables: the current budget q_i and a scheduling deadline d_i^s. The current budget q_i measures the processor time consumed by the task τ_i in the current period. It is comprised between 0 and Q_i, and periodically recharged to Q_i. The scheduling deadline, d_i^s, is used to keep all active servers ordered

in an Earliest Deadline First (EDF [79]) queue. The server with the earliest deadline is selected to be executed. One peculiarity of the CBS algorithm is the rule for recharging the budget when it is *"depleted"* (i.e., it reaches zero):

- When the budget q_i is depleted and the task has not completed its execution yet, the budget is *immediately* recharged, the server scheduling deadline is postponed to $d_i^s = d_i^s + P_i$, and the EDF queue is reordered. If, after postponing the server deadline, the server is still the earliest deadline server, the task continues to execute, otherwise the task is preempted. This behaviour is known as *soft reservation*.

CBS is not affected by the previous problem because it does not rely on an evaluation of the WCET. In fact, as soon as the task tries to execute more than its WCET, the budget of its server reaches zero, and the server deadline is postponed. Thus, the server may no longer be the earliest-deadline one. In this case, the scheduler preempts the task in execution and assigns the processor to the task served by the earliest deadline server.

4.1.4 Algorithms based on CBS

Over the past years, many algorithms based on the original CBS algorithm have been proposed. Here, we just describe those that are more closely related to our work. To better classify them, let first introduce two important characteristics of the algorithms:

- An algorithm implements *hard reservations* [104] if, when the server budget is depleted, the task is suspended until the next *replenishment time*. Notice that this rule is the opposite of the *soft reservation* rule of CBS (i.e., the server continues to execute until it is not the earliest deadline server anymore).

- An algorithm implements *reclamation* if it is able to reclaim the excess bandwidth from other servers executing less than their allocation, and/or by other unallocated bandwidth in the system.

The CASH (CApacity SHaring [34]) algorithm adds reclamation to the CBS by using a queue of budgets. When a task completes, the excess budget (and the corresponding server deadline) is inserted in a queue of budgets ordered by server deadlines (the CASH queue). A server is allowed to use all excess budgets from the queue with a deadline less than the current server deadline. CASH is able to reclaim only the excess bandwidth from servers handling periodic tasks, and it is very effective for hard and soft real-time tasks, but it cannot be used (at least in its current form) in systems with aperiodic tasks and where tasks can dynamically enter and leave the system.

The IRIS (Idle-time Reclaiming Improved Server [83]) algorithm adds the hard reservation property to CBS and uses such property to perform reclaiming. The idea is that, when the processor becomes idle, then it is possible to anticipate the replenishment time

of all servers. The algorithm is very simple and effective, but it might suffer from an excessive number of context switches, as briefly explained in Section 4.3.3.

The BEBS algorithm [19] is very similar to the IRIS algorithm. In addition to IRIS, BEBS dynamically modifies reservation periods and budgets to adapt the reservations to the needs of the applications.

4.1.5 Issues with CBS

The original CBS algorithm has been designed to allow scheduling mixtures of hard real-time, soft real-time and non real-time tasks so that the real-time tasks are not penalized by the other tasks running in the system. Due to the dynamic nature of many applications, it can happen that during some intervals of time a task requires more than the reserved resources (leading to a *transient overload*). In this situation, the original CBS will let the served task execute even for more than the reserved time (if such time is not used by other reservations), but to do so it will have to postpone its scheduling deadline. In the original paper [6] it is shown how the CBS can recover from a transient overload if there are intervals of time when the task requires less than the reserved time. However, experience with real systems showed that the time needed to recover may be too long [7, 39].

To understand in which conditions the original CBS behaviour can be suboptimal, consider two processor-intensive tasks τ_1 and τ_2 served by two servers with the same period, as shown in Figure 4.1. If the two tasks are activated simultaneously, the CBS guarantees that each of the two tasks gets the proper share of the processor in every interval of time.

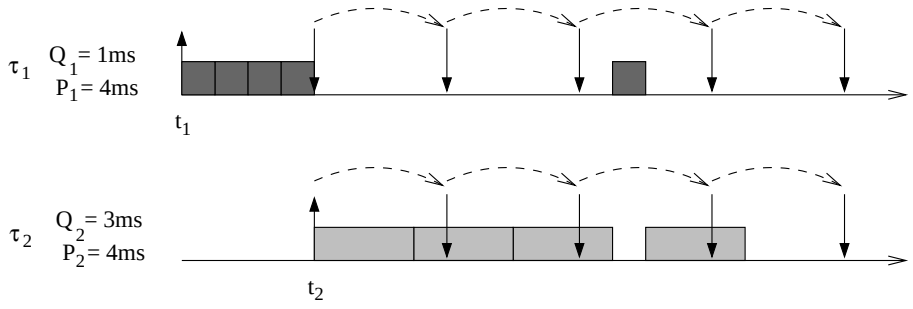

Figure 4.1: The "Greedy Task" problem of CBS.

However, if τ_1 is activated at time t_1 and τ_2 is activated at time $t_2 >> t_1$, according to the CBS rules, τ_1 is scheduled without interruptions in the interval $[t_1, t_2]$. As a consequence, the scheduling deadline d_1^s is postponed many times and can soon assume a large value. This phenomenon is known as *"deadline aging"*. When τ_2 is activated, its scheduling deadline d_2^s is much smaller than d_1^s, and τ_2 is continuously scheduled for a long interval of time, until the two scheduling deadlines d_1^s and d_2^s are comparable. As a consequence, τ_1 is not executed for a large interval of time. After that, the schedule

continues as expected and the processor is fairly shared between the two tasks. In the following, we will refer to this problem as the *"Greedy Task"* problem.

Intuitively, the problem is related to the fact that the task τ_1, starts consuming its *future* reservation, even if there is not other task executing. When τ_2 is activated, τ_1 has consumed much of its future reserved share, and so it cannot use the processor until the task τ_2 "catches up".

This issue is similar to the problem of the fairness of the WFQ algorithm, highlighted in [22, 21] — similarly to WFQ, the original CBS algorithm only cares about respecting deadlines, but allows the task to execute "too early". As noted by Bennet and Zhang [22], there are situations (for example, hierarchical scheduling) in which a better fairness is needed. Such an issue was addressed by the WF^2Q algorithm by introducing the *worst-case fairness* property [21].

4.2 The GRUB Algorithm

We now present the GRUB (Greedy Reclamation of Unused Bandwidth) algorithm [76, 75, 77] that will be used as basis for our energy-aware algorithm. GRUB, in turn, is based on the CBS algorithm [6], but it effectively solves the Greedy Task problem.

Several server-based schedulers (CBS included), can offer performance guarantees somewhat similar to the ones offered by GRUB. However, GRUB has an added feature that is not to be found in many of the other schedulers — an ability to *reclaim* unused processor capacity ("bandwidth") that is not used because some of the servers may have no outstanding jobs awaiting execution.

The GRUB algorithm is very general and does not assume the knowledge of tasks periodicity. More formally, the algorithm makes the following assumptions:

- The arrival times of the jobs (the $r_{i,j}$s) are not known *a priori*, but are only revealed on-line during system execution. Hence, GRUB's scheduling strategy cannot require knowledge of future arrival times. This assumption will hold also in our energy-aware scheduling algorithm based on GRUB, described in Chapter 5. Notice that many energy-aware algorithms for periodic task sets (like DRA [17, 18, 16] or RTDVS [98]) exploit the knowledge of future arrival times to simplify the solution.

- The exact execution requirements $c_{i,j}$ are also not known beforehand: they can only be determined by actually executing $\tau_{i,j}$ to completion.

The objective is to find a scheduling algorithm with the following properties:

- Provide timely service to *hard* and *soft* real-time tasks. In other words, it must provide the same guarantees as the CBS algorithm.

- Maximize the throughput. More specifically, the scheduler should never idle the processor when there is some active task; moreover, it should avoid unnecessary preemptions and context switches.

- Handle any kind of task — periodic, sporadic and aperiodic tasks. Of course, to be able to do schedulability analysis (i.e., to guarantee that all deadlines will be met), the designer must know the minimum interarrival times and the worst-case execution times of the tasks. However, if these values are not known, the designer can assign a certain amount of processor bandwidth to each task. GRUB, in fact, provides the temporal isolation feature: each task can be analyzed and guaranteed *in isolation* — i.e., without making any assumption on the other tasks in the system.

- The design of the original GRUB algorithm was driven by the interest in integrating the scheduling methodology based on Resource Reservation with traditional real-time scheduling — in particular, the goal was the design of a scheduler being a minor variant of the classical Earliest Deadline First (EDF) scheduling algorithm [79]. Therefore, the scheduling strategy was required to be as similar to EDF as possible. In particular, this rules out the use of scheduling strategies based upon "fair" processor-sharing, such as GPS [97] and its variants.

4.2.1 Formal model of the GRUB algorithm

In this Section, we introduce the models and the notation that will be used in the rest of the thesis.

Server Model

We consider a system comprised of n servers $S_1, S_2, \ldots, S_n$. Each server S_i is characterized by two parameters, (U_i, P_i), where U_i is the server bandwidth and P_i is the server period. The server bandwidth U_i represents the fraction of the total processor utilization assigned to the task modelled by the server — i.e., $U_i = \frac{Q_i}{P_i}$.

We restrict our attention to systems where all of these servers execute on a single shared processor (without loss of generality, this processor is assumed to have unit processing capacity) — we therefore require the sum of the processor shares of all the servers to be no more than one; i.e.,

$$\left(\sum_{i=1}^{n} U_i \right) \leq 1 \ .$$

GRUB Variables

For each server S_i in the system, algorithm GRUB maintains two variables: a *deadline* d_i^s and a *virtual time* V_i. Initially, these variables are both initialized to zero. Intuitively,

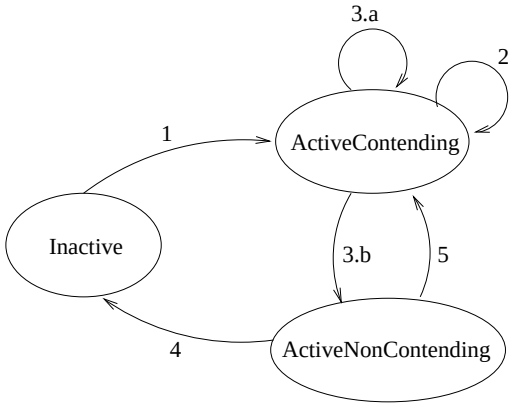

Figure 4.2: State transition diagram of GRUB.

the value of d_i^s at each instant is a measure of the *priority* that GRUB accords server S_i at that instant, and it is used to select which server is executing on the processor — GRUB essentially implements the EDF algorithm among all active servers. The value of the virtual time V_i at any instant is a measure of how much of the reserved bandwidth the server has already consumed at that time. Algorithm GRUB will attempt to update the value of V_i in such a manner that, *at each instant in time, server S_i has received the same amount of service that it would have received by time V_i if executing on a dedicated processor of capacity U_i*. The meaning of these variables will be clearer later.

Server states

At any instant in time during run-time, each server S_i is in one of three states: Inactive, ActiveContending, or ActiveNonContending. The initial state of each server is Inactive. Intuitively, at time t_o a server is in the ActiveContending state if it has some jobs awaiting execution at that time; in the ActiveNonContending state if it has completed all the jobs that arrived prior to t_o, but in doing so has "used up" its share of the processor until beyond t_o (i.e., its virtual time is greater than t_o); and in the Inactive state if it has no jobs awaiting execution at time t_o, and it has *not* used up its processor share beyond t_o.

At each instant in time, from among all servers that are in the ActiveContending state, algorithm GRUB chooses for execution (the next job of) the server S_i, whose deadline parameter d_i^s is the smallest. If there are no ActiveContending servers, then the processor is idled. While (a job of) S_i is executing, its virtual time V_i increases (the exact rate of this increase will be specified later); while S_i is not executing V_i does not change.

Algorithm rules

Certain (external and internal) events cause a server to change its state (see Figure 4.2).

1. If server S_i is in the Inactive state and a job $\tau_{i,j}$ arrives (at time-instant $r_{i,j}$), then the server variables are updated as follows

$$V_i \leftarrow r_{i,j}$$

$$d_i^s \leftarrow V_i + P_i$$

and server S_i enters the ActiveContending state.

2. If at any time the virtual time becomes equal to the deadline ($V_i == d_i^s$), then the deadline parameter is incremented by P_i:

$$d_i^s \leftarrow d_i^s + P_i$$

and the server remains in the ActiveContending state. Notice that this may cause S_i to be no longer the earliest-deadline active server, in which case it may yield control of the processor to an earlier-deadline server.

3. When a job $\tau_{i,j-1}$ of S_i completes (notice that S_i must then be in the ActiveContending state), the action taken depends upon whether the next job $\tau_{i,j}$ of S_i has already arrived.

   a) If so, then the deadline parameter d_i^s is updated as follows:

$$d_i^s \longleftarrow V_i + P_i \ ,$$

   and the server remains in the ActiveContending state.

   b) If there is no job of S_i awaiting execution, then server S_i changes state, and enters the ActiveNonContending state.

4. For a server S_i in ActiveNonContending state it is required that $V_i > t$ at any instant t. If this is not so (either immediately upon transiting into this state, or because time has elapsed but V_i does not change for servers in the ActiveNonContending state), then the server enters the Inactive state.

5. If a new job $\tau_{i,j}$ arrives while server S_i is in the ActiveNonContending state, then the deadline parameter d_i^s is updated as follows:

$$d_i^s \longleftarrow V_i + P_i \ ,$$

and server S_i returns to the ActiveContending state.

6. There is one additional possible state change — if the processor is ever idle, then *all* servers in the system return to their Inactive state.

Incrementing Virtual Time

It now remains to specify how the virtual time V_i of a server S_i changes when a job of S_i is executing. Let us first consider incrementing V_i at a rate $1/U_i$:

$$\frac{d}{dt}V_i = \begin{cases} \frac{1}{U_i} & \text{if (a job of) } S_i \text{ is executing,} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

— intuitively, executing S_i for one unit of time is equivalent to executing it for $1/U_i$ units of time on a dedicated processor of capacity U_i, thus we are updating V_i accordingly.

When V_i is updated as above (i.e., at a rate $1/U_i$ while S_i is executing, and not at all the rest of the time), algorithm GRUB is very similar to the CBS algorithm by Abeni and Buttazzo [6], and performance guarantees similar to the ones proved for CBS can be proved for algorithm GRUB as well. However, recall that one of the motivations driving the design of algorithm GRUB is being able of *reclaiming* processor bandwidth that may remain unused because some servers are in the Inactive state, and the goal of making *efficient* use of this reclaimed bandwidth. In using excess processor bandwidth, though, we must be very careful to not end up using any of the *future* capacity of currently inactive servers, since we cannot have any idea about when such servers will become active again.

Thus, algorithm GRUB maintains an additional global variable, called *total system utilization*, which, at each instant in time, is equal to the sum of the bandwidths U_i of the active servers:

$$U = \sum_{\substack{i=1 \\ \mathsf{S_i} \neq \mathsf{Inactive}}}^{n} U_i$$

where n is the number of servers in the system. This variable is initialized to zero and is updated every time a server enters or exits from state Inactive. In particular, when S_i exits from state Inactive U is increased by U_i, whereas when S_i enters state Inactive it is decreased by U_i.

Let $[t, t + \Delta t)$ denote an interval of time during which U does not change, and during which (a job of) S_i is executing. We assign the excess processor bandwidth during this interval — an amount equal to $\Delta t \cdot (1 - U)$ — to server S_i. Consequently, S_i has used an amount equal to

$$(\Delta t - \Delta t \cdot (1 - U)) = \Delta t \cdot U \tag{4.3}$$

of its own reserved processor bandwidth during this interval; equivalently, its virtual time V_i should increase by an amount equal to $\frac{\Delta t \cdot U}{U_i}$. Therefore, the rule for updating the virtual time of every server, is as follows:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{if (a job of) } S_i \text{ is executing,} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

The rate of increase of the virtual time is proportional to the current total bandwidth of the active servers, and is automatically adjusted depending on the current system load.

Observations The virtual time is an important variable as it gives a measure of the progress that a server task has done. Algorithm GRUB provides an abstraction of "slower processor": the task served by a server S_i with bandwidth U_i executes as it were executing on a dedicated slower processor with a minimum speed equal to U_i times the speed of the real processor.

Let us make a simple example to explain the way the algorithm updates the virtual time. Consider a server S_1 with bandwidth $U_1 = 0.25$ and period $P_1 = 20$msec. If the system is fully utilized (i.e., the total system bandwidth U is equal to 1), then Equation 4.4 tells us that the virtual time is increased at a rate of $1/0.25 = 4$. By looking at the algorithm rules, we see that the server executes for $P_1/4 = 5$msec before the server deadline is postponed.

In general, the bandwidth U_1 can be computed using some rule of thumb, or by performing a careful analysis of the application code. For our purposes, in this example, we assume that 5msec are enough to complete task's jobs in most cases.

However, suppose that, for some interval of time, the total system utilization U goes down to 0.75. Then, server S_1 can execute more than 5msec every period, because we can reclaim the spare bandwidth. According to Equation 4.4, the virtual time is increased at a rate of $0.75/0.25 = 3$. Therefore, if U is equal to 0.75 for the entire duration of the period P_1, server S_1 can execute for up to $P_1/3 = 6.66$msec within the period.

Thus, if our task sometimes requires more than 5msec to complete, it can take advantage of the reclaimed bandwidth and still execute inside the period boundary. This property can help us in setting the server bandwidth U_1 to a lower value for soft real-time tasks. For example, we can decide to set U_1 equal to the average bandwidth required by the task, and try to exploit the reclamation property of GRUB to dynamically get more bandwidth. Algorithm GRUB ensures that our application will take advantage of the spare bandwidth and execute more than $U_1 P_1$ whenever possible. This property of GRUB is called "reclamation", because we are giving the spare bandwidth to the needing servers.

4.2.2 Performance guarantees

The following theorems formally state the performance guarantees that can be proved for algorithm GRUB *vis à vis* the behaviour of each server when executing on a dedicated processor. For proofs of the following theorems, see the papers by Lipari et al. [76, 75].

Lemma 1 *At all times and for all servers S_i during run-time, the values of the variables*

V_i and d_i^s maintained by algorithm GRUB satisfy the following inequalities:

$$V_i \leq d_i^s \leq V_i + P_i \tag{4.5}$$

Theorem 2 *Given a set of servers $S_1, \ldots, S_n$, with $\sum_{i=1}^{n} U_i \leq 1$, then all servers execute within their deadlines, regardless of the served tasks. More formally, at each instant t, $\forall i = 1, \ldots, n \;\; d_i^s \geq t$.*

Theorem 3 *Suppose that job $\tau_{i,j}$ would begin execution at time-instant $R_{i,j}$, if all jobs of server S_i were executed on a dedicated processor of capacity U_i. In such a dedicated processor, $\tau_{i,j}$ would complete at time instant $F_{i,j} \stackrel{def}{=} R_{i,j} + (c_{i,j}/U_i)$, where $c_{i,j}$ denotes the execution requirement of $\tau_{i,j}$. If $\tau_{i,j}$ completes execution by time-instant $f_{i,j}$ when GRUB is used, then it is guaranteed that*

$$f_{i,j} \leq R_{i,j} + \left\lceil \frac{(c_{i,j}/U_i)}{P_i} \right\rceil \cdot P_i \; . \tag{4.6}$$

From the previous inequality, it follows that $f_{i,k} < F_{i,k} + P_i$. Thus, the period P_i represents the *granularity* of the time from the point of view of the server: by using algorithm GRUB, every job finishes at most P_i time units later than the completion time on a dedicated slower processor (the smaller is the value of P_i, the closer is the virtual time to the real time).

Moreover, the GRUB algorithm is able to serve hard real-time periodic tasks without any deadline miss, as stated by the following theorem.

Theorem 4 ([75]) *Let τ_i be a hard real-time periodic task with worst-case execution time C_i and period T_i. If task τ_i is assigned a server S_i with bandwidth $U_i \geq \frac{C_i}{T_i}$ and period $P_i = T_i$, then no deadline of τ_i will be missed.*

The interested reader can refer to the paper by Lipari et al. [76] for an evaluation of the algorithm GRUB and for a comparison with the CBS algorithm through a set of experiments. Here we just report that the schedules generated by Algorithm GRUB tend to have fewer context-switches than schedules generated by the CBS bandwidth-allocation scheme [76].

4.3 Coping with short periods

A problem presented by both the CBS and the GRUB algorithms occurs in presence of servers with different periods (see Figure 4.3). As discussed in Section 4.1.2, the server period is a measure of the "granularity" of the reservation — i.e., how often a task (or an application) is allocated the reserved budget. However, when the difference between

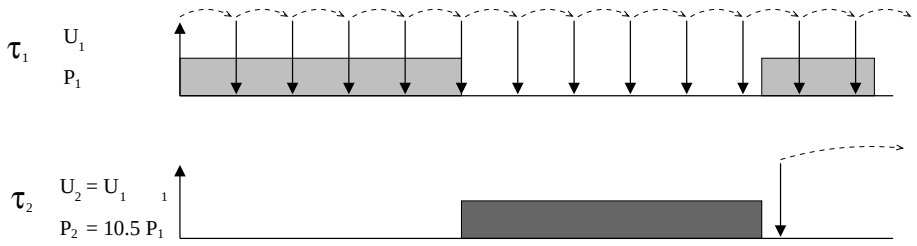

Figure 4.3: The "Short Period" problem of CBS and GRUB.

reservations' periods is too high it often happens that short period servers are not scheduled often enough, causing unexpected execution patterns. Again, this problem is related to the *soft reservation* rule of the CBS (inherited also by GRUB). When a short period reservation is depleted, the scheduling deadline is postponed but it might still remain the shortest one. Hence, the served task continues to execute for several server instances, as if it was served by a reservation with a longer period. In the following, we will refer to this problem as the *"Short Period"* problem.

4.3.1 The HGRUB algorithm

The Short Period problem can be effectively solved by adding Hard Reservation [104] to the original GRUB algorithm. In the original algorithm, when the virtual time of a running server becomes equal to the deadline, the server remains in the ActiveContending state and the server deadline parameter is incremented of P_i. To address the Short Period problem, we can add a further state, called Depleted and change the state transition 2 (see Figure 4.2) as follows:

2.a If at any time the virtual time of a server in ActiveContending state becomes equal to the deadline ($V_i == d_i^s$), then the server enters the Depleted state

2.b If at any time the virtual time of a server in Depleted state becomes equal to the current time ($V_i == t$), then the deadline parameter is incremented by P_i:

$$d_i^s \leftarrow d_i^s + P_i$$

and the server enters the ActiveContending state.

We call the modified algorithm HGRUB (Hard GRUB). The new state transition diagram is shown in Figure 4.4.

4.3.2 Formal properties of HGRUB

We now prove some interesting properties about the HGRUB algorithm. First, we define an ideal kind of task as follows.

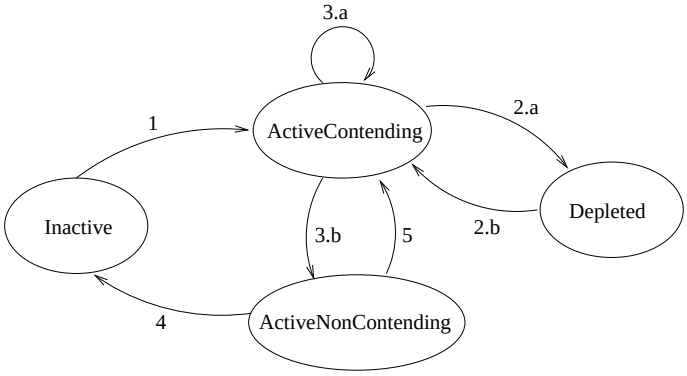

Figure 4.4: State transition diagram of HGRUB.

Definition 22 *A cpu-intensive task is a task that is continuously active and that never blocks (i.e., it continuously requires some computation from the processor).*

Scientific computation programs are typical examples of such cpu-intensive tasks. The goal of the scheduler for these tasks is to fairly allocate the processor time to all tasks, so that they can all progress concurrently. Clearly, this is only an ideal definition, since tasks (especially, real-time tasks) usually do block waiting on certain conditions (e.g., data from memory). However, the notion of cpu-intensive tasks allows us to prove some interesting properties of the HGRUB algorithm.

In the next Theorems we will provide an upper bound for the execution time of a cpu-intensive task in a system scheduled using HGRUB.

Theorem 5 *Consider a set of HGRUB servers, each one serving a cpu-intensive task. If the system workload does not change (i.e., if no server enters in or exits from the* Inactive *state), a server* (U_i, P_i) *can execute* at most *for a time equal to* $P_i \frac{U_i}{U}$ *inside a period* $[kP_i, (k+1)P_i]$ *with* $k \in \mathbb{N}$.

Proof: Consider the generic period $[0, P_i]$ of the server. At the beginning of the period, $V_i = t = 0$ and the current deadline is set equal to $d_i^s = P_i$.

During execution, the virtual time is updated as follows:

$$\frac{d}{dt} V_i = \frac{U}{U_i}$$

The server enters the Depleted state when $V_i = d_i^s$, and it remains in such state until the time is equal to V_i (i.e., until the end of the period).

Since the workload does not change in $[0, P_i]$, U is constant and the condition $V_i = d_i^s$ occurs after an execution time ΔT given by

$$(d_i^s - 0) = \frac{U}{U_i} \Delta T$$

Then, we have

$$\Delta T = \frac{U_i}{U} d_i^s = \frac{U_i}{U} P_i$$

Thus, the server enters the Depleted after a time ΔT and cannot execute for more than such time.

□

Theorem 6 *Consider a set of HGRUB servers, each one serving a cpu-intensive task, and an instant of time t_o. If the system workload does not change (i.e., if no server enters in or exits from the* Inactive *state), a server can execute* at most *for a period of time equal to $\lfloor \frac{t_o}{P_i} \rfloor \frac{P_i U_i}{U}$ before its scheduling deadline d_i^s is postponed beyond t_o.*

Proof: From Theorem 5, every time that the server executes for $\frac{P_i U_i}{U}$, it enters the Depleted state, and its scheduling deadline is postponed by P_i.

Consider the interval of time $[0, t_o]$. When $t = 0$ the scheduling deadline is set to $d_i^s = P_i$. After an execution time equal to $\lfloor \frac{t_o}{P_i} \rfloor \frac{P_i U_i}{U}$ the deadline of the server is postponed $\lfloor \frac{t_o}{P_i} \rfloor$ times, each time of an amount equal to P_i.

Therefore, after an execution time equal to $\lfloor \frac{t_o}{P_i} \rfloor \frac{P_i U_i}{U}$, the deadline is postponed at least to $\lfloor \frac{t_o}{P_i} \rfloor P_i + P_i \geq t_o$.

□

In the next Theorem we provide a lower bound for the execution time of a cpu-intensive task in a system scheduled using HGRUB.

Theorem 7 *Consider a set of HGRUB servers, each one serving a cpu-intensive task. If the system workload does not change (i.e., if no server enters in or exits from the* Inactive *state), each server S_i executes* at least *for a time $\frac{P_i U_i}{U}$ in a period $[kP_i, (k+1)P_i]$ with $k \in \mathbb{N}$.*

Proof: Consider the interval of time $[0, P_i]$. When $t = 0$, the scheduling deadline d_i^s of server S_i is set equal to P_i.

From Theorem 6, a generic server S_j with period P_j can execute at most for $\lfloor \frac{P_i}{P_j} \rfloor \frac{P_j U_j}{U}$ before its scheduling deadline is postponed beyond P_i (i.e., before S_i is given higher priority than S_j).

If we consider all servers, S_i has the highest priority at most after a time equal to

$$\sum_{k \neq i} \lfloor \frac{P_i}{P_k} \rfloor \frac{P_k U_k}{U}$$

At this time, the amount of time that S_i can execute before its scheduling deadline is equal to

$$P_i - \sum_{k \neq i} \lfloor \frac{P_i}{P_k} \rfloor \frac{P_k U_k}{U}$$

This value is greater or equal to

$$P_i - \sum_{k \neq i} (\frac{P_i}{P_k}) P_k \frac{U_k}{U} = P_i - \frac{P_i}{U} \sum_{k \neq i} U_k = \frac{P_i}{U}(U - \sum_{k \neq i} U_k) = P_i \frac{U_i}{U}$$

This is at least the amount of time that S_i can execute before its deadline.

□

Using the results of the previous Theorems (i.e, upper and lower bounds) we can now specify the exact amount of execution time guaranteed to a cpu-intensive task scheduled by HGRUB.

Theorem 8 *Consider a set of HGRUB servers, each one serving a cpu-intensive task. If the system workload does not change (i.e., if no server enters in or exits from the* Inactive *state), a task executes* exactly *for a time* $\frac{P_i U_i}{U}$ *in the server period* $[kP_i, (k+1)P_i]$ *with* $k \in \mathbb{N}$.

Proof: It follows directly from Theorems 6 and 7.

□

This result can be exploited, in turn, to prove that the scheduled generated by HGRUB for cpu-intensive tasks does not have any idle time (i.e., the processor always executes some task).

Theorem 9 *Consider a set of HGRUB servers, each one serving a cpu-intensive task. If the system workload does not change (i.e., if no server enters in or exits from the* Inactive *state), the resulting schedule does not have any idle time.*

Proof: Consider the hyperperiod

$$\Delta T = LCM_k\{P_k\}$$

where P_k is the period of the server S_k.

Since each server executes a cpu-intensive task, Theorem 8 states that within ΔT it executes for a time equal to $\frac{U_k}{U}\Delta T$.

If we consider all servers, the processor utilization in ΔT is

$$\sum_k \frac{U_k}{U}\Delta T = \frac{\Delta T}{U}\sum_k U_k = \Delta T$$

Therefore, there are not idle times.

□

Notice that this theorem does not assume any particular value for U or U_i. It only assumes that servers serve only cpu-intensive tasks, which, by definition, are always active (therefore, the value of U is constant).

Finally, it can be proved that HGRUB is a *fair* algorithm, meaning that it shares the reclaimed processor time fairly among servers serving cpu-intensive tasks.

Theorem 10 *Consider a set of HGRUB servers, each one serving a cpu-intensive task. If the system workload does not change (i.e., if no server enters in or exits from the* Inactive *state), the exceeding processor time is given to servers proportionally to their bandwidths (i.e.,* fairness*).*

Proof: Consider again the hyperperiod $\Delta T = LCM_k\{P_k\}$. If the generic server S_i executed exactly with a bandwidth U_i, then it would execute for $U_i\Delta T$ in an interval of time equal to ΔT.

If we consider all N servers, the spare processor time to be divided among them is

$$t_{spare} = \Delta T - \sum_i U_i \Delta T = (1 - U)\Delta T$$

Theorem 8 states that within the hyperperiod ΔT each server S_i executes for a time $\frac{U_i}{U}\Delta T$. Therefore, the amount of exceeding time that the generic server S_i has executed is equal to

$$\frac{U_i}{U}\Delta T - U_i\Delta T = (\frac{U_i}{U} - U_i)\Delta T = \frac{U_i}{U}(1 - U)\Delta T = \frac{U_i}{U}t_{spare}$$

This means that the exceeding processor time t_{spare} has been given to each server S_i proportionally to its bandwidth U_i.

□

4.3.3 Considerations about HGRUB

In this section, we discuss the properties of the HGRUB algorithm, compared to other similar algorithms.

Reclamation Similarly to algorithm GRUB, Algorithm HGRUB reclaims unused bandwidth from tasks (or applications) using less than reserved as well as from free bandwidth in the system. This is done through the rule for updating the budget while the server is executing. When a server executes for Δt units of time, the CBS algorithm (and all other variations like IRIS), update the server budget as $q_i = q_i - \Delta t$, while the GRUB and HGRUB algorithm update the server budget as $q_i = q_i - \Delta_t * U$, where $U \leq 1$ is the sum of the servers that are not in the Inactive state.

In general, this leads to less preemptions than other reclaiming algorithms like IRIS, BEBS and CASH. In particular, IRIS performs reclamation through identification of idle times, but the *average* number of preemptions of IRIS is higher than with GRUB. In fact, when IRIS reclaims the unused bandwidth, there is a context switch at most every C_i, due to the hard reservation rule. When GRUB performs the reclamation, instead, the server

budget is increased and set equal to $\frac{C_i}{U}$. Thus, if U is less than 1, the server can execute for a longer time before experiencing a context switch.

Support for hierarchical applications As explained by Lipari et al. [77], hierarchical scheduling requires the hard reservation feature to work properly.

As a concrete example, consider a system based on CBS with two real-time tasks, τ_1 and τ_2, with guaranteed bandwidth U_1 and U_2, respectively. Suppose that task τ_1 has two internal threads, τ_1^1 and τ_1^2, with guaranteed bandwidth U_1^1 and U_1^2 (i.e., $U_1^1 + U_1^2 = U_1$). Finally, suppose that thread τ_1^1 is currently not active while thread τ_1^2 is cpu-intensive. Therefore, task τ_1 is always active and ready for execution: any time it receives the processor, it executes the internal thread τ_1^2 (because τ_1^1 is currently not active) and its server deadline is postponed according to the CBS rule. However, when thread τ_1^2 wakes up, it cannot execute immediately because the deadline of its server has been postponed by the execution of τ_1^1 (i.e., deadline aging).

A hard reservation server like HGRUB can be safely used for solving this kind of problem and implementing a hierarchical scheduling strategy [11].

Complexity and performance The HGRUB algorithm has the same complexity of algorithm CBS. As a matter of fact, the main practical difference between HGRUB and CBS is the use of an additional timer for keeping track of the replenishment time (transition 2.b), and the rule for updating the budget. The complexity of CBS is $O(n \log n)$, where n is the maximum number of active servers at any time. This is because at some time t the algorithm could have to handle at most n transitions, and each transition can require a queueing operation in a ordered queue of server descriptors, which in turns has a complexity of $\log n$. Remind that similar proportional share algorithms have the same complexity.

Also, HGRUB presents very good performance compared to other algorithms because it is a work-conserving algorithm and it is possible to use different temporal granularities for different servers.

Fairness Finally, note that HGRUB is not "fair" in distributing the excess bandwidth across tasks different than cpu-intensive ones. In some pathological cases, in fact, it may happen that all the excess bandwidth is given to one single server. However, in long intervals of time, if tasks are independent from each other, the bandwidth is distributed to tasks in proportion to their bandwidth. Although this is only a statistical evidence, it may be enough for most cases.

4.4 Summary

In this chapter we introduced the idea behind the Resource Reservation technique, which provides the temporal isolation property. We described the GRUB algorithm, which adds reclaiming to the original CBS algorithm. We explained how GRUB is capable of scheduling both hard and soft, and periodic, sporadic and aperiodic real-time tasks on the same system. Finally, we provided some considerations about the schedule produced by GRUB and we proposed the HGRUB algorithm as possible solution to the "Short Period" problem.

In the next chapter we will present our energy-aware real-time scheduling algorithm, GRUB-PA, based on the GRUB algorithm.

Chapter 5

The GRUB-PA Algorithm

In Chapter 4, we have introduced the Resource Reservation technique and we have described the GRUB algorithm [76, 75, 77].

In this chapter, we will modify GRUB for energy-aware scheduling. The novel algorithm is called GRUB-PA (*Greedy Reclamation of Unused Bandwidth—Power Aware*).

Like GRUB, our energy-aware algorithm is based on the Resource Reservation framework, therefore it can support both hard and soft real-time tasks. Moreover, unlike most energy-aware algorithms, GRUB-PA is able to deal with periodic, sporadic and even aperiodic tasks. Of course, to be able to do schedulability analysis, the designer must know the minimum interarrival times and the worst-case execution times of the tasks. However, since our algorithm provides temporal isolation, each task can be analyzed and guaranteed *in isolation* — i.e., without making any assumption on the other tasks in the system.

Notice that, since GRUB reclaiming and Hard Reservation behaviour are orthogonal properties, the extensions that we will discuss for the GRUB can be done to HGRUB as well.

The results shown in this chapter have been already presented at some conferences [111, 112] and have been published on the *IEEE Transactions on Computers* journal [113].

5.1 Introduction

Intuitively, the problem of reclaiming the spare bandwidth is similar to the problem of energy-aware scheduling. We can divide both problems into two parts. The first part consists on identifying the spare bandwidth (or the slack time) in the system, whereas the second part consists on deciding how to use the spare bandwidth. The first part of the problem is common to both the bandwidth reclamation and the energy-aware scheduling problems. The second part, instead, differs radically: in the reclamation problem, the goal is to use the spare time to anticipate the execution time of aperiodic tasks, whereas in the energy-aware scheduling problem the goal is to lower the processor frequency as

much as possible. We believe that many reclamation algorithms can be used as energy-aware schedulers by modifying their "second part". In this chapter, we modify GRUB for energy-aware scheduling following the previous idea.

Processor Model

We assume that the tasks are executed on a single processor with a variable operating frequency. Many energy-aware algorithms make the assumption of continuous frequency scaling, even though no existing processor can vary its frequency with continuity. In fact, all processors that support DVS provide a set of operating modes, each one characterized by a value of frequency and voltage [130, 131, 55, 56, 58, 57, 59, 60].

We assume that the processor can provide M frequencies, $\phi_1, \ldots, \phi_M$, in increasing order. A supply voltage $V_{DD-1}, \ldots, V_{DD-M}$ and a normalized processor "speed" $\overline{U}_1, \ldots, \overline{U}_M$ are associated to each frequency, again in increasing order, with $\overline{U}_M = 1$. The computation times of the tasks are relative to the maximum operating speed, $\overline{U}_M = 1$, and they vary linearly with the processor speed: therefore, if a job executes for $c_{i,j}$ units of time when the processor speed is 1, it executes for $c_{i,j}/\overline{U}_k$ when the processor speed is set equal to $\overline{U}_k$. It is important to note that we are implicitly assuming that the execution time of a task varies linearly with the processor frequency. In Section 5.4.2 this assumption will be validated experimentally on a real embedded system for audio applications.

One issue that must be taken into careful consideration is the overhead of changing frequency. Changing frequency is not "free", as most processors need some time to adjust to the new frequency. The duration of this transitory is variable, and varies a lot from processor to processor. For example, on the Intel PXA250 it can go up to $500\mu sec$ [58, 57]. Even though in many soft real-time applications this can be considered negligible, it should not be ignored. We will show how to account for this delay in the GRUB-PA algorithm in Section 5.2.4.

The presence of an energy overhead at every frequency switch is undeniable as well. This overhead depends on the particular kind of processor the algorithm is running on, and it is quite difficult to estimate and measure. Hence, we decided to not explicitly take into account this energy overhead in this model. However, in Section 5.2.4 we will devise a technique to limit the number of switches in an interval of time, therefore limiting the maximum amount of energy spent for switching frequency.

5.2 Algorithm GRUB-PA

As first step, let us assume that the processor speed can be varied continuously, from a maximum speed factor of 1 (i.e., the processor works at its maximum speed) to a minimum of 0 (i.e., processor halted), and that the time to change speed is negligible. We will relax

these simplifying assumptions in Sections 5.2.3 and 5.2.4.

As explained previously, GRUB maintains a global variable U that is the sum of the bandwidths of all servers that are not in the Inactive state. *The key idea is that, if we set the speed factor of the processor to be equal to U, no server will miss its deadline.* This idea is similar to the one on which the DVSST algorithm [101] is based (refer to Section 2.4.6 for a description of the algorithm). However, GRUB-PA updates the variable U in a more effective way, allowing additional power saving also in the case of periodic tasks, as shown in Section 5.2.1.

The original GRUB algorithm can be divided into two different parts: a set of rules for identifying the spare bandwidth $(1 - U)$, and a set of rules for reassigning the spare bandwidth. The second part can be adapted for energy-aware scheduling. In practice, if the processor is not fully utilized ($U < 1$) the exceeding bandwidth $(1 - U)$ can be used in two ways:

1. To execute the active servers for a longer time, so that they can execute faster and finish earlier. This is the "reclamation" property, and it is the original goal for which the GRUB algorithm was designed.

2. To slow down the processor. Each active server will execute for a longer time, but at a slower speed. The net effect is that its performance is not degraded (i.e., the number of processor cycles guaranteed to the server remains the same).

The reclamation rule in GRUB is given by the following Equation:

$$\frac{d}{dt}V_i = \begin{cases} \frac{U}{U_i} & \text{if (a job of) } S_i \text{ is executing,} \\ 0 & \text{otherwise} \end{cases}$$

Thus, the increment in the virtual time depends on the amount of bandwidth actually used in the system. This rule can also be used in the energy-aware part to automatically adapt the server bandwidth to the new frequency. Moreover, we need an additional rule that sets the processor speed equal to U whenever a server goes in (or leaves) the Inactive state.

Hence, in the new GRUB-PA algorithm, state transitions 1 and 4 (see Figure 5.1) are modified as follows:

1. When a job $\tau_{i,j}$ arrives at time instant $r_{i,j}$, the following variables are updated:

$$\begin{aligned} V_i &\leftarrow r_{i,j} \\ d_i^s &\leftarrow V_i + P_i \\ U &\leftarrow U + U_i \end{aligned}$$

Moreover, the processor speed is set equal to U.

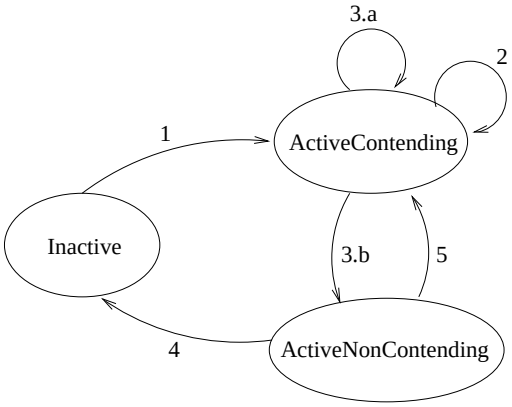

Figure 5.1: State transition diagram of GRUB.

4. When a server is in the ActiveNonContending state and $V_i = t$, then the server goes in the Inactive state and the system utilization is updated as follows:

$$U \leftarrow U - U_i$$

Moreover, the processor speed is set equal to U.

5.2.1 An example

In this section, we present a complete example showing how the GRUB-PA algorithm updates the processor speed depending on the bandwidth of the active servers. Consider a system consisting of two tasks. Task τ_1 is a sporadic task with minimum interarrival time $T_1 = 8$ and computation time varying between 2 and 4. This task is assigned a server with $U_1 = 0.5$ and $P_1 = 8$. The second task, τ_2, is a periodic task with period $T_2 = 10$ and constant execution time equal to 5. τ_2 is assigned a server with $U_2 = 0.5$ and $P_2 = 10$.

Suppose that the first job of task τ_1 arrives at time $t = 0$ requesting 2 units of computation time; the second job of τ_1 arrives at time $t = 12$ with computation time equal to 3. The resulting schedule is shown in Figure 5.2. As usual, the upward arrows denote the arrival times, while the downward arrows denote the deadlines. The plot under the schedule reports the variations of variable U during system evolution. In this case, we assume that deadline ties are broken in favor of the task with a lower index. However, in general ties can be broken arbitrarily.

Initially, all servers are active, so $U = 1$ and the processor speed $\overline{U}$ is set equal to 1. At time $t = 0$, task τ_1 is selected to execute, since the deadline of the server $d_1^s = 8$ is the earliest server deadline. The task executes until $t = 2$, when it completes. At this time, the virtual time is $V_1 = 2/U_1 = 4$, so the server goes into the ActiveNonContending state. Then, task τ_2 starts executing, and it executes for 2 time units until $t = 4$. At this time, the first server changes state from ActiveNonContending to Inactive: the total bandwidth

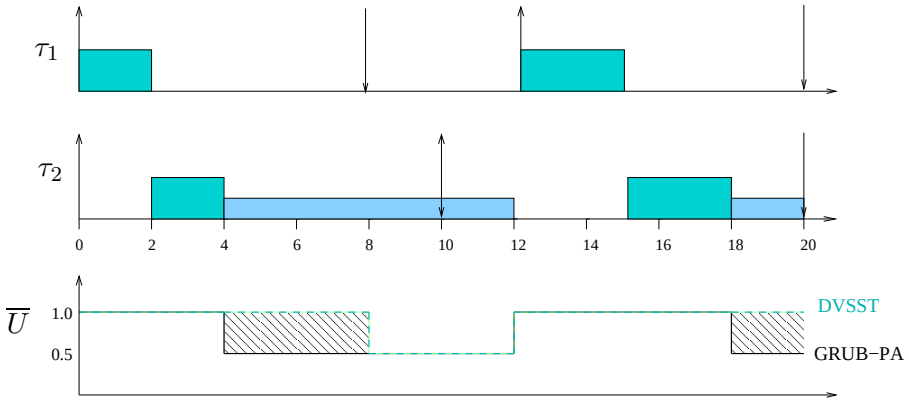

Figure 5.2: An example of schedule produced by GRUB-PA.

of all active servers is decreased to $U = U - U_1 = 0.5$, so the processor speed can be slowed down to $\overline{U} = 0.5$. Then, task τ_2 can continue executing at half the speed. However, its virtual time V_2 is also increased at half the speed: for each unit of execution, the virtual time will now increase at a rate of $dV_2 = dt\frac{U}{U_2} = dt$. Therefore, task τ_2 can now execute for 6 units of time, which correspond to 3 more units of the execution time at maximum speed, and complete just by the deadline at 10. However, at time $t = 10$ another job of task τ_2 arrives, so the second server remains in the ActiveContending state and τ_2 resumes execution at half the speed.

At time $t = 12$ the second job of τ_1 is activated. The server becomes ActiveContending and $U = U + U_1 = 1$. Therefore, the processor speed is again raised to $\overline{U} = 1$ and task τ_1 can start executing (as it is the one with the earliest server's deadline).

Notice that the mechanism used by the GRUB-PA algorithm is very similar to the one used by the DVSST algorithm [101]: they both use a global variable U to set the processor speed. However, there is a difference in the instant *when* the variable is updated. The DVSST algorithm does not keep track of the actual execution time of the tasks. Therefore, it can only subtract the bandwidth of a completed task *at task's deadline*. In the example above, even if task τ_1 completes by time $t = 2$, the DVSST algorithm must wait until time $t = 8$ to lower the processor speed. Instead, algorithm GRUB-PA can anticipate this time at $t = 4$ as it explicitly takes into account the fact that task τ_1 has executed less than expected. The GRUB-PA algorithm always anticipates this time with respect to algorithm DVSST, resulting in a larger amount of saved energy.

The difference in the setting of the processor speed made by the two algorithms in this particular example is shown in Figure 5.2. The schedule is represented using the typical GANNT chart. The assignment of a task to the processor is represented by a filled rectangular box, where the height of the box (i.e., vertical axis) represents the processor speed at which the task is executed. The oblique lines represent the difference in processor

speed between GRUB-PA and DVSST.

5.2.2 Properties of GRUB-PA

In this section, we formally prove that Theorem 2 (stating that all GRUB servers execute within their deadlines, regardless of the served tasks) is valid for the algorithm GRUB-PA as well. First, we define an ideal algorithm GPS-PA (which stands for *Generalized Processor Sharing - Power Aware*) that allocates the processor in proportion to the bandwidths U_i of the active tasks.

Definition 23 *Algorithm GPS-PA is a fluid algorithm [97] that adjusts the processor frequency and allocates the processor to tasks according to the following rules:*

- *The processor speed is set equal to the sum of the utilization of all active tasks;*
- *For every interval Δt, the processor is allocated to all active tasks in proportion to their utilization.*

Clearly, GPS-PA is an ideal algorithm and cannot be implemented in practice, since it is impossible to allocate any infinitesimally small interval Δt to different tasks in proportion to their utilizations. GPS-PA will be used only as a reference algorithm for GRUB-PA.

GPS-PA has the following interesting properties.

Lemma 11 *Under algorithm GPS-PA, under the constraint that the sum of the utilization of all tasks is upper bounded by 1, all jobs will complete exactly at time $F_{i,j} = max(R_{i,j}, F_{i,j-1}) + \frac{c_{i,j}}{U_i}$.*

Proof: At all times, GPS-PA sets the processor speed equal to the sum of the bandwidths of all active tasks U. Thus, the processor is allocated to each task in proportion to its bandwidth. The rate of execution of task τ_i is constant and equal to

$$R_i = \frac{U \cdot U_i}{U} = U_i.$$

Therefore, the finishing time $F_{i,j}$ of job $\tau_{i,j}$ does not depend on the presence of other tasks in the system, and each task executes as it was on a slower dedicated processor of constant speed U_i. The finishing time of the first job of task τ_i is $F_{i,0} = \frac{c_{i,0}}{U_i}$. Any successive job of τ_i starts at the latest time between the finishing time of the previous job and its arrival time. Hence, the lemma is proved.

□

Now, we divide each job in one or more subjobs, each one of maximum length $U_i \cdot P_i$.

- A job $\tau_{i,j}$ with $c_{i,j} \leq U_i \cdot P_i$ is transformed into a subjob $\tau_{i,j}(1)$, with the same execution time, the same arrival time, and deadline $d_{i,j}(1) = r_{i,j} + P_i$;

- A job $\tau_{i,j}$ with $c_{i,j} > U_i \cdot P_i$ is divided into $K = \left\lceil \frac{c_{i,j}}{U_i P_i} \right\rceil$ subjobs $\tau_{i,j}(1), \ldots, \tau_{i,j}(K)$. All subjobs have execution time $c_{i,j}(k) = U_i \cdot P_i$, except the last one, which can be shorter. Each subjob is assigned an arrival time and a deadline: the first subjob is assigned an arrival time equal to the arrival time of the original job, and a deadline $d_{i,j}(1) = r_{i,j} + P_i$. The following ones are assigned arrival times and deadlines as follows:

$$r_{i,j}(k) = d_{i,j}(k-1) \qquad d_{i,j}(k) = r_{i,j}(k) + P_i$$

Corollary 12 *For each subjob, $F_{i,j}(k) \leq d_{i,j}(k)$.*

Proof: This splitting operation does not influence the behaviour of algorithm GPS-PA. Therefore, the corollary trivially follows from Lemma 11.

□

At this point, we will show that the schedule generated by algorithm GPS-PA can be "transformed" into the schedule generated by GRUB-PA maintaining certain important properties. The transformation is done by following a well-known technique described by Coffman and Denning [37, Chapter 3]. First we transform the schedule generated by GPS-PA into a non-fluid schedule. Then, we transform this second schedule into the schedule generated by GRUB-PA.

Definition 24 (Job Transformation) *Let $\sigma_f(t)$ be the schedule generated by GPS-PA. It is a function with multiple values: for every time t, $\sigma_f(t)$ is the set of executing subjobs, that coincides with the set of active subjobs.*

Now, we generate a function $\sigma_i(t)$ in the following way. For every t, let $[t_1, t_2]$ be a maximal interval containing t, in which $\sigma_f(t)$ is constant and no subjob completes in (t_1, t_2)[1]. It follows that, either a subjob completes in t_2, or a subjob is activated in t_2. Let x be the number of jobs active in (t_1, t_2).

Then, we divide interval $[t_1, t_2]$ into x subintervals, one for each active subjob $\tau_{i,j}(k)$, each one of length $(t_2 - t_1) \cdot U_i$. Then, function $\sigma_i(t)$ assumes value $\tau_{i,j}(k)$ in the corresponding subinterval.

Moreover, in the new schedule the processor frequency is changed at the same instants as in schedule $\sigma_f(t)$.

By construction, the finishing times of any subjob in $\sigma_i(t)$ is not greater than the finishing times of the same subjob in $\sigma_f(t)$. Therefore, the following corollary is trivially proved.

Corollary 13 *No subjob misses its deadline in schedule $\sigma_i(t)$.*

Lemma 14 *GRUB-PA changes frequency at the same instants of time as GPS-PA.*

[1] As usual, symbols [] denote a closed time interval, and symbols () denote an open time interval.

Proof: In GPS-PA, frequency is updated at arrival times of the jobs $R_{i,j}$ or at the finishing times $F_{i,j}$. At each instant t, the virtual time $V_i(t)$ in GRUB-PA corresponds to the instant of time in the GPS-PA in which the task has received the same amount of service as in the GRUB-PA.

GRUB-PA changes the U (and possibly the frequency) when the task arrives (i.e., $V(t) = t$) or when the task goes into the Inactive state (again, $V(t) = t$). In the first case, we have $V(t) = R_{i,j} = t$. In the second case, we have $V(t) = t = F_{i,j}$. Hence, the lemma is proved.

□

Finally, the last step of our demonstration is to transform the schedule $\sigma_i(t)$ into the schedule generated by GRUB-PA.

Theorem 15 *Server S_i never misses its deadline. In other words, at any instant t, the server deadline is always greater than t.*

Proof: We use a well-known technique by Dertouzos [41], originally used for proving the optimality of EDF. Given a feasible schedule $\sigma_i(t)$ as obtained by the technique described in Definition 24, by the optimality of EDF, with an exchange procedure, we can obtain a feasible schedule $\sigma(t)$ in which the subjobs are scheduled in EDF order. Notice that the deadlines of the subjobs are equal to the deadlines of the servers as assigned by GRUB-PA. Therefore, the schedule is the same as obtained by GRUB-PA, as GRUB essentially performs EDF on the subjobs.

Since $\sigma(t)$ is feasible, the theorem follows.

□

Thus, for the GRUB-PA algorithm too it is guaranteed that no server will miss its deadlines. In other words, the change of processor speed introduced in GRUB-PA (see Section 5.2) does not affect the guarantees that have been proved for the GRUB algorithm in Chapter 4.

5.2.3 Discrete frequencies

No existing processor can vary its frequency with continuity. All processors that support DVS provide a discrete set of frequencies [131, 58, 57, 59, 60, 56, 55]. Correspondingly, we can set some "thresholds" on the values of the total system bandwidth. Suppose that the processor supports M different frequencies $\phi_1, \ldots, \phi_M$. We can compute $\overline{U}_1, \ldots, \overline{U}_M$ different values of the bandwidth. If $U(t)$ is comprised in $(\overline{U}_k, \overline{U}_{k+1}]$ for some k, then the processor frequency is set equal to ϕ_{k+1}.

It is easy to see that, by using this simple approach, the properties of the GRUB-PA algorithm continue to hold. In fact, the actual speed of the processor is always set to a value $\overline{U}_{k+1}$ greater than or equal to the theoretical desired bandwidth U.

Unfortunately, the net effect of this approach is that some energy is wasted as the desired frequency is always approximated by a higher frequency. One possibility would be to apply a technique similar to the one proposed by Ishihara et al. [63]. It consists in alternating between the two frequencies ϕ_k and ϕ_{k+1} so that the average utilization is equal to the desired utilization. If the requested bandwidth is constant over an interval $[t_1, t_2]$, it is possible to compute an instant $t_1 < t' < t_2$ so that in interval $[t_1, t']$ the processor frequency is set equal to ϕ_{k+1} and in interval $[t', t_2]$ the processor frequency is set equal to ϕ_k. The original technique was devised as an off-line algorithm for periodic tasks with constant execution time. This idea has also been recently proposed by Bini et al. [25] in the context of static DVS. Their methodology consists in computing the minimum theoretical processor speed that, if constantly applied to the system, makes the task set schedulable. Then, if the corresponding frequency is not available in the set of processor frequencies, the methodology selects two available frequencies that will be alternated in a *duty cycle*.

Applying such methodology to our model is not trivial. In GRUB-PA, tasks may be periodic or aperiodic and the system dynamically varies the value of U at instants of times that cannot be predicted a-priori. In particular, it is not possible to know how long the system will maintain a certain value of U. Therefore, only a clairvoyant algorithm can find the *optimal* way of alternating the two frequencies ϕ_k and ϕ_{k+1}. Nevertheless, it is possible to approximate the previous technique in a conservative way, for example by dividing the time line into intervals of arbitrary length and applying the previous technique. Since we always start with the highest frequency, we are guaranteed that in case the bandwidth changes, no deadline will be missed. The smaller are such intervals, the more precise is the approximation. However, the overheads in terms of time and energy of additional frequency switches need to be carefully considered. We are currently investigating the possibility of finding such suboptimal algorithm for the above problem.

From a practical point of view, observe that the waste of energy is less evident as the number of available processor frequencies increases. Some modern processors provide a large number of combinations voltage/frequencies (the Transmeta Crusoe TM5800 [131], for instance, has seven possible frequencies) and, therefore, the difference between desired frequency and actual frequency can be very little.

As a final consideration, it is important to note that the GRUB-PA algorithm, as well as GRUB, is able to reclaim spare capacity for soft real-time tasks. Suppose that the processor speed is set equal to $\overline{U}_{k+1}$, while the actual bandwidth is $U < \overline{U}_{k+1}$. Then, the difference $\overline{U}_{k+1} - U$ is automatically accounted by the algorithm as spare bandwidth, and reclaimed for the tasks that need to execute more than their assigned bandwidth. The spare bandwidth is assigned entirely to the currently executing server (in this sense, GRUB-PA is a *"greedy"* algorithm).

5.2.4 Overhead

Since most processors need some time to adjust to a new frequency, it is important to avoid limit situations in which the processor keeps changing its frequency up and down, because this would completely trash the system.

For example, suppose that a task with a very low bandwidth is activated and de-activated very often. If the total utilization is close to one threshold value $\overline{U}_k$, every activation would cause an increase of the frequency, and every de-activation would cause a decrease in the frequency.

To avoid these situations, when the total system bandwidth U goes over one of the thresholds $\overline{U}_k$, we immediately increase the processor frequency because we do not want to risk a hard real-time task to miss its deadline. When a decrease of the total system bandwidth U goes below one of the thresholds $\overline{U}_k$, instead, we do not change the frequency immediately. In fact, in case of a short temporary decrease of the bandwidth, we could end up changing the frequency very often up and down. Therefore, when U goes below a threshold, we set a timer. If the timer expires and U is still below the threshold then we lower the frequency. Otherwise, if U goes above the threshold again, we cancel the timer. This way, we limit the number of frequency switches.

We now explain how it is possible to account for the delay of frequency/voltage switching through a proper tuning of system's parameters. Let δ be the maximum time the processor takes to switch frequency. Suppose that we want a timer expiration interval equal to Δ (i.e., every Δ we want a maximum of two frequency switches, one to go down and another one to go up). In the worst-case, this accounts for a bandwidth reduction of $\frac{2\delta}{\Delta}$. Therefore, we can admit new servers up to a total bandwidth of $1 - \frac{2\delta}{\Delta}$ and set the processor speed to $U + \frac{2\delta}{\Delta}$.

As anticipated in the previous sections, we decided to not consider the energy spent during a frequency switch. In particular, we do not account for this energy overhead in the simulation model presented in the next section. Instead, the presence of this overhead has been automatically accounted for in our experimental results (see Section 5.4.2). In fact, the total energy consumed by our test-bed also comprises the energy due to frequency changes.

5.3 Evaluation of the algorithm

We evaluated our algorithm through a set of comparisons with different energy-aware algorithms proposed in the literature. In particular, we chose to compare our algorithm with the DRA algorithm, proposed by Aydin et al. [17, 18, 16], with the EDF version of the RTDVS algorithms, proposed by Pillai and Shin [98], and with the DVSST algorithm, proposed by Qadi et al. [101]. Refer to Section 2.4 for a detailed description of these

Frequency (MHz)	Voltage (V)	Normalized speed	Normalized power consumption
100 MHz	0.85	25%	11%
200 MHz	1	50%	30%
300 MHz	1.1	75%	54%
400 MHz	1.3	100%	100%

Table 5.1: Operating parameters for the Intel Xscale PXA250 processor.

algorithms.

To compare the algorithms, we used a simulation environment called *RTSim* (which stands for "Real-Time system SIMulator") [96, 4], release 0.3. RTSim is a collection of programming libraries written in C++ for simulating and analyzing real-time control systems. In this tool, a simulation is a C++ program that must be linked to an appropriate library of components which includes schedulers, task models, etc. RTSim started as an academic project, and it has been used primarily for experimenting with new scheduling algorithms and solutions. For this reason, it contains, already implemented, many scheduling algorithms proposed in the literature. The tool is released as Open Source (under the GNU General Public License (GPL)), to give researchers a common simulation platform for comparing the performance of new scheduling algorithms. For our purposes, we extended the processor components of RTSim, to include models of processors with varying speed. Moreover, we implemented the new energy-aware schedulers (namely, DRA, RTDVS and DVSST).

In particular, we modelled the power consumption of both an Intel Xscale PXA250 [58, 57] processor, using four different operating frequencies, and a Transmeta Crusoe TM5800 processor [131], using seven operating frequencies. Tables 5.1 and 5.2 show the operating parameters for these models of processors. From the values contained in the tables it is possible to notice that in both cases the greater is the processor speed, the greater is its energy consumption, following a convex power/speed curve as discussed in Section 2.2.3. An important consequence is that the minimal energy consumption is obtained by setting the processor speed most uniform as possible [63, 98] — i.e., maintaining a constant speed $\overline{\alpha}$ is better than switching between two different speeds across $\overline{\alpha}$.

The power consumption model chosen in the simulations is very simple but effective and it consists of associating a power consumption to each processor frequency according to Equation 2.3. It is the model most used in the literature so far for the evaluation and comparative analysis of energy-aware scheduling algorithms [98, 17, 18, 101].

Frequency (MHz)	Voltage (V)	Normalized speed	Normalized power consumption
300 MHz	0.8	30%	11%
433 MHz	0.875	43%	20%
533 MHz	0.95	53%	28%
667 MHz	1.05	67%	44%
800 MHz	1.15	80%	63%
900 MHz	1.25	90%	83%
1000 MHz	1.3	100%	100%

Table 5.2: Operating parameters for the Transmeta Crusoe TM5800 processor.

5.3.1 Comparison with DRA and RTDVS

We compared our algorithm GRUB-PA with the DRA [17, 18] and the RTDVS [98] algorithms. For the DRA algorithms, we compared GRUB-PA with the basic algorithm, as well as with both its extensions (for the aggressive one, we chose the AGGR1 policy described in [18]). For the RTDVS algorithms, we made the comparison against the EDF versions, since they are more similar to the GRUB-PA algorithm.

To compare the algorithms, we performed two different kinds of simulations. We followed the same methodology of Aydin et al. [18]. For GRUB-PA, we generated a server for each task, with P_i equal to the task period T_i, and U_i equal to the ratio $WCET/P_i$, so that, according to Theorem 4, no task ever misses its deadline.

Let WCET and BCET indicate the worst-case and the best-case execution times, respectively. In the first set of simulations, we fixed a constant WCET/BCET ratio for each task, while the average workload varied from 10 percent to 70 percent. Notice that the value 70 percent corresponds to a WCET equal to the task period. We did not use higher values for the average workload because we were interested in a comparison between the energy consumption of the algorithms, and not between the number of deadline misses in case of overload.

For each value of the workload, we simulated 100 different task sets, each one consisting of 15 different periodic tasks with randomly generated periods. The results are shown in Figures 5.3 and 5.4 for a WCET/BCET ratio equal to two, and in Figures 5.5 and 5.6 for a ratio equal to four. The simulations have been performed for both the Intel Xscale PXA250 (Figures 5.3 and 5.5) and the Transmeta Crusoe TM5800 (Figures 5.4 and 5.6).

The second test measured the amount of power consumption with a constant average workload (50 percent) and a variable WCET/BCET ratio. For each value of the WCET/BCET ratio we ran 100 simulations using different task sets, again each set consisting of 15 tasks.

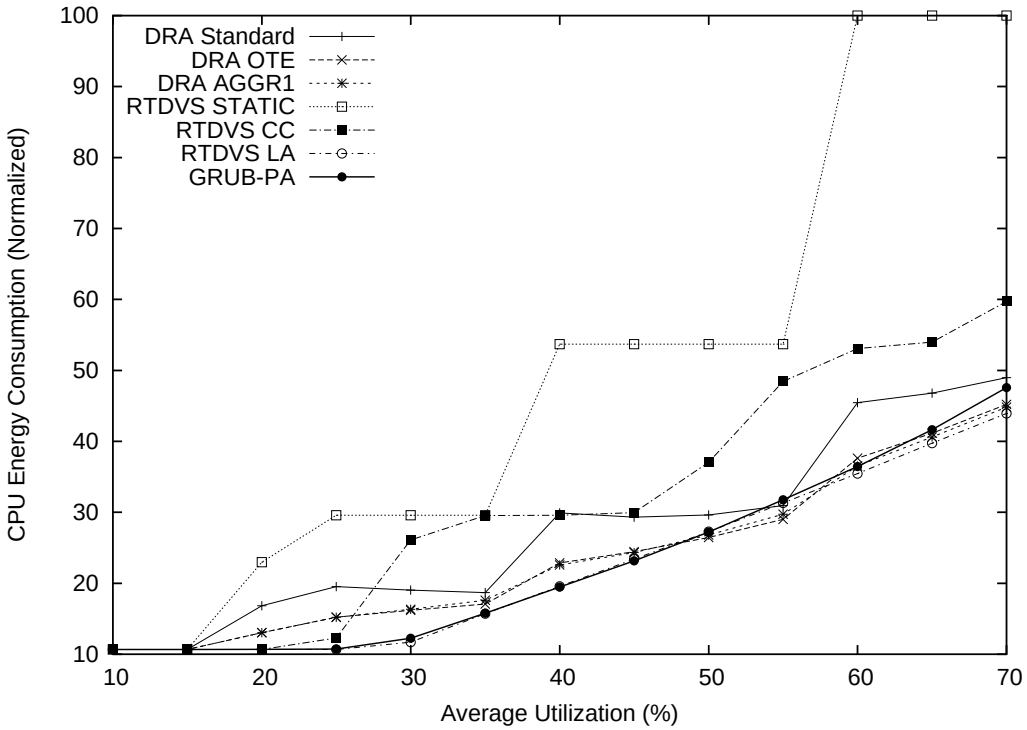

Figure 5.3: Energy consumption with WCET/BCET ratio equal to two on a PXA250.

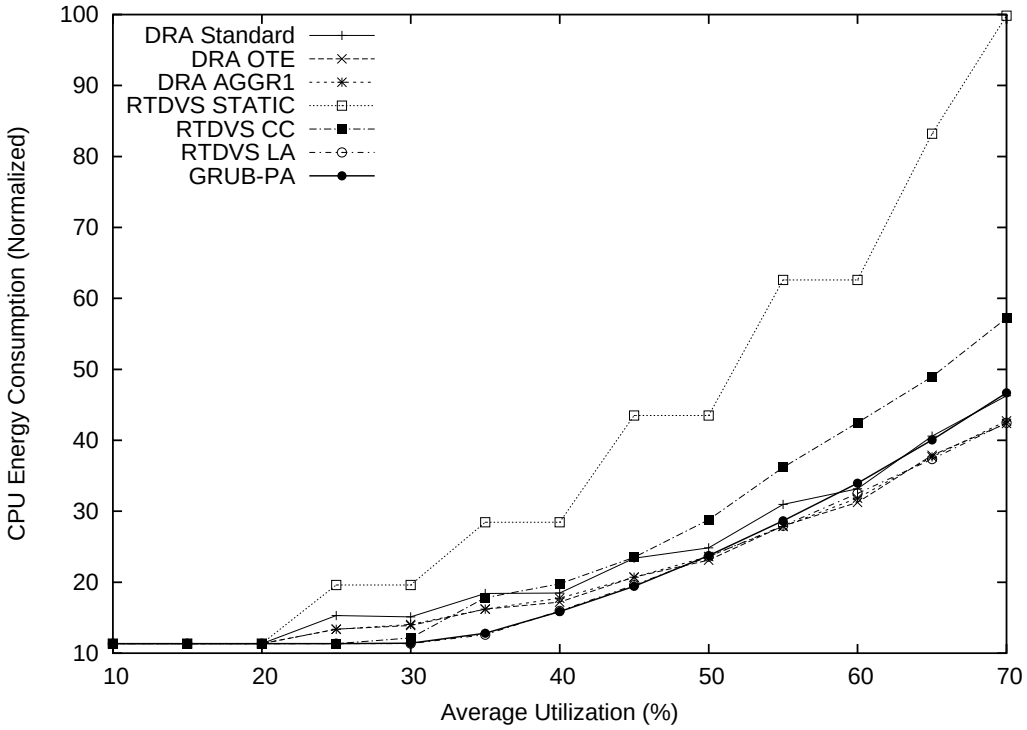

Figure 5.4: Energy consumption with WCET/BCET ratio equal to two on a TM5800.

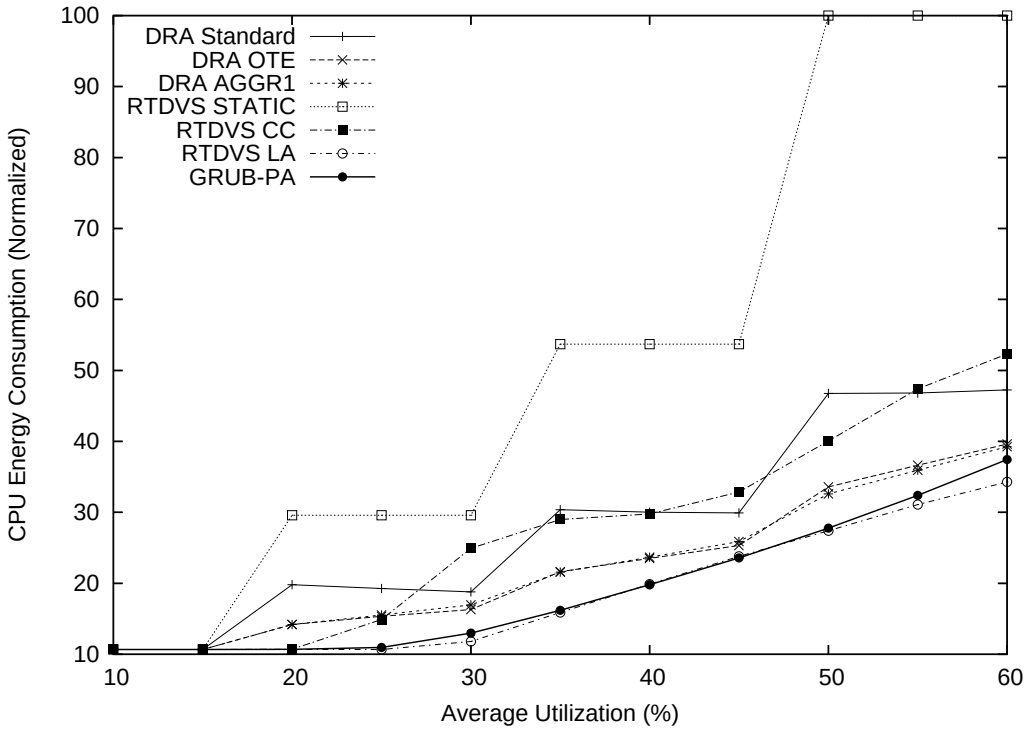

Figure 5.5: Energy consumption with WCET/BCET ratio equal to four on a PXA250.

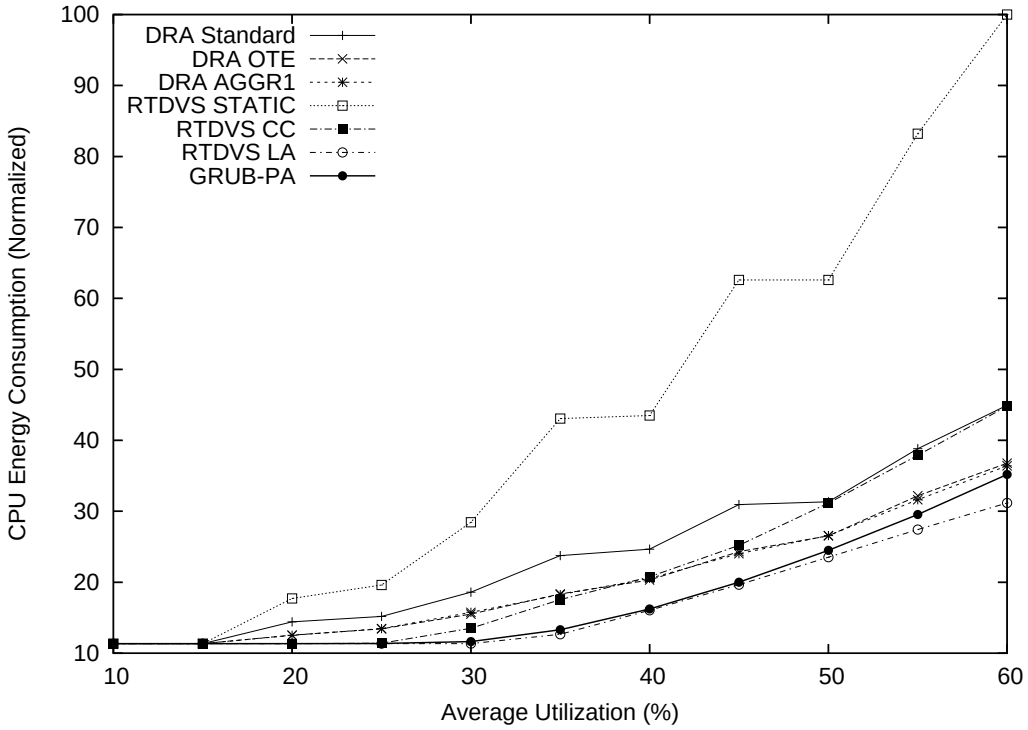

Figure 5.6: Energy consumption with WCET/BCET ratio equal to four on a TM5800.

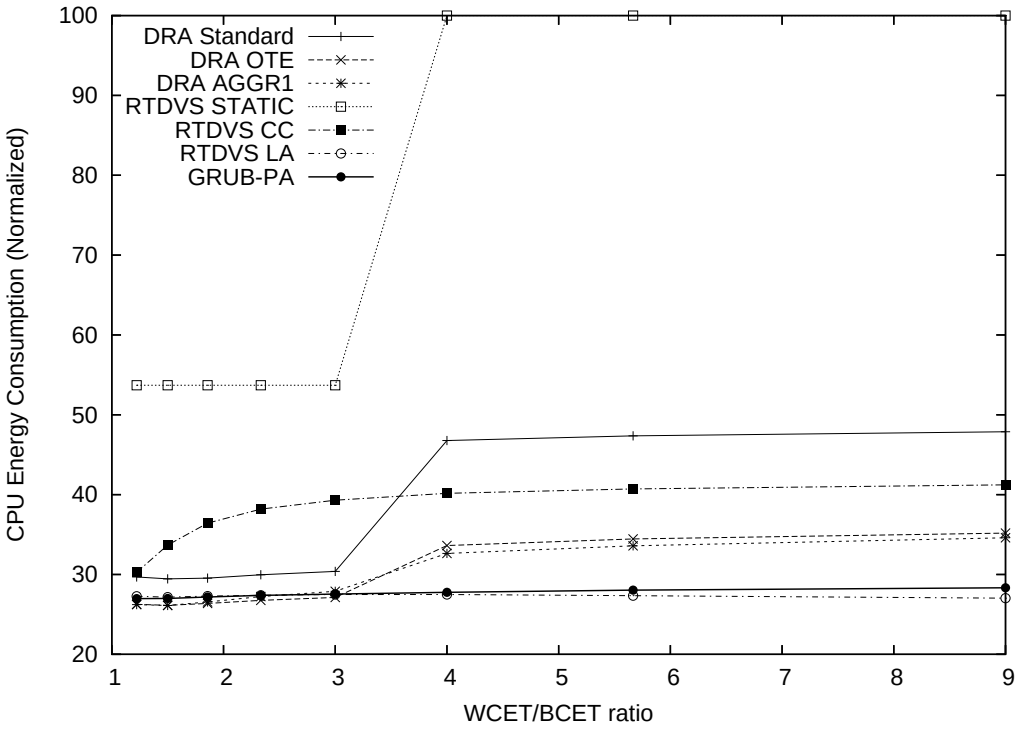

Figure 5.7: Energy consumption with constant average workload on a PXA250.

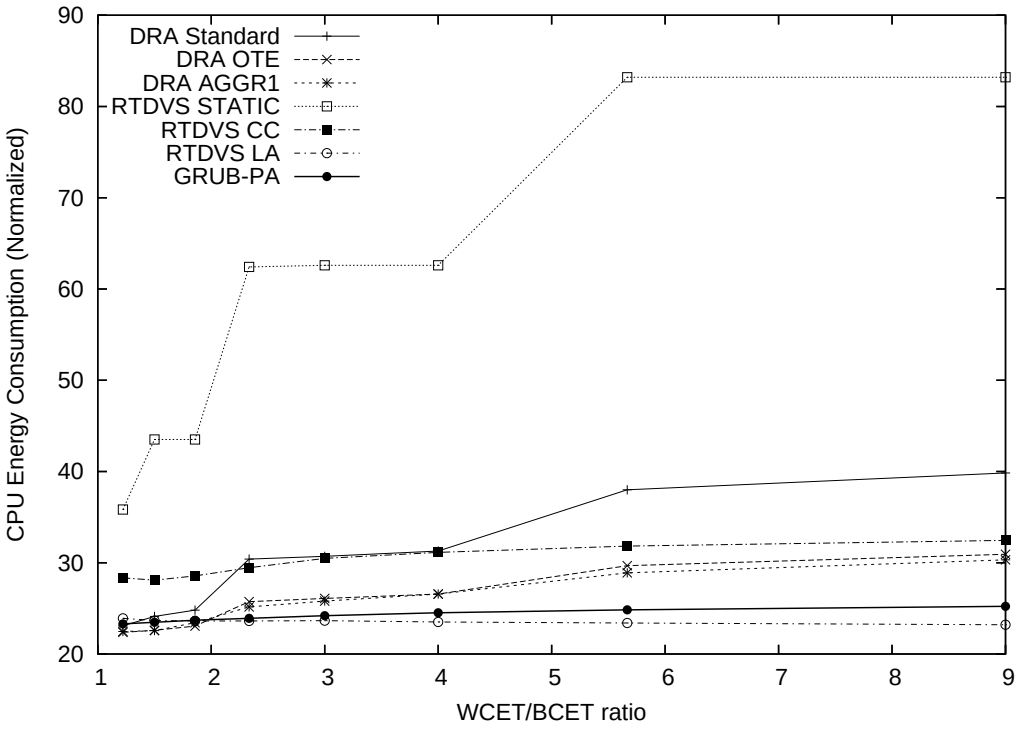

Figure 5.8: Energy consumption with constant average workload on a TM5800.

The confidence intervals obtained during the simulations have not been shown since they were very small (in all simulations, all algorithms presented a 99 percent confidence interval less than one unit of the normalized value of the energy consumption).

From all simulations, it is possible to conclude that GRUB-PA shows performance similar to the other algorithms. In particular, for values of the average utilization less than 50 percent, GRUB-PA shows performance better than most of them. Among all algorithms, RTDVS-Look Ahead proposed by Pillai and Shin [98] presented the lowest average energy consumption. However, GRUB-PA has very similar performance compared to RTDVS-Look Ahead. Moreover, it is important to point out that, unlike DRA and RTDVS, GRUB-PA does not assume a hard real-time periodic task model, and it can be applied to both hard and soft, periodic, sporadic or even aperiodic tasks.

Notice that the worst performance has been shown by RTDVS-Static which, being a static algorithm, is not able of dynamically reclaim the slack time. Moreover, notice that worse algorithms are more sensitive to the discretization of processor frequencies because their dynamic reclamation is worse than dynamic reclamation of the better algorithms.

5.3.2 Comparison with DVSST

We also compared GRUB-PA against the DVSST algorithm proposed by Qadi et al. [101], since it is an algorithm that assumes a sporadic task model. Refer to Section 2.4.6 for a description of the algorithm.

In each simulation run, we generated eight sporadic tasks with minimum interarrival times T_i randomly chosen between 1,000 and 10,000 and with actual interarrival time uniformly distributed between T_i and $T_i * 1.1$. Each task has a variable computation time, with a 20 percent of variation over the central value. In each experiment, the sum U_{max} of the maximum bandwidth requested by all tasks is constant. Finally, U_{max} is varied between 10 percent and 90 percent. The results for the PXA250 and the TM5800 processors are shown in Figures 5.9 and 5.10, respectively.

As it is possible to see, the DVSST algorithm is much more sensitive to the discretization of processor frequencies with respect to GRUB-PA, due to the lack of reclamation of early tasks' completions. The irregularity of the pattern for DVSST decreases as the number of available discrete frequencies increases, as it can be noticed by comparing Figure 5.9 with Figure 5.10. As expected, GRUB-PA presents an improvement up to 40 percent with respect to DVSST.

5.4 Implementation and experimental results

We implemented GRUB-PA in the Linux operating system in the context of the OCERA (*"Open Components for Embedded Real-time Applications"*) project, funded by the Euro-

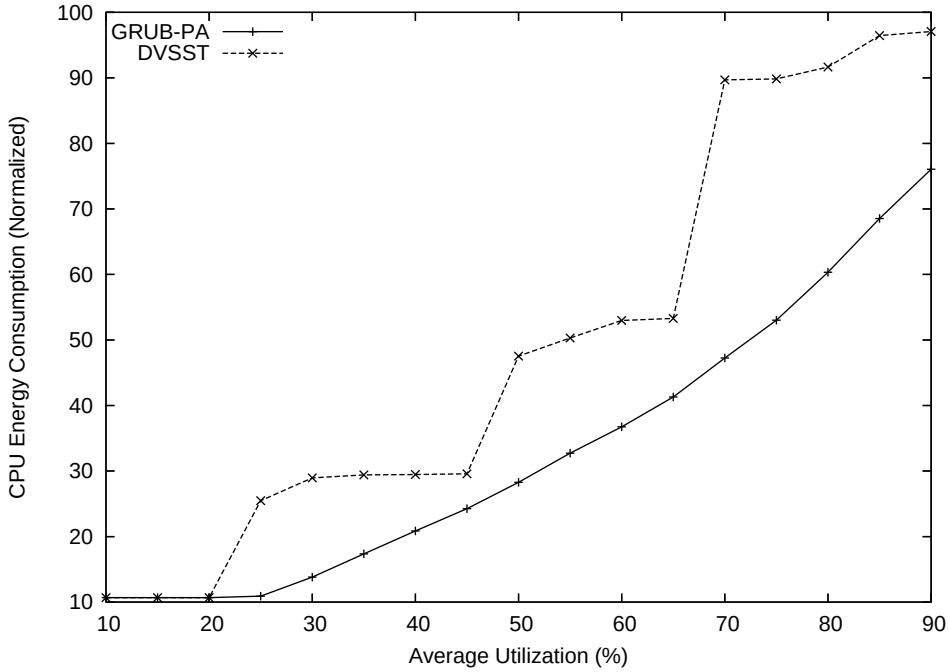

Figure 5.9: Energy consumption with variable average workload on a PXA250.

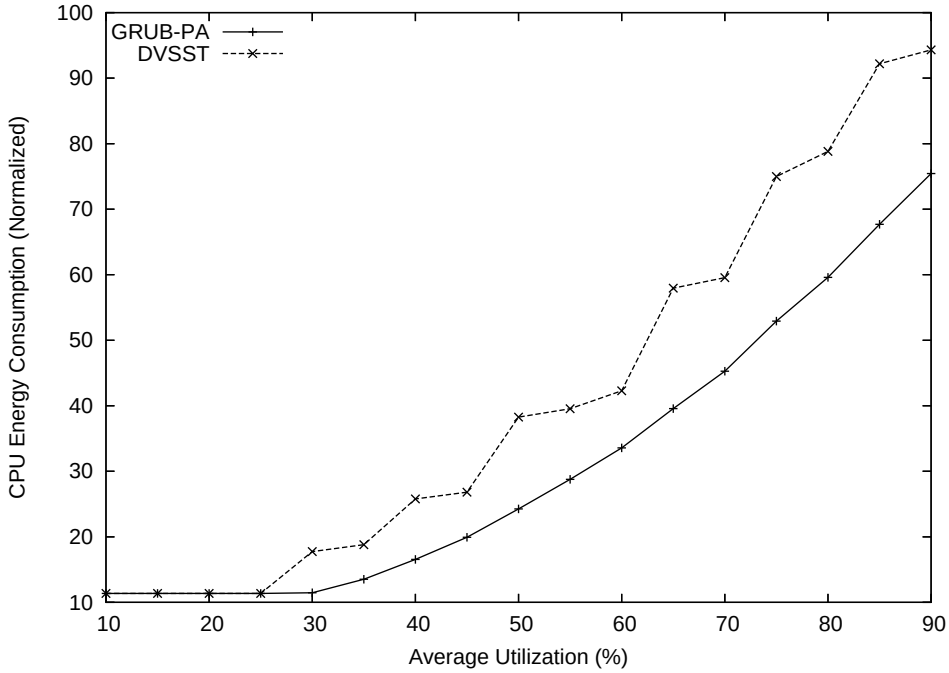

Figure 5.10: Energy consumption with variable average workload on a TM5800.

pean Commission in the Fifth Framework programme (IST-35102). Our implementation is available as Open Source code [9, 106, 111, 112, 39] . The main objective of this project was the design and implementation of a library of free software components for embedded real-time systems. These components have been used to create flexible (supporting a wide variety of applications), configurable (scalable from a small to a fully featured system), robust (fault-tolerant and with high performance) and portable (adaptable to several hardware and software configurations) systems.

We modified the scheduling policy of Linux 2.4.18. Since we wanted to limit as much as possible the modifications to the standard Linux scheduler, the real-time scheduler has been developed as a loadable kernel module [38]. A small patch (called *"Generic Scheduler Patch"*) applied to the Linux kernel exports the necessary symbols and the relevant events to the real-time scheduler. The interface to the scheduler has been exported through the standard `sched_setscheduler()` system call, adding a new scheduling policy, and extending the structure `sched_param`.

The real-time scheduler needs to know all the relevant events regarding the processes in the system (i.e., process creation, termination, blocking and unblocking). For this reason, the patch exports some *hooks* that are used to intercept the interesting scheduling events:

- *block_hook* is invoked when a task is blocked, so that the scheduler understands that the current job has finished execution.

- *unblock_hook* is invoked when a task is unblocked, so that the scheduler is informed of the arrival of a new job.

- *fork_hook* is invoked when a new task is created through a `fork()` and a pointer to the task is passed as parameter.

- *cleanup_hook* is invoked when a task is terminated, so that the scheduler can free the internal resources concerning the task.

- *setsched_hook* is invoked when the system calls `sched_setscheduler()` or `sched_setparam()` are called by the user.

All the hooks, except `setsched_hook`, have a parameter that is a pointer to the structure `task_struct` of the corresponding task.

The patch inserts a new field called `private_data` in the `task_struct` data structure, of type `void*`. It is a pointer used by our scheduler to access the private real-time data of every task — in our case, a pointer to the server handling the task. If necessary, the scheduler must set this field to the appropriate data structure during the `fork_hook`. When the module is removed, it must ensure that all tasks have their `private_data` set to `NULL`.

Figure 5.11: Intrinsyc Cerfcube.

Our dynamically loadable scheduler modifies the task priority, raising the selected task to the maximum priority, and then calls the standard Linux scheduler. Based on the information received by the hooks, our scheduler selects which task has to be executed and sets its policy to `SCHED_FIFO` or `SCHED_RR` and the `rt_priority` to the maximum real-time priority + 1. Then, it invokes the Linux scheduler. In practice, the Linux scheduler acts as a dispatcher for our external real-time scheduler. Thus, the modifications to the standard Linux scheduler are minimal.

Notice that, in this implementation, the scheduling algorithm does not assume any periodic behaviour of the task. As a matter of fact, the scheduler only intercepts the blocking/unblocking events of a task, and it is task's responsibility to implement a periodic behaviour, if required. Thus, our scheduler is able to serve any kind of task, from non-periodic legacy Linux processes to periodic soft real-time tasks. An in-depth description of the implementation can be found in [111].

5.4.1 Test-bed

We tested GRUB-PA on a Intrinsyc CerfCube 250 architecture (see Figure 5.11). It consists of 32 MB Flash ROM, 64 MB SDRAM, and a Ethernet 10/100 Mbps. The processor is the Intel PXA250 [58, 57] described in Figure 5.12. It is a superpipelined 32 bits RISC processor based on the Intel *Xscale* micro-architecture. This processor permits a on-the-fly switch of the clock frequency and a sophisticated power consumption management.

In particular, the processor can be in one of the following states:

1. Turbo Mode: the processor core works at the peak frequency.

2. Run Mode: the processor core works at its "normal" frequency. In this mode, it is assumed that the processor frequently accesses external memory, so it is convenient for it to work at a frequency lower than the Turbo Mode frequency.

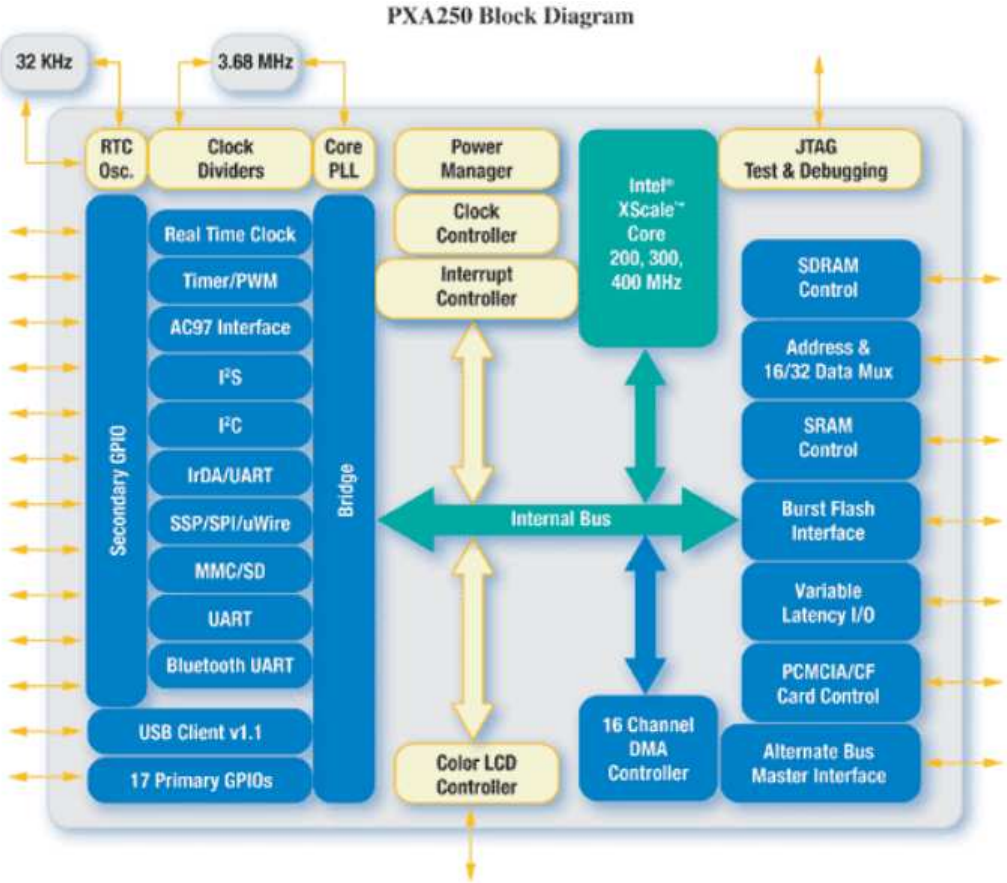

Figure 5.12: Intel PXA250 block diagram.

The register in which it is possible to select the clock frequency is called CCCR (*Core Clock Configuration Register*) and can be found at the address `0x41300000`. The CCCR register manages the core clock frequency which the memory controller clock and the DMA clock depend upon. In this register, the following parameters are specified:

- Frequency multiplier from the quartz frequency to the memory controller (L)
- Frequency multiplier from the memory frequency to the processor frequency in Run Mode (M)
- Frequency multiplier from the processor frequency in Run Mode to the processor frequency in Turbo Mode (N)

The value of L is chosen depending on the constraints of the external memory and it is usually constant, while the values of M and N can change in order to change the speed of the processor. The value M is chosen based on the bus speed constraints and on the minimal performance requirements. The value N is based on the values of peak performance.

To modify the system clock frequency, register CCLKCFG can be used, that is register number 6 of the co-processor 14 (which is dedicated to power management at the lower level). It is a 32 bit register that is used to enter the Turbo Mode and the Frequency Change Sequence. Register CCLKCFG can only be modified through the following assembler instructions:

```
MCR p14, 0, R0, c6, c0, 0
```

to read the value of the register and put it in `R0`, and

```
MRC p14, 0, R0, c6, c0, 0
```

to copy the content of register `R0` into the register CCLKCFG.

As the reader can see, there is a great deal of flexibility in setting the clock frequencies. We had to choose how to implement our algorithm, which frequency to use as base frequencies, and so on. Notice that the PXA250 processor does not allow many values for the core frequency without changing the memory speed as well. However, we wanted to maintain the memory frequency constant because different values could affect our experimental results. Thus, we configured the system to use only three different processor frequencies — i.e., 100 MHz, 200 MHz, 400 MHz — whereas the memory frequency did never change. By using these 3 levels, we were able to use the minimum possible frequency (100 MHz) and the maximum one (400 MHz). Therefore, we had two thresholds, $U_1 = 1/4$ and $U_2 = 1/2$.

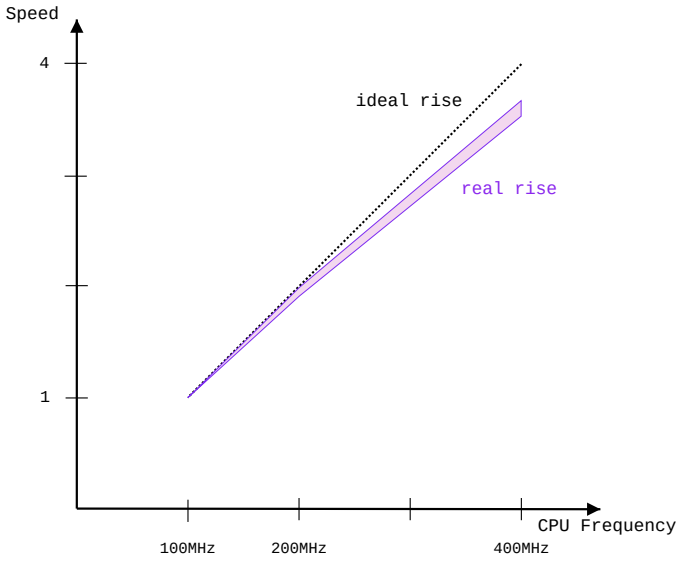

Figure 5.13: Decompression speed related to processor speed.

5.4.2 Experimental results

Our study was particularly focused on multimedia applications. Therefore, we decided to evaluate the performance of our system using a multimedia application. However, our approach can be used for a large range of different applications, because it is completely transparent to the application characteristics. Unfortunately, our test-bed system, the Intrinsyc CerfCube, does not present a video output. So, we decided to focus our attention on a audio decoder. We selected the decoder provided by the *Xiph.Org Foundation*[2]. The audio format was *Ogg Vorbis*, a non-proprietary high quality (from 8 to 48 bits, polyphonic) compressed audio format with fixed or variable bitrate ranging from 16 to 128 Kbps per channel. It is in direct competition with the MPEG format et similia.

One may argue that varying the processor frequency only, without touching the peripherals frequencies (like memory, for example) does not bring appreciable advantages. We performed some experiments showing that, in the considered test-bed, this is not the case. To execute the first test, we decompressed a set of audio streams at 44100 Hz and two channels, measuring the time necessary to decompress every stream under different fixed clock frequencies. From the measured values, we extracted how much the speed of decompression is related to the speed of the processor. The result is shown in Figure 5.13, where we show on the x-axis the frequency of the processor, and on the y-axis the decompression speed. As the reader can see, the relationship is almost linear. This justifies our assumption that by doubling the processor frequency, the computation time of one task's job halves.

[2]Xiph.Org is a "non-profit organization dedicated to protecting the foundations of Internet multimedia from control by private interests".

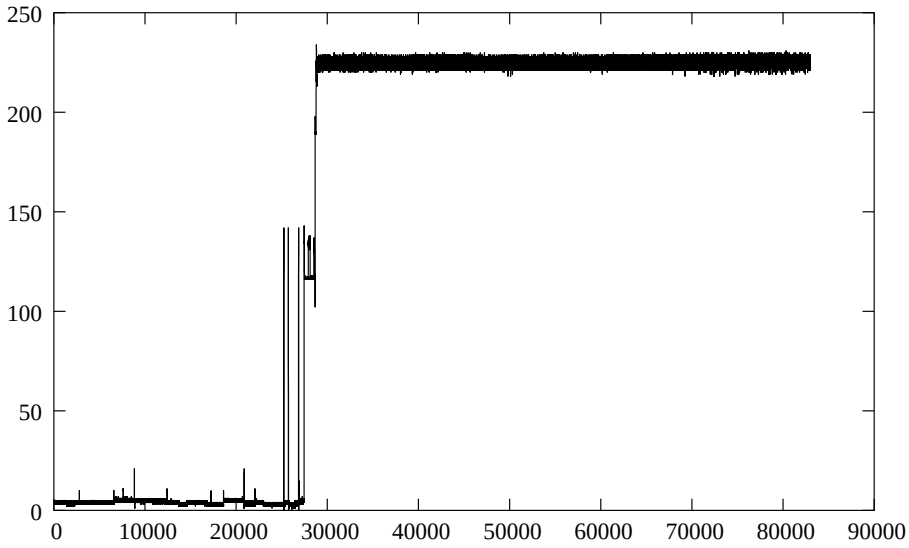

Figure 5.14: Current entering in the CerfCube system during the boot.

Processor frequency	Current
100 MHz	446.0 mA
200 MHz	508.5 mA
400 MHz	579.9 mA

Table 5.3: Average values of the input current.

Then, we evaluated the power consumed by our system under different conditions, with and without GRUB-PA. We inserted a dedicated electronic circuit between the CerfCube board and the power supply to measure the input current to the board. The circuit is powered by a separate 9V battery: it puts a very small resistor in series with the CerfCube board and measures the voltage at the ends of the resistor. The resulting data are sampled and sent through a serial cable to a host PC that collects and shows them. Figure 5.14, for instance, shows the input current measured during the boot of the embedded device.

We did many experiments using the multimedia application described above, in order to ensure that the scheduler was working properly and that the assumptions of the GRUB-PA algorithm were satisfied. In particular, we measured the temporal evolution of the current entering into the board in different conditions, with or without our algorithm, and under different workloads. The average values of the input current that we observed are reported in Table 5.3.

We also considered the most general case of many real-time applications activating and deactivating dynamically. We executed several instances of the audio decoder concurrently, each one on a different audio stream. A different value of the bandwidth was assigned to each instance of the application, so that the total load of the system was highly variable and we could take fully advantage of the DVS technique. We observed that, at any time,

the power consumption was proportional to the value of the U variable of GRUB-PA (respecting the discretization of the processor frequencies, of course). In particular, for a value of U comprised between 0 and 0.25, we measured an amount of energy saved equal to *38.4* percent of the total power consumed by the system. Notice that this is the percentage of energy saved with respect to the *total* energy consumed by the board, although our algorithm acts only on the processor. This value proves that the DVS technique is as an effective way of reducing energy consumption in modern computational systems: using GRUB-PA, in fact, the energy consumption of the whole system has been easily reduced of one third.

The interested reader can refer to the original paper [111] for a more detailed description of the experiments.

5.5 Summary

In this chapter, we presented GRUB-PA, a novel energy-aware scheduling algorithm suitable for systems consisting of hard periodic and soft aperiodic real-time tasks. The algorithm is based on the Resource Reservation framework, so it does not make any restrictive assumption on the characteristics of the tasks.

We theoretically analyzed the main properties of the algorithm, simulated its behaviour and set up a test bed.

Our simulations show that GRUB-PA, besides giving guarantees about the temporal execution of tasks (i.e., temporal isolation), presents performance similar to those provided by other energy-aware scheduling algorithms presented in the literature so far. In particular, for values of the average utilization less than 50 percent, GRUB-PA shows performance better than most of the algorithms presented. However, unlike most algorithms, GRUB-PA can also be applied to hard and soft, and periodic, sporadic or even aperiodic tasks.

Moreover, we presented an implementation of GRUB-PA in the Linux operating system. The experimental results on this real test-bed system show that using GRUB-PA we can save up to 38.4 percent of the total power consumed by the system with respect to the unmodified one.

Chapter 6

Final Remarks

6.1 Conclusions

In this thesis, we have proved that achieving real-time guarantees and energy efficiency at the same time is actually possible by using proper energy-aware real-time scheduling algorithms. Starting from the definitions and the notation concerning real-time systems provided in Chapter 1, we have formally defined a scheduling model to study the problem of energy minimization in real-time system. We have also provided a possible taxonomy of energy-aware scheduling algorithms similar to the one proposed by Kim et al. [65] as well as the state of the art of such algorithms in the real-time literature so far.

In Chapter 3, we studied the problem of energy minimization from an analytical point of view, integrating the concept of probabilistic execution time within the framework of energy minimization. In particular, we found the optimal values of the instant of frequency transition (i.e., *transition point*) and speed assignments when probabilistic information about task execution times is known. The optimal values have been found using a very general model for the processor that accounts for idle power and for both the time and the energy overheads due to voltage/frequency transition. Our result is optimal and represents an improvement over the sub-optimal scheme proposed by Zhu. et al. [43, 141] in the DVS-EDF algorithm (i.e, providing the average number of cycles in the first portion and running the second portion at the maximum processor speed). We also studied how the overhead and the idle power affect the optimal values. Very interestingly, we showed that, in our model, the power consumed during the idle operating mode can be easily taken into account by adjusting the active operating modes of the processor. Then, we showed how our results can be applied to some significant cases (namely, uniform and exponential densities).

In Chapter 4, we provided an overview of the Resource Reservation technique [104] which provides the *temporal isolation* property: the temporal behaviour of each task (i.e., its ability to meet its deadlines) is isolated from the rest of the system and it is not affected

by the behaviour of the other tasks. If a task misbehaves and requires a large execution time, it cannot monopolize the processor. Such property is particularly useful when mixing hard and soft real-time tasks on the same system. Then, we investigated some anomalies in the schedule generated by the CBS [6] and GRUB [76, 75, 77] algorithms and we proposed a novel algorithm, called HGRUB [11], which maintains the same features of CBS and GRUB but it is not affected by the problems described. We also proved some properties about this novel algorithm when scheduling *cpu-intensive* tasks.

In Chapter 5, we modified GRUB for energy-aware scheduling. The novel algorithm has been called GRUB-PA (*Greedy Reclamation of Unused Bandwidth—Power Aware*). Like GRUB, GRUB-PA is based on the Resource Reservation framework, so it does not make any restrictive assumption on the characteristics of the tasks. Thus, unlike most of the algorithms proposed in the literature, GRUB-PA can handle any mixture of hard, soft, periodic, sporadic and aperiodic tasks. The mechanism used by the GRUB-PA algorithm (i.e., *Utilization Updating*) is very similar to the one used by the DVSST algorithm [101]: they both use a global variable U to set the processor speed. However, we have shown that, thanks to its reclamation property, GRUB-PA always anticipates the time at which the processor speed is lowered with respect to DVSST, resulting in a larger amount of energy saved.

We theoretically analyzed the main properties of GRUB-PA, simulated its behaviour and set up a test bed. We evaluated the algorithm through a set of comparisons with several energy-aware algorithms proposed in the literature. In particular, we compared our algorithm with the DRA algorithm, proposed by Aydin et al. [17, 18, 16], with the EDF version of the RTDVS algorithms, proposed by Pillai and Shin [98], and with the DVSST algorithm, proposed by Qadi et al. [101]. To make the comparisons, we implemented the algorithms in our Open-Source scheduling simulation environment *RTSim* [96, 4], release 0.3. For our purposes, we extended the processor components of RTSim, to include models of processors with varying speed, and we modelled the power consumption of both the Intel Xscale PXA250 [58, 57] and the Transmeta Crusoe TM5800 [131] processors. Through a set of simulations, we showed that our algorithm, besides giving guarantees about the temporal execution of tasks, achieves performance similar or better than those provided by the other energy-aware scheduling algorithms, but it has the net advantage of handling different kinds of tasks (i.e., hard, soft, periodic, sporadic and aperiodic tasks).

Finally, we implemented the GRUB-PA algorithm on the Linux kernel 2.4.18 and we tested its effectiveness on a real test-bed by experimentally decoding some compressed audio files. We have built a dedicated electronic circuit to measure the input current entering into our test-bed. We measured the temporal evolution of the current in different conditions, with or without our algorithm, and under different workloads. We observed a maximum energy reduction equal to *38.4* percent of the total energy consumed by the

system. Notice that this is the percentage of energy saved with respect to the *total* energy consumed by the board, although our algorithm acts only on the processor.

In Appendix A, we also provide an overview of the existing real-time extensions of the Linux kernel, showing that creating a RTOS starting from a general-purpose operating system is actually possible. We explain the typical problems in supporting real-time activities on such kind of systems. Then, we describe the possible approaches to solve these problems as well as the advantages and disadvantages of each technique.

6.2 Ongoing work

Currently, we are working on several topics related to Resource Reservations and energy efficiency. These topics can be summarized as follows.

Processor model and energy management scheme

We are working on several extensions of the model and results discussed in Chapter 3. In particular, we are addressing

- the extension to the case of discrete processor frequencies, developing an on-line algorithm with low complexity to find the best pair of frequencies to be used;
- the extension to multiprocessor and multicore systems, considering the case in which the speed of each computing unit can be changed independently from the others;
- the extension to the case of multiple speeds (i.e., more than two) resembling the approach of PACE algorithms; this extension would be an improvement over the classical algorithms because it would consider the more general case with idle power and with both time and energy overheads due to voltage/frequency transitions. We are also studying the optimal number of speeds for such algorithm.

Resource Reservations

We are currently investigating the use of the Resource Reservation technique in the following areas.

- We are working on a Resource Reservation mechanism for webservers. This allows to provide QoS to each served request as well as temporal isolation among different requests. In particular, we are implementing a Resource Reservation mechanism on top of the *Apache* [1] webserver.
- We are investigating the feasibility of an energy-aware extension of the IRIS [83] algorithm, similarly as what we have done for GRUB. A comparison between GRUB-PA and this novel algorithm would be very interesting.

- Extending GRUB-PA to multiprocessor and multicore systems. In particular, starting from the results obtained in a preliminary study [107], we are evaluating the possibility of implementing GRUB-PA in the *Condor* [2] workload management system to create a flexible energy-aware webserver.

Operating systems

In the field of operating systems, we plan to work in the following sectors.

- We are evaluating the possibility of using the virtualization technique provided by modern operating systems (e.g., *Xen*) and task migration to reduce the energy consumed by a cluster of computers. In fact, in case of low workload, the tasks running on a cluster unit could be migrated to a different unit in order to shut down one computation unit and reduce energy consumption.

- An implementation of the Resource Reservation framework on the current mainstream of the Linux kernel, similarly as what we have done for the 2.4.18 kernel in the OCERA project. This implementation should exploit the newest services offered made available by the work of Ingo Molnar [89].

- Exploit the *Dynamic Tick* feature of the Linux kernel, by implementing an algorithm that sets the value of the system tick based on the current workload and the timing constraints of the real-time tasks currently in execution.

Appendix A

Supporting Real-Time Applications on Linux

In the last years, there has been a considerable interest in using Linux as a RTOS, especially in control systems. The simple and elegant design of Linux guarantees robustness and very good performance, while its Open Source license allows to modify and change the source code according to the user needs.

However, Linux has been designed to be a general-purpose operating system where applications can dynamically activate at any time, and the scheduler cannot make any restrictive assumption on the characteristics of the running programs. Therefore, it presents some issues, like unpredictable latencies, limited support for real-time scheduling, and coarse grain timing resolution that might be a problem for real-time applications [40].

For these reasons, several modifications have been proposed to add "real-time" features to the kernel. In this appendix, we give a brief description of the many existing approaches to support real-time applications on Linux. Moreover, we take a look at the expected trends, presenting what we believe will be the future of Linux for what concerns real-time support.

A.1 Introduction

Linux is a full-featured operating system, originally designed to be used in server or desktop environments. Since then, Linux has evolved and grown to be used in almost all computer areas — among others, embedded systems and parallel clusters [1].

In the last years, there has been a considerable interest in using Linux for real-time control systems, from both academic institutions, independent developers and industries.

[1] Linux currently supports almost every hardware processor, including x86, AMD x86-64, ARM, Compaq Alpha, CRIS, DEC VAX, H8/300, Hitachi SuperH, HP PA-RISC, IBM S/390, Intel Xscale IXP and PXA, Intel IA-64, MIPS, Motorola 68000, PowerPC, Samsung S3C24XX, SPARC and UltraSPARC.

There are several reasons for this raising interest. First of all, Linux is an Open Source project, meaning that the source code of the operating system is freely available to everybody, and can be customized according to the user needs, provided that the modified version is still licensed under the GNU General Public License (GPL). This license allows anybody to redistribute, and even sell, a product as long as the recipient is able to exercise the same rights (access to the source code included). This way, a user (for example, a company) is not tied to the OS provider anymore, and is free of modifying the OS at will. The GPL Open Source license helped the growth of a large community of researchers and developers who added new features to the kernel and ported Linux to new architectures. Nowadays, there is a huge amount of programs, libraries and tools available as Open Source code that can be used to build a customized version of the Linux OS.

Another reason for the usage of Linux in real-time systems is its wide popularity and success. It has the simple and elegant design of the UNIX OSs, which guarantees a very stable, robust and secure system. Moreover, it has excellent performance and a good network protocol stack implementation. The portability of code from different UNIX operating systems is ensured by the well known *"Portable Operating System Interface"* (POSIX) API. This is an IEEE standard defining the basic environment and set of functions offered by the operating system to the application programs. Finally, the huge community of engineers and developers working on Linux makes finding expert kernel programmers very easy.

Thus, when compared to commercial real-time operating systems (RTOSs) in terms of cost of development, Linux has good chances to be the winner. Unfortunately, the standard mainline kernel (as provided by Linus Torvalds) is not adequate to be used as RTOS. Linux has been designed to be a general-purpose operating system (GPOS), and thus not much attention has been dedicated to the problem of reducing the latency of critical operations. Instead, the main design goal of the Linux kernel has been (and still remains) to optimize the average throughput (i.e., the amount of "useful work" done by the system in the unit of time). As we will show in Section A.3, a Linux program may suffer high latencies in response to critical events.

To overcome these problems, many approaches have been proposed in the last years to modify Linux in order to make it more "real-time". Kernel developers have worked in parallel toward the goal of reducing the worst-case latency of the standard Linux kernel, and proposed some possible solutions. At the same time, a new approach (called *Resource Reservation*) is slowly making its way to real-time system programming, and many Linux-based implementations of this approach are already available. Because of these improvements, the final goal of supporting real-time activities on a general purpose operating system [40] like Linux is finally becoming possible.

In this appendix, we discuss the state of the art of the different approaches to a Linux-

based RTOS, and we take a look at the future trends. The appendix is organized as follows. In Section A.2, we describe the problems in supporting real-time activities using the standard Linux kernel. Then, we present the possible approaches to create a real-time version of Linux. In particular, in Section A.3 we describe the approach called *Interrupt Abstraction*, and we present some implementations of this mechanism. In Section A.4, instead, we present some different techniques to add real-time capabilities to the standard Linux kernel. Finally, in Section A.5 we state our conclusions and we try to predict the next steps of Linux kernel development for what concerns real-time support.

A.2 Towards a Real-Time Linux Kernel

A.2.1 Problems Using Standard Linux

There are several issues that must be analyzed in supporting real-time activities in a general-purpose operating system like Linux. All these issues are related to non-deterministic behaviours of the system, which makes real-time processes experience latencies of unpredictable length during execution [8, 48]. All real-time applications, however, consist of time-sensitive activities having strict timing constraints (like deadlines) that must be satisfied, otherwise the system does not work properly.

The "latency" of an OS can be defined in many different ways. In general, latency is the time it takes between the occurrence of an event and the beginning of the action that will respond to the event. In the case of real-time control applications, it is often defined as the time between the interrupt signal arrives to the processor (signaling that an external event like a sensor reading has occurred) and when the handling routine actually starts execution (for example the real-time task that will respond to the event). In the development of critical real-time control systems, it is necessary to account for the worst-case scenario: hence we are particularly interested in the maximum latency values.

In the 2.4.x versions of Linux, the maximum latency can be very high: for example, it can go up to 230 msec on a native standard kernel running on a Desktop computer [135]. Such a large interval of time is considered inadequate even for soft real-time applications. A control application requiring a sampling rate of 10 Hz (and hence a sampling period of 100 msec) cannot be safely executed in real-time on Linux 2.4.17, as in the worst-case up to two invocations can be delayed or even skipped.

The two main sources of latency in general-purpose operating systems are *task latency* and *timer resolution*. Task latency is experienced by a process when it cannot preempt a lower priority process because this is executing in kernel context (i.e., the kernel is executing on behalf of the process). Typically, monolithic operating systems do not allow more than one stream of execution in kernel context, so that the high priority task cannot execute until the kernel code either returns to user-space or explicitly blocks. This is

equivalent to having a *lock* for all the kernel code: whenever a task invokes a kernel routine, the kernel is *locked*, and no other kernel activity (except very low-level interrupt handling, commonly referred as *top-half* in Linux) can be executed. In case of Linux 2.4.x, many portions of the kernel code require a considerable execution time. If a high priority task is activated in response to an interrupt while the kernel is locked, it must wait for the kernel lock to be released before starting execution.

Another source of latency is related to timing resolution. Every operating system needs to keep track of the flow of time, because a large number of kernel functions (e.g., process scheduling) are time-driven. Operating systems keep track of time through an electronic timer circuit that issues a hardware interrupt after a pre-programmed amount of time. When the timer issues the interrupt, the kernel knows that the specified interval of time is elapsed. Typically, general-purpose operating systems like Linux set the system timer in order to have periodic interrupts at a certain frequency. The value of the period is called *tick* and it is often a configurable option which depends on the processor speed. For example, in Linux 2.6 the tick value can vary between 1 msec (on fast processors) up to 40 msec (on slow machines). The periodic tick rate directly affects the granularity of all timing activities and it is one of the major causes of latency in operating systems. The kernel, in fact, is not able of measuring (or deferring activities for) intervals of time below a certain threshold. This represents a problem in real-time systems which need an accurate estimation of the current time and the execution of tasks at precise instants.

Finally, another problem in supporting real-time processes in general-purpose operating systems is the limited support for proper real-time scheduling policies. Linux provides the POSIX-compliant SCHED_FIFO and SCHED_RR policies, that are simple fixed priority schedulers. Both policies enable a task to execute until it explicitly releases the processor. Although fixed priority is adequate for real-time scheduling in embedded systems, it is not suitable for supporting real-time activities in general-purpose operating systems. Notable drawbacks of fixed priority schedulers are the fairness and the security among processes [9]. In fact, if a regular non-privileged user is enabled to access the real-time scheduling facilities, then he can rise his processes to the highest priority, starving the rest of the system. On the other hand, it is very difficult to provide real-time guarantees if only privileged users are allowed to access the scheduling facilities. Moreover, even trusted users may crash the system due to some mistake during development or debugging.

A.2.2 Classification of Linux-based RTOSs

For the above reasons, the standard Linux kernel is not suitable for supporting real-time control applications. Thus, during the last years, several approaches have been proposed to add real-time features to the kernel. These techniques can be grouped in the following two classes of approaches:

1. *Interrupt Abstraction*, which adds a new abstraction layer beneath the kernel to take full control of interrupts and system timers. This approach creates a hard RTOS that executes Linux as a background task. We describe this approach in Section A.3;

2. *Kernel Preemption* approaches, that make the behaviour of the system *more deterministic*, by improving kernel preemption, response times and timing resolution. These techniques are described in Section A.4.

A.3 Interrupt Abstraction

The approach based on Interrupt Abstraction consists of creating a layer of virtual hardware between the standard Linux kernel and the computer hardware, as shown in Figure A.1. This layer is also called *Real-Time Hardware Abstraction Layer* (RTHAL) [42], although it only virtualizes interrupts. Then, a separate complete *real-time subsystem* that consists of a RTOS and a set of real-time tasks and device drivers, runs together with the Linux OS.

The mechanism is the following. Every interrupt source is marked as real-time or non real-time. Real-Time interrupts are served by the real-time subsystem, whereas non-real-time interrupts are managed by the Linux kernel. To avoid latencies when executing real-time code, every time an interrupt arrives (the arrow marked with **(a)** in Figure A.1), the RTHAL checks if it is a real-time interrupt. If so, the interrupt is immediately served by the real-time subsystem. A non-real-time interrupt, instead, is not forwarded to the Linux kernel immediately, but it is stored in a "pending interrupts" vector. The pending interrupts (and all other Linux activities) can be served only when no other real-time activity is running (arrow marked with **(b)** in Figure A.1). In practice, the resulting system is a multithreaded RTOS, in which the *standard Linux kernel is the lowest-priority thread.* The Linux kernel, and all the normal Linux processes are managed by the abstraction layer as the lowest priority task — i.e., the Linux kernel only executes when there are no real-time tasks to run and the real-time kernel is inactive.

Three main modifications must be done to the Linux kernel in order to virtualize the hardware and take full control of the machine. The abstraction layer must:

1. take direct control of all the hardware interrupts. The new interrupt handler intercepts all hardware interrupts, and checks whether the interrupt is related to a real-time activity or not, according to the mechanism described in Figure A.1;

2. take the control of the hardware timer (8254 and APIC when available) and implement a virtual timer for Linux;

3. remove the basic control of the hardware interrupts from Linux by replacing all the `cli` and `sti` function calls (disable and enable interrupt flag, respectively) from the

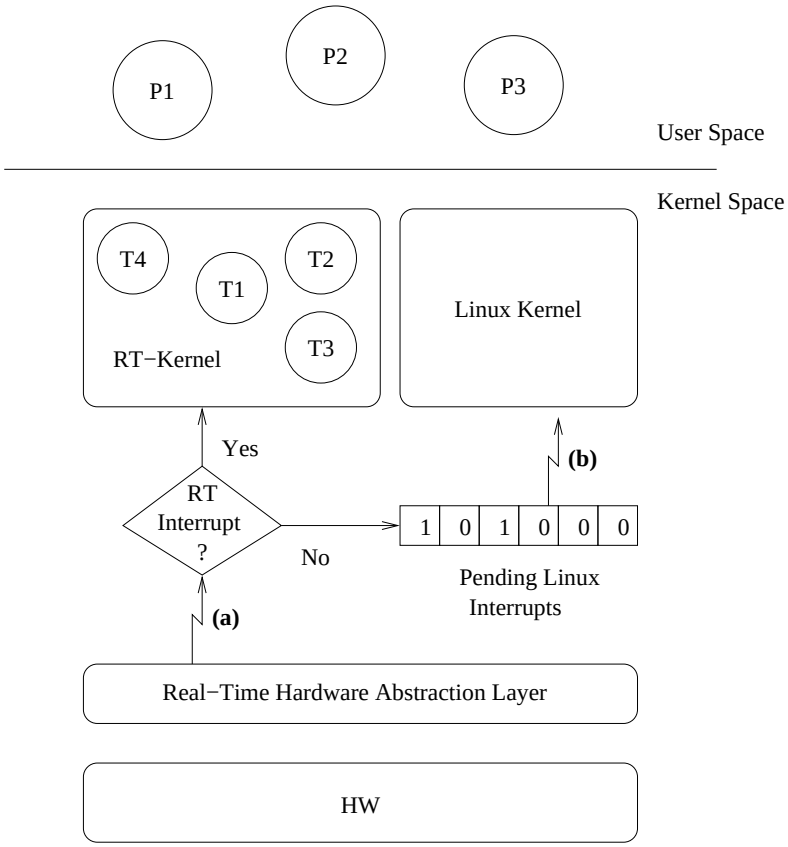

Figure A.1: Interrupt Abstraction.

kernel code so that Linux cannot disable hardware interrupts (but only their virtual counterparts).

The Interrupt Abstraction approach has been successfully implemented in some existing RTOSs, the most famous being RTLinux and RTAI. **RTLinux** [140] is a patch developed at *Finite State Machine Labs* (FSMLabs) to add hard real-time features to the standard Linux kernel. The project started in 1995 and it is released in two different versions: an Open Source (under GPL license) version, and a more featured commercial version. The RTLinux patch implements a small and fast RTOS, compliant with the POSIX 1003.13 "minimal real-time system" profile. This means that it has basic thread management, IPC primitives, semaphores, signals, spinlocks, FIFOs, etc. Some function calls, however, do not follow the POSIX standard. RTLinux is covered by US Patent 5885745 issued on November 30th, 1999. The patent is not valid outside of the USA, but FSMLabs has expressed its intention to enforce the patent. This has generated a massive transition of community developers efforts towards RTAI.

RTAI is the acronym of *"Real-Time Application Interface"* [42, 3]. The project started as a variant of RTLinux in 1997 at Dipartimento di Ingegneria Aerospaziale of Politecnico

			Linux 2.4	RTAI
Interrupt latency	Idle system	Avg.	4.3	5.8
		Max.	32.5	25.9
	Loaded system	Avg.	14.3	17.9
		Max.	162.9	64.9
Task latency	Idle system	Avg.	49.7	33.2
		Max.	332.3	68.0
	Loaded system	Avg.	3147.5	63.0
		Max.	84585.0	142.0

Table A.1: Interrupt and task latency in the standard Linux 2.4 and in RTAI. All measures are in microseconds.

di Milano (DIAPM), Italy. The project is under LGPL license, and it is supported by a large community of developers based upon the Open Source model. Although the RTAI project started from the original RTLinux code, the API of the projects evolved in opposite directions. In fact, the main developer (Prof. Paolo Mantegazza) has rewritten the code adding new features and creating a more complete and robust system. With respect to the Open Source version of RTLinux, RTAI has a greater amount of supported architectures and a larger number of mechanisms for communication between processes.

An in-depth comparison of the latency between the standard Linux kernel 2.4 and RTAI on a platform with an Axis ETRAX processor has been done in [14]. The main results of the experiments are summarized in Table A.1. The values are measured without system load (*Idle system*) and when a load is applied (*Loaded system*), respectively. The upper part of Table A.1 shows the values of the *interrupt latency* (i.e., the time between the interrupt arrival and the execution of the interrupt handler). Notice that, on average, the RTHAL imposes a slight increase in latency, due to the additional overhead of intercepting every interrupt with the RTHAL. However, the maximum latency values using RTAI are much smaller than using a standard kernel (especially for a loaded system), meaning that the determinism and the responsiveness of the system have been actually improved.

The bottom part of Table A.1 shows the values of the task latency (i.e., the time between the interrupt arrival and when the task starts processing). Task latency is essentially composed by two components: the interrupt and the scheduling latencies. In this case, the behaviour of the two systems differs even in those situations where the interrupt latency values were almost close. This difference is due to how the Linux scheduler works: while on RTAI a real-time process is scheduled maintaining the interrupts disabled, on Linux the interrupts are re-enabled after the interrupt handler finishes, leading to much more non-determinism.

FSMLabs, the owner of RTLinux, did not clear up the uncertainty around the legal repercussion of its patent on RTAI. For this reason, the RTAI community has developed the *Adaptive Domain Environment for Operating Systems* (Adeos) nanokernel as alternative

for RTAI's core, to get rid of the old kernel patch and exploit a more structured and flexible way to add a real-time environment to Linux [42]. The purpose of the Adeos nanokernel is not limited to be the new RTAI's core, but also to provide a flexible environment for sharing hardware resources among multiple operating systems (or among multiple instances of the same OS).

A.3.1 Advantages

It is important to highlight the advantages of using the Interrupt Abstraction approach. First of all, the latency reduction is really effective: measurements show a maximum latency below the microsecond [64] on a Intel Pentium M processor at 1.60 GHz. This allows the implementation of very fast control loops for applications like vibrational control. Also, thanks to the interrupt virtualization, it is possible to use a full-featured OS like Linux for the non-real-time activities. As a matter of fact, even the most critical control application includes non real-time activities, like logging and monitoring, man-machine interface, remote access through Internet, and so on. Using a system like Linux can reduce considerably the effort in developing this part of the system, and the programmer can focus on the most critical part. Finally, a further advantage is the possibility of developing and then executing the code on the same hardware platform, simplifying considerably the complexity of the development environment.

A.3.2 Limitations of RTLinux and RTAI

Both RTLinux and RTAI in their basic versions suffer from some software engineering and programming problems. As shown in Figure A.1, the real-time subsystem (RTOS and tasks) executes in the same memory space and with the same privileges as the Linux kernel code. This means that there is no protection of memory between the real-time tasks and the Linux kernel. The real-time tasks are typically executed as modules dynamically loaded into the kernel. Therefore, a real-time task with errors (like wrong memory references, or unbounded execution time) may crash the entire system. Such situation is frequent during debugging and development, and it is a very common experience for programmers of such systems to reboot the computer several times before identifying the error. Both the commercial version of RTLinux and the most recent versions of RTAI partially solved this problem. In particular, RTAI supports the LXRT interface that lets developers try out real-time tasks in user space, where memory protection is enabled, at the cost of some more latency. Once the task has been properly debugged, it can be executed on RTAI without changing the task code. The LXRT mechanism has then evolved to the Xenomai project which will be described in the next section.

Another problem is the communication with the non-real-time Linux activities. In particular, the real-time subsystem *cannot use the Linux device drivers*. For example, both

RTLinux and RTAI have their own network protocol stacks for communicating through Ethernet and with the serial driver, because the real-time tasks cannot use the Linux protocol stack. Therefore, in the same system, there is duplication of code for both the real-time and the non-real-time parts. Moreover, the effort of developing device drivers is always a consistent part of the development.

A.3.3 The Xenomai approach

A spin-off of the RTAI project [2], Xenomai [45] brings the concept of virtualization one step further. Like RTAI, it uses the Adeos nanokernel to provide the interrupt virtualization, but it allows a real-time task to execute in user space. Xenomai uses the concept of domain provided by Adeos extensively. In particular, Xenomai defines a *primary domain*, which is controlled by the RTOS (called *RT-Nucleus*), and a secondary domain, which is controlled by the Linux scheduler. A real-time task can execute in user space or in kernel space. Normally, it starts in the *primary domain*, where it remains as long as it invokes only the RTOS API. When the real-time task invokes a function belonging to the Linux standard API or libraries, it is automatically *migrated* to the secondary domain, under the control of the Linux scheduler. However, it keeps its real-time priority, being scheduled with the SCHED_FIFO or SCHED_RR Linux policies. While the real-time task is in the secondary mode, it can experience some delay and latency, due to the fact that it is scheduled by Linux. However, at any time after the function call has been completed, it can go back to the primary mode by explicitly calling a function. In this way, at the cost of some limited unpredictability, the real-time programmer can use the full power of Linux also for real-time applications. In fact, real-time tasks can run in their own memory space and are protected from the other tasks. This isolation facilitates debugging and fault confinement, reducing considerably the development time, and adding robustness to software faults.

Regarding the latency, the tasks in primary domain experience latencies comparable with the execution on RTAI. In secondary domain, instead, the maximum latency is higher, but still acceptable. As stated by Philippe Gerum [46], Xenomai leader, improvements on the standard Linux latency can help Xenomai too. For this reason, Xenomai developers put a constant effort in ensuring the simplicity and minimal invasivity of their approach with respect to the Linux code, thus that it is possible to use Xenomai along with separate solutions (like the PREEMPT_RT presented in Section A.4.3) proposed by other developers.

[2] Xenomai is the evolution of the Fusion project (in turn a generalization of the LXRT interface), which was an effort to execute real-time RTAI tasks in user space.

A.4 Making the Kernel More Predictable

An alternative to interrupt and hardware abstractions consists on making the Linux kernel more deterministic, by improving some parts that do not allow a predictable behaviour. As we discussed in Section A.2, the main sources of unpredictable behaviour in Linux are the kernel latency, the timing resolution and the process scheduling [8, 48]. We now present all the solutions that have been proposed to address these issues.

A.4.1 Reducing Kernel Latency

In the past, two different approaches were proposed to reduce kernel latency in the 2.4 version of the Linux kernel. These two approaches were the *Low Latency Patch* and the *Preemptible Kernel Patch*, respectively. The former patch was introduced by Ingo Molnar and then maintained by Andrew Morton [91]. Rather than attempting a brute-force approach (i.e., preemption) in a kernel that is not designed for it, this patch focuses on introducing explicit preemption points in blocks of kernel code that may execute for long intervals of time. The idea is to find places that iterate over large data structures and figure out how to safely introduce a call to the scheduler. Sometimes this implies releasing a spinlock, scheduling and then reacquiring the spinlock, which is also known as *"lock breaking"*.

A different strategy has been proposed by Robert Love with MontaVista's Preemptible Kernel Patch [82]. This patch makes the kernel preemptible, just like user-space: if a high priority task becomes runnable, the patch allows a context switch even if another process is running in kernel context. Hence, it becomes possible to preempt a process at any point, as long as the kernel is in a consistent state (i.e., no lock is held). Kernel preemption is subject only to Symmetric Multi-Processing (SMP) locking constraints (i.e., spinlocks are used as markers for regions of preemptibility). With the advent of Linux 2.6, Robert Love's patch has been accepted in the mainline kernel, thus that the Linux kernel has become a fully preemptive kernel [82], unlike most existing operating systems (UNIX variants included).

A comparison of the two techniques has been performed by Clark Williams [135] and is summarized in Table A.2. The hardware used for the experiments is a 700 MHz AMD Duron system with 360MB RAM and a 20GB Western Digital IDE drive connected to a VIA Technologies VT82C686 IDE controller. The experiments show that the maximum latency on a native 2.4.17 standard kernel can be as high as 232.7 msec, which is not a negligible value even on Desktop machines. The Preemptible Kernel Patch can reduce this value, but it is the Low Latency Patch that really makes the difference in the latency behaviour of the kernel, allowing a maximum latency of 1.3 msec. Obviously, the two techniques can also be combined together. In this case, the result is quite unexpected: the

	Linux 2.4.17	Preempt. Kernel	Low Latency	Both Patches
Avg.	88 μsec	53.8 μsec	54.2 μsec	52 μsec
Max.	232.7 msec	45.3 msec	1.3 msec	1.2 msec

Table A.2: Average and maximum latency values using a standard Linux 2.4.17, the Preemptible Kernel and the Low Latency patches.

maximum latency measured is 1.2 msec, which is a small improvement with respect to the gain obtained using only the Low Latency Patch.

A.4.2 Improving Timing Resolution

The fact that periodic timer interrupts are not suitable for real-time kernels is well known in the literature [8]. For this reason, most of the existing real-time kernels provide a *"High Resolution Timers"* (HRT) API, that issues the interrupts aperiodically — i.e., the system timer is programmed to generate the interrupt after an interval of time that is not constant, but that depends on the next event scheduled by the operating system. Often, these implementations exploit also processor-specific hardware (like the APIC on modern x86 processors) to obtain a better timing resolution (typically, in the order of microseconds or even fractions of microseconds).

There are two different projects to provide HRT in the Linux kernel. The first project, called *High-Resolution POSIX Timers* [15], started in 2001 as a separate patch and never became part of the standard kernel.

Very recently, a newer API developed by Thomas Gleixner has been accepted into the 2.6.16 version of the mainline kernel [94]. Rather than using a "timer wheel" data structure, this implementation uses a time-sorted linked list, with the next timer to expire being at the head of the list. A separate red/black tree is also used to enable the insertion and removal of timer events without scanning through the list. A new type (called `ktime_t`) is used to store a time value in nanoseconds and it is meant to be used as an opaque structure. Interestingly, its definition changes depending on the underlying architecture. On 64-bit systems, it is just a 64-bit integer value in nanoseconds. On 32-bit machines, instead, it is a two-field data structure: one 32-bit value holds the number of seconds and the other holds nanoseconds. The order of the two fields depends on whether the host architecture is big-endian or not — they are always arranged so that the two values can, when needed, be treated as a single 64-bit value. Doing things this way complicates the header files, but provides efficient time value manipulation on all architectures.

Kernel	sys load	Aver	Max	Min	StdDev
Vanilla-2.6.12	None	5.8	51.9	5.6	0.3
	Ping	5.8	49.1	5.6	0.8
	lm. + ping	6.1	53.3	5.6	1.1
	lmbench	6.1	77.9	5.6	0.8
	lm. + hd	6.5	128.4	5.6	3.4
	DoHell	6.8	555.6	5.6	7.2
RT-V0.7.51-02	None	5.7	48.9	5.6	0.2
	Ping	7.0	62.0	5.6	1.5
	lm. + ping	7.9	56.2	5.6	1.9
	lmbench	7.3	56.1	5.6	1.4
	lm. + hd	7.3	70.5	5.6	1.8
	DoHell	7.4	54.6	5.6	1.4
Ipipe-0.7	None	7.2	47.6	5.7	1.9
	Ping	7.3	48.9	5.7	0.4
	lm.+ ping	7.6	50.5	5.7	0.8
	lmbench	7.5	50.5	5.7	0.9
	lm. + hd	7.5	50.5	5.7	1.1
	DoHell	7.6	50.5	5.7	0.7

Table A.3: Latency comparison between Standard Linux, Linux with the PREEMPT_RT patch, and Adeos. All numbers are in microseconds.

A.4.3 The PREEMPT_RT patch

The latest modification is the PREEMPT_RT patch by Ingo Molnar [89]. This work brings the kernel preemption to an unprecedent level of sophistication by introducing the Priority Inheritance Protocol in the kernel locks. The Priority Inheritance (PI) protocol, first proposed by Sha et al. [117], solves the problem of unbounded *priority inversion*. A priority inversion happens when a high priority task must wait for a low priority task to complete a critical section of code and release the lock. If the low priority task is preempted by a medium priority task while holding the lock, the high priority task will have to wait for a long time. The priority inheritance protocol dictates that in this case, the low priority task *inherits* the priority of the high priority task while holding the lock, preventing the preemption by medium priority tasks. Refer to Section 1.8 for an in-depth explanation of the problem.

In the general case (i.e., nested spinlocks, readers/writers locks) the priority inheritance mechanism is a complex algorithm to implement. Nevertheless, it can help reduce the latency of Linux activities even further, reaching the level of the *Interrupt Abstraction* methods.

In Table A.3 we report the results of a comparison between a standard Linux (denoted as Vanilla-2.6.12), the same Linux with the RT patch applied, and the Adeos microkernel, used by both RTAI and Xenomai (denoted with Ipipe-0.7)[3].

[3]The results are provided by Paolo Mantegazza and are taken from `https://mail.rtai.org/pipermail/rtai/2005-October/013265.html`. We are not aware of the original source of these numbers.

On each kernel configuration, a number of standard tests have been run to stress the system in order to measure the worst-case latency. The interrupt latency (i.e., the time it takes from the raise of the interrupt signal to the execution of the first instruction of the interrupt handler) has been measured in all cases. As the reader can see from the table, the maximum latencies are quite high in the Vanilla kernel (in the order of half a millisecond), while the maximum latencies in the RT kernel and with the Adeos microkernel are comparable. However, other tests seems to show a slight advantage to the Adeos approach. It is important to point out that these numbers are referred to *interrupt latency*, while task latency can be much higher and depends also upon timer resolution and scheduling latency.

A.4.4 Resource Reservations

As we have seen, the scheduling policies offered by Linux are not suitable for supporting the execution of real-time applications. A real-time general-purpose OS should support scheduling policies providing *temporal isolation* among the running processes. This means that the timely execution of a process should not be affected by the behaviour of the other processes executing on the system. This way, if a process misbehaves, and tries to use all the resources of the system, it cannot starve the other processes. The same problem is present in the Interrupt Abstraction methods: if a real-time task enters an infinite loop of code, the other low priority activities in the system cannot execute anymore. It is important then to provide *temporal isolation* among different tasks, similarly to the way the Linux kernel provides memory protection. Such characteristics would also help in mixing hard and soft real-time applications on the same operating system.

As we have seen in Chapter 4, the Resource Reservation mechanism [84, 85] is an effective way for providing such temporal protection in GPOSs. This technique provides support for time-sensitive applications by allowing the integration of classical real-time techniques, developed to meet timing constraints on RTOSs, with the general-purpose allocation strategies used on GPOSs. The basic idea behind the resource reservation technique is to *reserve* a fraction of the time to real-time applications. This way, real-time priorities can be securely used even by non-privileged users. The mechanism works as follows. Each real-time process is assigned a "reservation" (Q_i, T_i), meaning that the process is reserved the processor for a time of length Q_i every period T_i. During its execution, the task is executed at an appropriate real-time priority. However, if the task tries to execute for a longer time, then it is suspended and resumed later. This way, each task is constrained to not use more than its reserved share — i.e., a maximum of Q_i every P_i units of time. Refer to Chapter 4 for a description of some scheduling algorithms based on Resource Reservation.

Notable examples of Resource Reservations in the Linux OS are Linux/RK and RED

Hook	Idle	10 tasks	20 tasks	30 tasks
creation	119	117	107	105
termination	48	44	39	35
unblock	316	387	421	483
block	138	6431	8101	9164
budget exhaustion	202	252	276	312

Table A.4: Overhead introduced by the hooks. All numbers are in nanoseconds.

Linux [54, 133, 104].

A real-time scheduler based on Resource Reservation has been developed for Linux 2.4.18 within the OCERA (*"Open Components for Embedded Real-time Applications"*) European project, and it is available as Open Source code. The implementation details of this project have already been described in Section 5.4. The scheduler implements the CBS [6, 9, 39], the GRUB [76, 77], the HGRUB, and the GRUB-PA [111, 112] scheduling algorithms. The execution of the hooks introduces an overhead which is *at most* 10 μsec (see Table A.4) on a AMD Athlon XP at 1.6 GHz running Linux 2.4.27 with High Resolution Timers and Linux Trace Toolkit patches. Although not comparable with the values obtainable using Interrupt Abstraction, this overhead is acceptable for most soft real-time applications.

A Resource Reservation scheduling policy for Linux has been developed also by Davide Libenzi with the SCHED_SOFTRR project [74]. Using this policy, a task can run with real-time priority, but it is subject to a constraint on the maximum processor time it can consume. Thus, non-privileged users can have deterministic latencies when running time-sensitive applications, while system stability and fairness are enforced by the bound.

Another scheduling policy, called SCHED_ISO (that stands for "Isochronous Scheduling") has been implemented by Con Kolivas [66]. Also this policy does not require superuser privileges and is starvation-free. Tasks running under the SCHED_ISO policy actually execute as SCHED_RR unless the processor usage exceeds a specified limit (i.e., 70 percent). The value of this limit can be configured through the `proc` filesystem.

A.5 Summary

Linux has become very popular for supporting real-time applications for many reasons, among the others the availability of a huge amount of programs distributed with Open Source license, the robustness and flexibility of the kernel and its standard interface. Many projects have been proposed to make Linux more real-time, both by using the Interrupt Abstraction approach, and by directly modifying kernel internals (preemption patches, and resource reservations).

The choice of which Linux flavor to use for executing a real-time application depends entirely on the requirements of the application. For hard real-time applications with very

small constants of time (below the milliseconds), it is still necessary to use RTLinux, RTAI or Xenomai, since they can provide very low latencies. Also, RTAI and Xenomai guarantee nice integration with control design tools, like Scilab/Scicos, Matlab/Simulink, etc.

On the other hand, thanks to the constant attention to reducing the latency of the standard Linux kernel, soft real-time applications, or even hard real-time applications with large constants of time, can be scheduled directly by the Linux scheduler, in case with the help of a resource reservation scheduler like GRUB, SCHED_SOFTRR or SCHED_ISO.

In the future, we believe that the two approaches will merge into a single product, able to provide different levels of services and latencies to different applications. In this sense, Xenomai is paving the way to such integration.

Bibliography

[1] *The Apache Software Foundation*. `http://www.apache.org`.

[2] *Condor, High Throughput Computing*. `http://www.condorproject.org`.

[3] *RTAI - Beginner's Guide*. `http://www.aero.polimi.it/~rtai/documentation/articles/guide.html`.

[4] *RTSim ("Real-Time system SIMulator")*. `http://rtsim.sourceforge.net`.

[5] *SimpleScalar LLC*. `http://www.simplescalar.com`.

[6] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium (RTSS)*, pages 4 – 13, Madrid, Spain, December 1998.

[7] Luca Abeni and Giorgio Buttazzo. Hierarchical QoS Management for Time Sensitive Applications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

[8] Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. A Measurement-Based Analysis of the Real-Time Performance of Linux. In *Proceedings of the* 8^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 133 – 142, San Jose, California, September 2002.

[9] Luca Abeni and Giuseppe Lipari. Implementing Resource Reservations in Linux. In *Real-Time Linux Workshop*, Boston (MA), December 2002.

[10] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a Reservation Feedback Scheduler. In *Proceedings of the 23rd IEEE Real Time Systems Symposium (RTSS)*, pages 71–80, Austin, US, December 2002.

[11] Luca Abeni, Claudio Scordino, Giuseppe Lipari, and Luigi Palopoli. Serving Non Real-Time Tasks in a Reservation Environment. In *Proceedings of the 9th Real-Time Linux Workshop (RTLW)*, Linz, Austria, November 2007.

[12] Nevine AbouGhazaleh, Daniel Mossé, Bruce Childers, and Rami Melhem. Toward the placement of power management points in real time applications. In *Workshop on Compilers and Operating Systems for Low Power (COLP01)*, Barcelona, Spain, 2001.

[13] Nevine AbouGhazaleh, Daniel Mossé, Bruce Childers, and Rami Melhem. *Toward The Placement of Power Management Points in Real Time Applications.* Kluwer Academic Publishers, 2003.

[14] Martin P. Andersson and Jens Henrik Lindskov. Real-time linux in an embedded environment. In *Master of Science Thesis*, Lund Institute of Technology, Lund University, Sweden, January 2003.

[15] George Anzinger. *High Resolution POSIX timers.* `http://high-res-timers.sf.net/`.

[16] Hakan Aydin. On aggressive speed adjustment algorithm for real-time dynamic voltage scaling. Technical report, Department of Computer Science, George Mason University, US, 2006.

[17] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the IEEE Real-Time System Symposium (RTSS)*, pages 95–105, London, UK, December 2001.

[18] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, May 2004.

[19] Scott A. Banachowski, Timothy Bisson, and Scott A. Brandt. Integrating best-effort scheduling into a real-time system. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 139–150, 2004.

[20] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[21] John C. R. Bennett and Hui Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proceedings of INFOCOM'96*, pages 120 – 128, March 1996.

[22] John C. R. Bennett and Hui Zhang. Why WFQ is not good enough for integrated services networks. In *Proceedings of NOSSDAV'96*, April 1996.

[23] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems. In *Proceedings of the*

3rd International Workshop on Worst-Case Execution Time Analysis, pages 21–38, Porto, Portugal, July 2003.

[24] Riccardo Bianchini and Ram Rajamony. Power and Energy Management for Server Systems. *Computer*, 37(11):68–74, November 2004.

[25] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe Lipari. Speed modulation in energy-aware real-time systems. In *Proceedings of the* 17^{th} *Euromicro Conference on Real-Time Systems*, pages 3 – 10, Palma de Mallorca, Spain, July 2005.

[26] Enrico Bini and Claudio Scordino. Optimal Two-Level Speed Assignment for Real-Time Systems. *To appear in International Journal of Embedded Systems (IJES), Special Issue on Low-Power Real-time Embedded Computing*, 2007.

[27] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. *The case for power management in web servers*. Kluwer Academic Publishers, 2002.

[28] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[29] Alan Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6:116–128, May 1991.

[30] Alan Burns, Guillem Bernat, and Ian Broster. A probabilistic framework for schedulability analysis. In *Proceedings of the EMSOFT*, pages 1–15, Philadelphia, PA, October 2003.

[31] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2nd edition*. Springer, 2005.

[32] Giorgio C. Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, January 2005.

[33] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 295–304, Orlando, Florida, December 2000.

[34] Marco Caccamo, Giorgio C. Buttazzo, and Lui Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(7):835–849, 2002.

[35] Jean Carle and David Simplot-Ryl. Energy-efficient area monitoring for sensor networks. *IEEE Computer*, 37:40–46, February 2004.

[36] Anantha P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[37] Edward J. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliff, NJ, 1973.

[38] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers,* 3^{rd} *edition*. O'Reilly, February 2005.

[39] Tommaso Cucinotta, Luigi Palopoli, Luca Marzario, Giuseppe Lipari, and Luca Abeni. Adaptive Reservations in a Linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 238–245, Toronto, Canada, May 2004.

[40] Zhong Deng and Jane W. S. Liu. Scheduling real-time applications in open envirovment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, page 308, December 1997.

[41] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, pages 807 – 813, 1974.

[42] Lorenzo Dozio and Paolo Mantegazza. Real-Time Distributed Control Systems Using RTAI. In *Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, pages 11 – 18, Hakodate, Hokkaido, Japan, May 2003.

[43] Ajay Dudani, Frank Mueller, and Yifan Zhu. Energy-Conserving Feedback EDF Scheduling for Embedded Systems with Real-Time Constraints. In *Proceedings of ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, pages 213–222, June 2002.

[44] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered calculation of worst-case execution times. In *Proceedings of the* 6^{th} *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 51–62, San José, CA, October 2003.

[45] Philippe Gerum. *The XENOMAI Project, Implementing a RTOS emulation framework on GNU/Linux*, November 2002.

[46] Philippe Gerum. RTAI/fusion and Ingo Molnar's extension. Xenomai mailing list, `https://mail.rtai.org/pipermail/rtai/2005-June/011778.html`, June 2005.

[47] T.M. Ghazalie and Theodore P. Baker. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time System*, 9:31–67, 1995.

[48] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity OS. In *Proceedings of the* 5^{th} *symposium on Operating systems design and implementation(OSDI'02)*, volume 36, pages 165 – 180, Boston, MA, December 2002.

[49] Flavius Gruian. Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS Processors. In *International Symposium on Low Power Electronics and Design*, pages 46–51, Huntington Beach (CA), US, August 2001.

[50] Flavius Gruian. *Energy-Centric Scheduling for Real-Time Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund, Sweden, November 2002.

[51] Philip Holman and James H. Anderson. Implementing pfairness on a symmetric multiprocessor. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 544–553, 2004.

[52] Inki Hong, Darko Kirovski, Gang Qu, Miodrag Potkonjak, and Mani Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th Design Automation Conference*, pages 176–181, 1998.

[53] Inki Hong, Gang Qu, Miodrag Potkonjak, and Mani B. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium (RTSS)*, pages 178–187, Madrid, Spain, December 1998.

[54] TimeSys Inc. Timesys linux `http://www.timesys.com`.

[55] Intel Corporation, `http://developer.intel.com/design/mobile/centrinoplatformoverview.htm`. *Intel Centrino Mobile Technology*.

[56] Intel Corporation, `http://developer.intel.com/design/intelxscale/`. *Intel XScale Technology*.

[57] Intel Corporation. *Intel PXA250 and PXA210 Application Processors Design Guide*, February 2002.

[58] Intel Corporation. *Intel PXA250 and PXA210 Application Processors Developer's Manual*, February 2002.

[59] Intel Corporation. *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*, March 2004.

[60] Intel Corporation. *Intel PXA27x Processor Family Power Requirements*, 2004.

[61] International SEMATECH. International technology roadmap for semiconductors. `http://public.itrs.net/`.

[62] Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. In *ACM SIGACT*, volume 36, pages 63–76, June 2005.

[63] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 197–202, Monterey, CA, August 1998.

[64] Panagiotis Issaris. RTAI Testsuite LiveCD. `http://issaris.org/~takis/projects/rtai/livecd/`.

[65] Woonseok Kim, Dongkun Shin, Han-Saem Yun, Jihong Kim, and Sang Lyul Min. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In *Proceedings of the 8^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 219–228, San Jose, California, September 2002.

[66] Con Kolivas. Isochronous class for unprivileged soft RT scheduling. `http://ck.kolivas.org/patches/SCHED_ISO`.

[67] Pavan Kumar and Mani Srivastava. Power-aware multimedia systems using run-time prediction. In *Fourteenth International Conference on VLSI Design*, pages 64–69, 2000.

[68] Pavan Kumar and Mani Srivastava. Predictive Strategies for Low-Power RTOS Scheduling. In *Proceedings of the IEEE International Conference On Computer Design: VLSI In Computers & Processors (ICCD '00)*, pages 343–348, Austin, Texas, USA, September 2000.

[69] Seongsoo Lee and Takayasu Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37^{th} Design Automation Conference*, pages 806–809, Los Angeles, CA USA, June 2000.

[70] Yann-Hang Lee and C.M. Krishna. Voltage-Clock Scaling for Low Energy Consumption in Real-Time Embedded Systems. In *Sixth International Conference on Real-Time Computing Systems and Applications*, pages 272–279, Hong Kong, China, December 1999.

[71] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, December 2003.

[72] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 261–270, December 1987.

[73] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environment. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.

[74] Davide Libenzi. SCHED_SOFTRR Linux Scheduler Policy. `http://xmailserver.org/linux-patches/softrr.html`.

[75] Giuseppe Lipari. *Resource Reservation in Real-Time Systems*. PhD thesis, Scuola Superiore S.Anna, 2000.

[76] Giuseppe Lipari and Sanjoy K. Baruah. Greedy reclamation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the* 12^{th} *Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

[77] Giuseppe Lipari and Sanjoy K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 26 – 35, Taipei, Taiwan, May 2001.

[78] Giuseppe Lipari and Claudio Scordino. Linux and real-time: Current approaches and future opportinities. In *IEEE International Congress ANIPLA '06*, Rome, Italy, November 2006.

[79] C.L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[80] Yanbin Liu and Aloysius K. Mok. An integrated approach for applying dynamic voltage scaling to hard real-time systems. In *Proceedings of the* 9^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 116–123, Washington, DC USA, May 2003.

[81] Jacob R. Lorch and Alan J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *ACM SIGMETRICS*, pages 50–61, Cambridge, MA, June 2001.

[82] Robert Love. *Linux Kernel Development,* 2^{nd} *edition*. Novell Press, February 2005.

[83] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. IRIS: A New Reclaiming Algorithm for Server-Based Real-Time Systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–218, 2004.

[84] Clifford W. Mercer, Raguanathan Rajkumar, and Hideyuki Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, NC, 1993.

[85] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[86] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. In *Proceedings of IEEE international conference on Multimedia Computing and System*, May 1994.

[87] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Temporal protection in real-time operating systems. In *Proceedings of th 11th IEEE workshop on Real-Time Operating System and Software*, pages 79–83. IEEE, May 1994.

[88] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. A realistic variable voltage scheduling model for real-time applications. In *Proceedings of the International Conference on Computer Aided Design*, pages 726–731, San José, CA USA, November 2002.

[89] Ingo Molnar. *Real-Time Preempt.* `http://people.redhat.com/mingo/realtime-preempt/`.

[90] Justin Moore, Ratnesh Sharma, Rocky Shih, Jeff Chase, Chandrakant Patel, and Parthasarathy Ranganathan. Going Beyond CPUs: The Potential of Temperature-Aware Data Center Architectures. In 1^{st} *Workshop on Temperature-Aware Computer Systems*, June 2004.

[91] Andrew Morton. *Low Latency Patches.* `http://www.zipworld.com.au/~akpm/linux/schedlat.html`.

[92] Daniel Mossé, Hakan Aydin, Bruce Childers, and Rami Melhem. Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications. In *Workshop on Compilers and Operating Systems for Low Power (COLP00)*, 2000.

[93] Motorola, `http://e-www.motorola.com/webapp/sps/library/prod_lib.jsp`. *MPC5200: 32 bit Embedded Processor.*

[94] Linux Weekly News. *Kernel development.* `http://lwn.net/Articles/167315/`.

[95] Luigi Palopoli, Luca Abeni, and Giuseppe Lipari. On the applications of hybrid control to CPU Reservations. In *Proceedings of Hybrid systems Computation and Control (HSCC03), LNCS series*, pages 389–404, Prague, 2003.

[96] Luigi Palopoli, Giuseppe Lipari, Gerardo Lamastra, Luca Abeni, Bolognini Gabriele, and Paolo Ancilotti. An object oriented tool for simulating distributed real-time control systems. *Software: Practice and Experience*, 32, 2002.

[97] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[98] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the* 18^{th} *ACM Symposium on Operating System Principles*, pages 89–102, Banff, Canada, October 2001.

[99] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the* 7^{th} *ACM International Conference on Mobile Computing and Networking*, pages 251–259, Rome, Italy, July 2001.

[100] Johan Pouwelse, Koen Langendoen, and Henk Sips. Energy priority scheduling for variable voltage processors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 28–33, 2001.

[101] Ala' Qadi, Steve Goddard, and Shane Farritor. A dynamic voltage scaling algorithm for sporadic tasks. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 52–62, Cancun, Mexico, 2003.

[102] Gang Quan, Linwei Niu, Xiaobo Sharon Hu, and Bren Mochocki. Fixed priority based real-time scheduling for reducing energy on variable voltage processors. In *Proceedings of the* 25^{th} *IEEE Real-Time Systems Symposium (RTSS)*, pages 309–318, Lisbon, Portugal, December 2004.

[103] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits*. Prentice Hall, second edition, 2002.

[104] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[105] Glenn E. Reeves. Re: [Fwd: FW: What really happened on Mars?]. Technical report, `http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html`, December 1997.

[106] Ismael Ripoll, Pavel Pisa, Luca Abeni, Paolo Gai, Agnes Lanusse, Sergio Saez, and Bruno Privat. WP1 - RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis. Technical report, OCERA, 2002.

[107] Cosmin Rusu, Alexandre Ferreira, Claudio Scordino, Aaron Watson, Rami Melhem, and Daniel Mossé. Energy-efficient real-time heterogeneous server clusters. In *Proceedings of the* 12^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 418–428, San Jose, California, US, April 2006.

[108] Saowanee Saewong and Ragunathan Rajkumar. Optimal static voltage-scaling for real-time systems. Technical report, Real-Time and Multimedia Systems Laboratory, Carnegie Mellon University, Pittsburgh PA 15213, 2002.

[109] Saowanee Saewong and Ragunathan Rajkumar. Practical voltage-scaling for fixed-priority RT-systems. In *Proceedings of the* 9^{th} *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 106–114, Washington, DC, May 2003.

[110] Claudio Scordino and Enrico Bini. Optimal Speed Assignment for Probabilistic Execution Times. In 2^{nd} *Workshop on Power-Aware Real-Time Computing (PARC'05)*, NJ, September 2005.

[111] Claudio Scordino and Giuseppe Lipari. Energy Saving Scheduling for Embedded Real-Time Linux Applications. In 5^{th} *Real-Time Linux Workshop*, Valencia, Spain, November 2003.

[112] Claudio Scordino and Giuseppe Lipari. Using resource reservation techniques for power-aware scheduling. In *Proceedings of the* 4^{th} *ACM International Conference on Embedded Software*, pages 16–25, Pisa, Italy, September 2004.

[113] Claudio Scordino and Giuseppe Lipari. A resource reservation algorithm for power-aware scheduling of periodic and aperiodic real-time tasks. *IEEE Transactions on Computers*, 55(12):1509–1522, 2006.

[114] Jerry E. Sergent and Al Krum. *Thermal Management Handbook*. Mc-Graw Hill, 1998.

[115] Kiran Seth, Aravindh Anantaraman, Frank Mueller, and Eric Rotenberg. FAST: Frequency-aware static timing analysis. In *Proceedings of the* 24^{th} *IEEE Real-Time Systems Symposium (RTSS)*, pages 40–51, Cancun, Mexico, December 2003.

[116] Lui Sha and John B. Goodenough. Real-Time Scheduling Theory and ADA. *IEEE Computer*, 23:53–62, April 1990.

[117] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[118] Dongkun Shin and Jihong Kim. Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems. In *Proceedings of ASP-DAC '04*, pages 653–658, January 2004.

[119] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2):20–30, March 2001.

[120] Dongkun Shin, Woonseok Kim, Jaekwon Jeon, Jihong Kim, and Sang Lyul Min. SimDVS: An Integrated Simulation Environment for Performance Evaluation of Dynamic Voltage Scaling Algorithms. In *Power-Aware Computer Systems (PACS'02)*, pages 141–156, Cambridge, MA, February 2002.

[121] Youngsoo Shin, Kiyoung Choi, and Takayasu Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer Aided Design*, pages 365–368, November 2000.

[122] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, July 1989.

[123] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 2–11, San Juan, Puerto Rico, December 1994.

[124] Marco Spuri and Giorgio C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Journal of Real-Time Systems*, 10(2):179–210, March 1996.

[125] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, October 1988.

[126] John A. Stankovic, Krithi Ramamritham, Marco Spuri, and Giorgio Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[127] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 288–299, December 1996.

[128] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, 4(1), January 1995.

[129] The MathWorks. MATLAB, the language of technical computing. `http://www.mathworks.com/products/matlab/`.

[130] Transmeta Corporation, `http://www.transmeta.com/crusoe/`. *The Crusoe Processor.*

[131] Transmeta Corporation, `http://www.transmeta.com`. *Crusoe^{TM} Processor Model TM5800 Version 2.1 Data Book Revision 2.01*, June 2003.

[132] Department of Computer Science University of Pittsburgh. PARTS — Power Efficiency Test. `http://www.cs.pitt.edu/PARTS/demos/efficient`.

[133] Yu-Chung Wang and Kwei-Jay Lin. Enhancing the Real-Time Capability of the Linux Kernel. In *IEEE Real Time Computing Systems and Applications*, pages 11–20, Hiroshima, Japan, October 1998.

[134] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Real-Time Systems*, volume 21, pages 241–268, November 2001.

[135] Clark Williams. *Linux Scheduler Latency.* Red Hat Inc., `http://www.linuxdevices.com/articles/AT8906594941.html`, March 2002.

[136] Jia Xu and David Lorge Parnas. Priority scheduling versus pre-run-time scheduling. *The International Journal of Time-Critical Computing Systems*, 18(1):7–23, 2000.

[137] Ruibin Xu, Chenhai Xi, Rami Melhem, and Daniel Mossé. Practical PACE for embedded systems. In 4^{th} *ACM International Conference on Embedded Software (EMSOFT '04)*, pages 54–63, Pisa, Italy, September 2004.

[138] Ruibin Xu, Dakai Zhu, Cosmin Rusu, Rami Melhem, and Daniel Mossé. Energy-Efficient Policies for Embedded Clusters. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 1–10, June 2005.

[139] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the* 36^{th} *Annual Symposium on Foundations of Computer Science*, pages 374–382, Milwaukee, WI USA, October 1995.

[140] Victor Yodaiken. The RTLinux Manifesto. In *Proceedings of the Fifth Linux Expo*, Raleigh, North Carolina, March 1999.

[141] Yifan Zhu and Frank Mueller. Feedback EDF Scheduling Exploiting Dynamic Voltage Scaling. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 84–93, Toronto, Canada, May 2004.