Scuola Superiore Sant'Anna
di Studi Universitari e di Perfezionamento

Major area of Embedded Systems

Retis Lab.

PhilosophiæDoctor (Ph.D.) Thesis

Power management of real-time systems under communication constraints

Tutor:
Prof. Giorgio Buttazzo

Author:
Francesco Esposito

Academic Year 2010 - 2011

To all the people
who believe in me...

Contents

Preface 6

1 Introduction **9**

1.1 Real-Time Systems 10
1.2 Main definitions and modeling 12
1.3 Classification of scheduling algorithms 14
1.4 Periodic task scheduling 15
1.4.1 Utilization factor 17
1.4.2 Rate Monotonic scheduling 17
1.4.3 Earliest deadline First 20
1.4.4 EDF with deadline less or equal to the periods 22
1.5 Demand Bound Function (DBF) 24
1.6 Supply Bound Function (SBF) 25
1.6.1 Schedulability analysis 27

2 Power management in Real Time Systems **29**

2.1 Existing power models 30
2.2 Power aware scheduling 34
2.2.1 Dynamic Voltage/Frequency Scaling (DVS) 35
2.2.1.1 System-wide speed computation 35
2.2.1.2 Tasks with different power costs 36
2.2.1.3 Variable Computation Time 37
2.2.1.4 Reclaiming of unused computation time . . . 39
2.2.2 Dynamic Power Management (DPM) 40
2.2.3 Achieving predictability and Scheduling Anomalies . . 40

3 Power management in large-scale interconnected systems **45**

3.1 Related works 46
3.1.1 Problem description 47
3.1.2 Workload and Resource Models 49
3.1.3 Power Model 49
3.1.4 Energy Aware Scheduling (EAS) 51

3.1.4.1 Experimental Results 54
3.1.5 Discrete Energy Aware Scheduling (DEAS) 59
3.1.6 Workload Computation 65
3.1.6.1 Experimental Results 66

4 Power management of multiprocessor systems 71
4.1 Introduction 71
4.2 The case of Multiprocessor System-on-Chip (MPSoC) 72
4.2.1 History of MPSoCs 73
4.3 Simulation Infrastructure 74
4.3.1 Software support to dynamic bus assignment 76
4.4 Adaptive TDMA bus allocation 77
4.4.1 Task computation benefits 79
4.4.2 Bus Access Time and Periods 81
4.4.3 Quality of Control index 83
4.4.4 Power management considerations 84

5 Conclusions 89

A Erika 91

B MPARM 93

List of Figures

1.1 Queue of ready tasks 13
1.2 Rate Monotonic scheduling 19
1.3 Earliest Deadline First scheduling 21
1.4 demand function interval [7,22] 23
1.5 dbf and interesting deadlines in [0,12] 25
1.6 $sbf(t)$ and its linearization $lsbf(t)$ 26
1.7 $sbf(t)$ for $t = 25$ 26
1.8 $sbf(t)$ for $t = 25$ 28

2.1 CMOS power consumption trend 30
2.2 CMOS dynamic power consumption 31
2.3 CMOS short circuit power consumption 32
2.4 CMOS leakage power consumption 32
2.5 Taxonomy of power scheduling techniques 34
2.6 EDF: no power manag. (a), single speed (b) and PWM (c). . 37
2.7 system-wide speed selection (a) and per-task speed selection. 37
2.8 Two-speed energy management strategy. consumption. 38
2.9 Constant speed schema (a) and a two-speed scheme (b). . . . 39
2.10 Effect of the extra preemption overhead caused by DVS. . . . 41
2.11 Scheduling anomaly in the presence of resource constraints. . 42
2.12 Scheduling anomaly in the presence of non-preemptive tasks. 43
2.13 Effects of a permanent overload due to a speed reduction. . . 43

3.1 Node interface. 48
3.2 Bandwidth assignment. 48
3.3 Power model: transition costs. 51
3.4 DVS and DPM: transmission bandwidth awareness. 52
3.5 EAS: three possible processor executions speeds. 54
3.6 Power consumption: TI and DSPIC. 56
3.7 CPU comparison: $n_t = 4$, $B = 0.3$ and $n_B = 3$. 56
3.8 Average power consumption: $U = 0.3$, $n_t = 4$. 57
3.9 EAS with $B = 0.5$ and $n_B = 3$ and the TI processor. 58

3.10 Comparison: $B = 0.5$ and $n_B = 3$ and the TI processor. . . . 58
3.11 Comparison: $B = 0.5$, $n_B = 3$ and $n_t = 4$ (TI processor). . . . 59
3.12 Comparison: $B = 0.5$, $n_B = 3$ and $n_t = 4$ (DSPIC processor). 59
3.13 Example of the analysis' behaviour. 62
3.14 DEAS example. 64
3.15 Examples of effective workload. 65
3.16 Speed-Power characterization for different platforms. 67
3.17 Consumptions with a fully-DPM power model. 69
3.18 Consumptions with a fully-DVFS power model. 70
3.19 Consumptions with a Mixed power model. 70

4.1 Hardware architecture 75
4.2 Algorithm diagram 78
4.3 Task computation time variation 80
4.4 Task periods and service levels 81
4.5 Buss access time and service levels 82
4.6 Power-aware TDMA bus allocation schema 86
4.7 idle time caused by dynamic TDMA bandwidth assignement 87

List of Tables

1.1 RM Analysis: task set composed by three tasks (case 1) . . . 18
1.2 RM Analysis: task set composed by three tasks (case 2) . . . 19
1.3 task set with deadline less the period 22

3.1 Power model: allowed power modes. 51
3.2 Power profiles for processing devices. 55
3.3 CC2420 Transceiver power profile. 55

4.1 Service Levels 77
4.2 QoC Indexes. 84
4.3 case 1: two CPUs 88
4.4 case 2: four CPUs 88
4.5 case 3: five CPUs 88

Preface

Main objective of the thesis

Embedded and Real-Time Systems are systems often characterized by limited resources as: memory availability, processing power or battery lifetime. Since most of these systems are battery-operated, and for some applications human maintenance is prohibitive or not comfortable, energy consumption and system lifetime became crucial. For such a systems the objective of reducing power consumption is of paramount importance and it represents the main objective of this work. According with this requirement, this thesis focus on power management for Embedded and Real Time Systems and provides a novel set of strategies to cope with the problems of limiting power consumption and guaranteeing interconnection support. This last capability is fundamental since most of the Real Time and Embedded applications are distributed and they operate in a networked scenario.
Most of the modern embedded architectures provide the capability to reduce energy consumption by using DVS (Dynamic Voltage and Frequency Scaling) or DPM (Dynamic Power Management) strategies. These techniques represent the state of the art in matter of power management in last generation processors. They are typical techniques used on CPU[1] and devices to reduce power consumption through speed variation and power switching, respectively.

The effectiveness of DVS and DPM methods, needs to be taken into account in the development of a power management policies for systems characterized by the availability of DVS or DPM support.

[1]CPU: Central Processing Unit

Proposed solution

According with these requirements, the existing DVS and DPM methods would be combined in much more complex strategies to cope with the double-sided problem concerning limiting power consumption on one end, and guarantying the interconnection of the distributed systems on the other. This thesis focus on this issue: saving energy and guarantying interconnection among computing nodes in a networked scenario.
Some novel strategies concerning the problem of limiting power consumption and optimizing the use of a shared communication bus will be proposed. These approaches are based on DVS and DPM technologies but they also take into account the communication-media assignment policy. The already mentioned DVS and DPM power saving methodologies are combined in order to exploit their power and maximizing system lifetime. This objective is achieved by complying both the communication requirements and timing constraints, that are typical of a real time distributed systems.

Thesis overview

Chapter 1 introduces real time computing fundamentals. They consist of general definitions, main scheduling algorithms, mathematics frameworks and approximations used in the analysis of real time systems. This part of the thesis is developed according to the notions exposed in [29], but limiting the description only to the aspects strictly necessary to figure out the concepts of the next chapters. The end of the chapter illustrates the basic notions of Real Time Calculus according to [71], that synthesizes some of the relevant results mentioned in this thesis.

Chapter 2 focuses on power management of the embedded systems. In particular it starts with the problem statement, followed by an exhaustive presentation of the power model currently adopted. In conclusion, the chapter ends with the taxonomy of the main families of power management algorithms, and finally the most relevant results have been presented for each family.

Chapter 3 treats in detail the case of large-scale interconnected systems. In particular the chapter starts with the problem of a radio interconnected system, then two novel solutions to reduce power consumption have been proposed for this specific case. Most of the modern approaches to this problem consider power management and communication support separately,

this is not correct since battery-operated and distributed systems must be apply a power saving strategy and guarantee communication support at the same time to accomplish with their final goal. The novel solutions proposed in this chapter [60, 72] provide two algorithms to cope with the problem of saving energy and guarantee communication capability under the constraint of a assigned bandwidth chunk.

Chapter 4 investigates the problem of power management for multiprocessors systems sharing a communication bus. The chapter introduces an adaptive strategy to better assign communication bandwidth among processors, furthermore a novel strategy to cope with the problem of managing overload has been proposed [23]. It consists of a framework operating at two levels: software and hardware. At software level a Real Time kernel is in charge of managing workload by applying elastic scheduling, while at hardware level a sound dynamic assignment of communication bandwidth among processors over a shared bus is performed. The proposed approach demonstrates that such a strategy improves system performances both in term of quality of control and power saving.

Chapter 5 concludes the thesis by summarizing the obtained results presented in this work in chapter three and four. The conclusions concerning each problems have been synthetically pointed out in order to highlight the advantages and the effectiveness of the described solutions.

Chapter 1

Introduction

Real Time computing is becoming crucial in the modern world since most of the human activities are strongly dependent on computers. More and more often in our everyday life, we get in touch with these devices. Some applications where the use of real time systems is needful are avionics, automotive, transports, defense, robotics, biomedical, telecommunication, aerospace, air-traffic control, security or biomedical.

As just said, the importance of such a systems is of paramount importance and we expect it will increase in the future, even because they are recently adopted in much more familiar scenarios: entertainment systems, simulators, virtual reality, consumer electronics or home automation.

Real Time systems are often characterized by limited resources and much more often they operate in dangerous environments, where human intervention is dangerous and therefore technical maintainability are uncomfortable or even impossible. In these scenarios, system design and implementation represent a key features. One of the most important resource featuring real time systems is battery. Since a wide set of such a systems are battery driven, lifetime represents one of the most crucial point to be taken into account. According with this requirement, during the design phase, a specif project methodologies have to be considered in order to cope with this issue. A system able to face this problem provides power management capabilities.

Among all families of embedded and real time systems there exists a particular class that recently is becoming very pervasive, they are the networked embedded systems. The spread of these systems is increased fast, and a lot of them are becoming more and more familiar in our everyday life. Some examples of such a systems are WSN (Wireless Sensor Network) used

in intelligent building applications, traffic monitoring or surveillance. Another example is CPS (Cyber Physical Systems), mostly used in aerospace, civil infrastructures, chemical or transportation; for this kind of systems the combination, synchronization and interconnection among computing units and physical infrastructures is very tight. A subset of these systems can be found in other industrial domains like Automotive and Aerospace where their guidelines are respectively formalized in AUTOSAR (Automotive Open System Architecture) [68] and IMA (Integrated modular avionics) [39] standardizations.
An important point to be highlighted is that the interconnected systems do not scale only to the high level to compose complex systems or geographically distributed systems, but they can be even found in miniaturized systems like MPSoCs (Multi-Processor Systems on Chip), where each computing units provide power-saving support but the interconnection over a shared communication bus has to be guaranteed whatever the cost, in order to keep the system fully operating.
These systems have to guarantee communication link during data transmission or reception, therefore power consumption has to be reduced as mush as possible in order to extend the whole system lifetime.
This thesis focus on this kind of systems, in particular power management aspects will be considered in detail. The current chapter presents the most important results gained in the domain of Real Time and Embedded systems, and the notion exposed in the current chapter, have been introduced with the only intent to introduce the basic notion of Real Time Systems, in order to provide to the reader all the fundamentals needed to deal with this work.

1.1 Real-Time Systems

Real Time systems are computing systems that must to react within precise time constraints to events in the environment [29]. This means that the correctness of the result has not to be only computing correct, but even provided before a given time instant, otherwise it has not to be considered useful or still reliable. This last consideration depends on the fact that a delayed result can be useless, or for particular applications it may result in a catastrophic outcome.

Most of modern computing applications rely with control, these systems play a crucial role since they gather informations from the environment and determine the behavior of the systems in relationship with the information provided by the environment itself. Some examples of this class of applica-

tions requiring real time computing are:

- Chemical and nuclear plants;
- supply chain and industrial process;
- railway signaling;
- automotive;
- air traffic control and avionics systems;
- robotics;
- industrial and home automation;
- defense;
- space exploring;
- multimedia and entertainment;
- virtual reality and simulators.

These technologies are very common since long time, but despite they play an important role in our society, there is a sort of misconception about real time systems and most of the application having real time requirements are developed with ad hoc techniques or heuristic approaches. In fact, very often applications with real time constraints are developed using optimized assembly code, programming a lot of interrupts, manipulating interrupt priority or enabling or disabling tasks preemption in a multitasking scenario. Of course, code with this particular features can be very fast and performing but there are also a lot of disadvantages:

- **Tedious programming**. The code produced according with this strategy is very time consuming and often the result may depend on the software developer ability.
- **Difficult code understanding**. Often the only people able to figure out such a code is only the developer who wrote that code.
- **Maintainability**. more the complexity and the dimension of the application increases, more the effort to maintain code increase with it.

- **Validation of time constraints**. Without specific software tools for this kind of testing, timing constraints checking becomes impossible as well as the validation of the written code.

The consequences of this strategy is that the software empirically produced is error prone and the predictability of the system is seriously compromised. If the time constraints cannot be verified a priori and if the operating system does not provide features to handle stringent real time requirements, the systems is no more reliable and if it apparently seems to work properly during the test phase, this does not prevent from a failure in case of rare but possible scenarios.
The collapse of such a systems may have very catastrophic consequences, for instance people may be injured or serious damages can be caused to the surrounding environment.

Testing is fundamental for real time applications but it does not prevent from faults. Since most of real time applications are control based, the flow of control depends on input parameters, and often is not possible to exhaustively replicate all possible input scenarios and environment conditions. This does not means that testing is not necessary, but it has to be considered as the system response concerning with a subset of input parameters.

According with these concepts a higher level of robustness has to characterize these systems. This objective can be only achieved by adopting advanced design methodologies able to guarantee to the systems a better response even in presence of the most pessimistic scenario. This approach includes static analysis of code, off-line guarantee techniques and providing specific mechanisms at the operating system level, able to support computing even in presence of stringent timing constraints.

1.2 Main definitions and modeling

Real-Time systems represent an object of study since long time, and several methodologies have been introduced in the past to cope with the problem of improving the predictability of such a systems, as well as their performances. Predictability and performances can be considered being among the most important aspects featuring Real Time Systems because timing constraints and efficiency requirements are crucial in modern applications. In order to fugue out the concepts proposed in the next chapters, an introduction of basic aspects of this discipline is required.

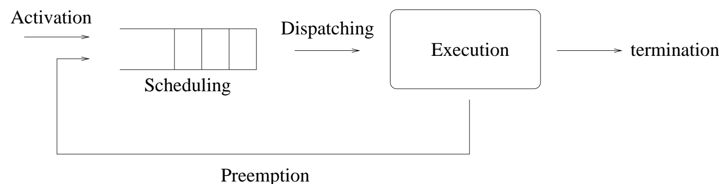

Figure 1.1: Queue of ready tasks

One of the basic point of this domain is the concept of process. Concerning with the definition of *process*, it is necessary to explain a common misconception about its meaning. The most relevant software entity treated by an operating system is the process; *process* can be considered as a computation activity performed by the CPU[1] in a sequential order. In this thesis the words *process* and *task* (or thread) are used as synonyms. However, it is worth to point up that some authors prefer to distinguish among task (or thread) as a sequential execution of code that does not suspend itself during execution, and process as a more complex software entity composed by many tasks.

The strategy according to which a single CPU is assigned to a particular task is called *scheduling algorithm* or *scheduling policy*. The scheduling algorithm (or scheduling policy) represents a set of rules that determines which is the task that have to be executed at any time. The particular operation of allocating a CPU to a single task is called *dispatching*.

Without delving into any particular implementation details, the status of a task with respect to the operating systems can be recognized to be: *active*, *ready* or *running*. According with the Figure 1.1, a task that can potentially executes on a CPU is called active. A task waiting for a CPU availability is called ready task, ready tasks are usually kept in a queue called ready queue. The task currently running on a CPU is called *running task*, a CPU can run only one task a time.

Lets consider a task set $J = \{J_1, \ldots\ldots, J_n\}$, a schedule of J can be considered as an assignment of tasks belonging to the task set J to the processor so that each tasks executes until its completion.

A more formal definition can be provided considering a *schedule* as a

[1]CPU: Central Processing Unit

function σ, so that:

$$\sigma : R^{+} \rightarrow N \mid \forall t \in R^{+}, \quad \exists t_1, t_2 \quad t \in [t_1, t_2) \wedge \forall t' \in [t_1, t_2) \sigma(t) = \sigma(t') \quad (1.1)$$

In other words, a schedule is an integer step function. Therefore $\sigma(t)=k$, with $k > 0$, means that task J_k is running at time t, while $\sigma(t) = 0$ means that CPU is idle.

1.3 Classification of scheduling algorithms

Real time systems literature consists of several scheduling algorithm and their variations, but it is possible to identify some main classes.

- **Preemptive**. By adopting a preemptive scheduling algorithm, a task can be interrupted at any time in order to assign the CPU to a different task, according with a given policy.
- **Non-preemptive**. By adopting this scheduling policy a task, once started, cannot be stopped until its completion, any other decision is taken after this instant;
- **Fixed**. According with this strategy, scheduling parameters are assign at task activation and they will never change at run-time;
- **Dynamic**. According with this strategy, scheduling parameters can be recomputed at run-time;
- **Off-line**. In this case scheduling algorithm run off-line over the whole task-set. The obtained solution is stored in a proper data structure and subsequently used as is;
- **On-line**. Scheduling decision taken at run-time. Whenever a new task is activated or an existing task terminates the execution, a new scheduling decision is taken at run-time;
- **Optimal**. An algorithm is said to be optimal if it minimizes a given cost function;
- **Heuristic**. An algorithm is said to be heuristic it searches for a feasible scheduling by using a objective function (*Heuristic function*);

Another fundamental concept concerning the classification of an algorithm in this domain is represented by the clairvoyance. An algorithm is said to be *clairvoyant* if it knows in advance the arrival time of all tasks. This kind of algorithm is obviously inexistent, but is used with the sole purpose of comparing the existing algorithms with best possible one.

1.4 Periodic task scheduling

Real-time control systems are mostly characterized by periodic activities, this features is strictly related with the nature of such a systems. In fact most of them are control systems and therefore their functioning depends on sensors feedback, low level activities, monitoring and so on. Each task is cyclically triggered at a given sample rate, and it has to perform its activity concurrently with other tasks. In this context, the role of the operating systems is crucial because all of these tasks have individual timing requirements and they have to execute within their deadlines.

This chapter focus on the problem of scheduling periodic tasks, and the main basic algorithm developed to cope with this specific issues will be treated in details. They are: rate monotonic, earliest deadline first, deadline monotonic and finally the EDF version to treat tasks with deadline less then periods. For each algorithm will be introduced basic concepts, schedulability analysis and guarantee test.

In order to make the read easy, the basic concepts will be pointed out and a set of hypothesis will be assumed to simplify the scenario without loss of generality. As a consequence of the foregoing mentioned, the following notations will be introduced:

Γ denotes a set of periodic tasks;

τ_i denotes a generic periodic ith task;

$\tau_{i,j}$ denotes the jth instance of task τ_i;

$r_{i,j}$ denotes the release time of the jth instance of task τ_i;

Φ_i denotes the *phase* of task τ_i. It also reppresents the release time of the forst instance of the task ($\Phi = r_{i,j}$);

D_i denotes relative deadline of task τ_i;

$d_{i,j}$ denotes the absolute deadline of the jth instance of task τ_i. It is also given by $d_{i,j} = \Phi_i + (j-1)T_i + D_i$;

$s_{i,j}$ denotes the start time of the jth instance of task τ_i. It represents when task start running;

$f_{i,j}$ denotes the finishing time of task jth instance of task τ_i. It represents when task stop running.

In order to simplify the analysis, the following hipothesys and assumptions will be considered:

A1. the instances of a periodic task are activated at constant rate. The interval T_i between two consecutive activations represents the *period* of task;

A2. all instances of a periodic task τ_i have the same worst case execution time (WCET) C_i;

A3. all instances of a task has the same relative deadline D_i. All the deadlines are assumed to be equal to the periods T_i;

A4. there not exist any precedence constraints among the tasks belonging to the task-set Γ;

A5. each task cannot suspend itselfe (i.e.: for I/O operations);

A6. task-set Γ is fully-preemptive;

A7. the system overhead is negligible.

According with the notation introduced above, a task-set can be summarized as follow:

$$\Gamma = \{\tau(\Phi_i, T_i, C_i), \quad i = 1, ..., n\}; \tag{1.2}$$

while arrival times $r_{i,j}$ and relative deadline $d_{i,j}$ of the generic kth instance of the ith task can be easily computed as:

$$r_{i,k} = \Phi_i + (k-1)T_i; \tag{1.3}$$

$$d_{i,k} = r_{i,k} + T_i = \Phi_i + KT_i. \tag{1.4}$$

Another set of parameters that characterized the scheduling of a periodic task-set are:

- **Response Time**. It is the time at which the task instance terminates. It is measured starting from the release time:

$$R_{i,k} = f_{i,k} + r_{i,k}; \tag{1.5}$$

1.4.1 Utilization factor

Given a teask-set Γ, composed by n tasks, the *utilization factor* is the fraction of time spent by the CPU to execute the task-set. It is formally defined by

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{1.6}$$

where $\frac{C_i}{T_i}$ denotes the fraction of time spent by the CPU for the execution of task τ_i [58]. The utilization factor can be improved by modifying C_i and T_i but there exist a maximum value of U that, in case of overcame, it uields a not schedulable task-set. This particular value depends on the features of the task-set and the adopted scheduling policy. Whe can denote with $U_{ub}(\Gamma, A)$ the upper bound of the utlization factor for a given task-set Γ and scheduling algorithm A. Under the particular condinition in which $U_{ub} = U_{ub}(\Gamma, A)$, the processor is know to be *fully utilized*, according with this status any other incresing of the computation time of even a single task, caused task-set to be not schedulable.
Given an algorithm A and a task-set Γ, let us to introduce the concept of *least upper bound* $U_{lub}(A)$ of the processor utilization factor, as the minimum of the utilization factors considering all possible task-set that fully utilize the processor:

$$U_{lub}(A) = \min_{\Gamma} U_{ub}(\Gamma, A). \tag{1.7}$$

Since $U_{lub}(A)$ is the minimum of all upper bounds, any task-set with a utilization factor less or equal to $U_{lub}(A)$ is certainly schedulable.

Another important results concerning the utilization factor is that a task-set with an utilization factor greater then one, cannot be schedulated by any algorithm:

$$\forall A, \Gamma \mid U(A) > 1 \Rightarrow \Gamma \, is \, not \, schedulable \, by \, A \tag{1.8}$$

1.4.2 Rate Monotonic scheduling

Rate Monotonic (RM) is a fixed priority scheduling algorithm that consist of a priority rule assignment based on tasks arrivals rate. Once the priorities are assigned they cannot be modified at run-time. Tasks with an high rate of arriving instants have high priorities, tasks that are characterized by a

low rate of arrival instants have low priorities.
Another feature of RM is that it is intrinsically preemptive, because if a new instance of a task arrives and it has a greater priority, it preempts the task currently executing.

In 1973, Liu and Layland [58] demonstrate that RM is optimal among all fixed priority scheduling algorithms. This means that no other fixed priority algorithms can schedule a given task-set that cannot be scheduled by RM.

	C_i	T_i
τ_1	1	4
τ_2	2	5
τ_3	3	10

Table 1.1: RM Analysis: task set composed by three tasks (case 1)

Let's consider a task-set composed by three tasks, whom main scheduling parameters are summarized in Table 1.1, the arrival time of the first instances of each tasks is at $t = 0$ ($a_i = 0 \quad \forall i = 1, 2, 3$). A typical scheduling performed by RM is depicted in Figure 1.2. At $t = 0$ the highest priority task is τ_1 because it has the shortest period, hence it execute first. It terminates at $t = 1$, instant in which τ_2 can start its execution till it terminates at $t = 3$. At $t = 3$, the lowest priority task τ_3 can start its execution till $t = 4$ where it has been preempted by the highest priority task τ_1. After that tasks τ_1 and τ_2 terminates respectively at $t = 5$ and $t = 7$, task τ_3 resumes, but it is preempted one more time at $t = 8$ when a new instance of task τ_1 arrived, it resumes at $t = 9$ and terminates at $t = 10$ that corresponds with its period. Notice that RM guarantees the high priority task to be executed without delay while, from this point of view, the most penalized task is the one with the lowest priority that experiences a major number of preemptions. This feature is typical of RM.

Under rate Monotonic, the U_{lub} value calculated for an arbitary number N of tasks composing the task set is

$$U_{lub} = n(2^{1/n} - 1) \tag{1.9}$$

This value decrese with n and for high value of n, the least upper bound converges to

$$U_{lub} = ln2 \simeq 0.69 \tag{1.10}$$

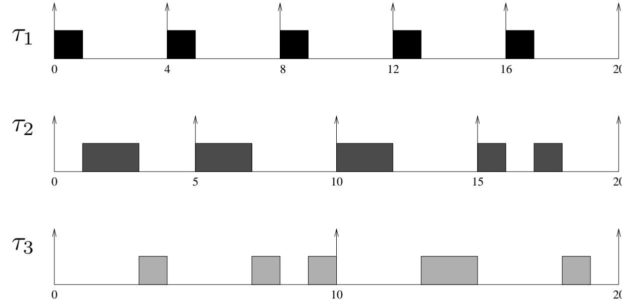

Figure 1.2: Rate Monotonic scheduling

To check for the schedulability of a given task set under RM, the following condition has to be verified

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} < U_{lub} = n(2^{1/n} - 1) \tag{1.11}$$

An alternative schedulability test called Hyperbolic Bound (HB) [19] [20], less pessimistic but characterzied by the same compexity of the original Liu and Layland bound is provieded as a sufficient condition for the schedulabiliy of a task set under RM.

Theorem 1 *Let $\Gamma = \tau_1, ..., \tau_n$ be a set of n periodic tasks, where each task τ_i is characterized by a processor utilization U_i. Then, Γ is schedulable with RM algorithm if*

$$\prod_{i=1}^{n} (U_i + 1) \leq 2. \tag{1.12}$$

As previously mentioned the hyperbolic bound provides a less pessimistic test to check for the schedulability of a task set under RM. In order to appreciate this feature of the HB test, let's consider the tesk set in Table 1.2.

	C_i	T_i
τ_1	1	3
τ_2	1	5
τ_3	2	8

Table 1.2: RM Analysis: task set composed by three tasks (case 2)

According with the formula reported in the equation 1.9 the U_{lub} for that task set is 0.78, while the utilization factor of the task set according with the equation 1.6 is $U = 1/3 + 1/5 + 2/8 = 0.7833$. But 0.783 is slightly greater then 0.78, that is the U_{lub} computed by 1.9, hence the task set should be considered not schedulable under RM.
If we compute the HB test in 1.12 with the same task set we obtain $(1/3 + 1) \times (1/5+1) \times (2/8+1) = 2$ that complies with the HB condition, therefore the task set is schedulable under RM.

Notice that these conditions are only sufficient to prove the schedulabity of periodic tasks set under RM. Therefore if the condition is verified the task set is schedulable, if it fails we cannot conclude anything about the feasibility of the given task set.

1.4.3 Earliest deadline First

The Earliest deadline First (EDF) algorithm belongs to the family of dynamic priority algorithms. In fact the priority assignment rule is based on the absolute deadlines position in time. The rule consists of assigning the highest priority to the task with the earliest absolute deadline at current time.
The formula used to compute the absolute deadline of the jth job of ith task is

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i \tag{1.13}$$

Acording with the formula 1.13 the priority is dynamic at task level but, once it is assigned at job level, it is kept fixed till the job completetion.

An important result concerning EDF is that it is otpimal in the sense of feasibility as demonstrated by Dertouzos in [34]. This means that if a feasible solution for a given task set Γ exists, then EDF is able to find it.

Let's consider the same task set of Table 1.1, the scheduling of such a task set under EDF is shown in Figure 1.3.
At $t = 0$ the highest priority task is τ_1 because it has the earliest absolute deadline at $t = 4$, hence it execute first. It terminates at $t = 1$, instant in which τ_2 can start its execution till it terminates at $t = 3$. At $t = 3$, the task τ_3 starts its executing because it is the only active task in the system, hence it can continues its execution till $t = 4$ where it has been preempted by the highest priority task τ_1, in fact at $t = 4$ the deadline of τ_1 is at $t = 8$, while the deadline of task τ_3 is at $t = 10$.

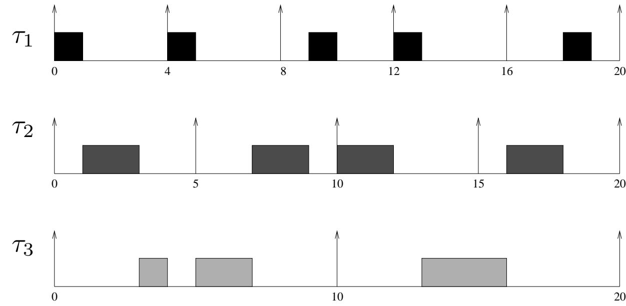

Figure 1.3: Earliest Deadline First scheduling

After that tasks τ_1 terminates at $t = 5$, a new job of τ_2 is activated and it has the next deadline at $t = 10$ like τ_3. Since task τ_3 is already suspended, it starts first, terminates the execution at $t = 7$, then τ_2 can start running till $t = 9$. Notice that at time $t = 8$ a new instance of τ_1 arrived but it cannot be executed in place of τ_2 since its deadline is at $t = 12$ while the deadline of τ_2 is at $t = 10$.

Under the EDF scheduling U_{lub} is equal to one. This means that our task set can exploits the processor up to 100%, still guaranteeing the schedulability of the tasks. This result formalized in the following theorem [58, 76] that provides a necessary and sufficient condition for the schedulability of a set of periodic tasks under EDF.

Theorem 2 *A set of periodic tasks is schedulable with EDF if and only if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{1.14}$$

	C_i	D_i	T_i
τ_1	1	4	6
τ_2	2	6	8
τ_3	3	5	10

Table 1.3: task set with deadline less the period

1.4.4 EDF with deadline less or equal to the periods

In case of EDF with deadlines less or equal to the periods, the analysis to guarantee the schedulability of the task set can be made by using the method proposed by Baruah et al. in [15] and also used by [49]. This approach is called *processor demand criterion* and it is based on the concept of *demand function*.

Definition 1 *The demand function for a task τ_i is a function defined in the interval [t_1,t_2], that represents the computation time that has to be executed for τ_i in [t_1,t_2] to make τ_i itselfe schedulable.*

$$df_i(t_1, t_2) = \sum_{a_i \geq t_1, d_i \leq t_2} C_i. \tag{1.15}$$

This is true for a single task, for the whole task set we have:

$$df_i(t_1, t_2) = \sum_{i=1}^{n} df_i(t_1, t_2). \tag{1.16}$$

For instance, let's consider the task set in table 1.3

The demand function in the interval [7,22] is:

$df(7, 22) = 2 \times C_1 + 2 \times C_2 + 1 \times C_3 = 9;$

This computation example can be easily understood by looking at the Figure 1.4.

Theorem 3 *A task set is schedulable under EDF if and only if:*

$$\forall t_1, t_2 \quad (t_1 < t_2) \quad df(t_1, t_2) < t_2 - t_1. \tag{1.17}$$

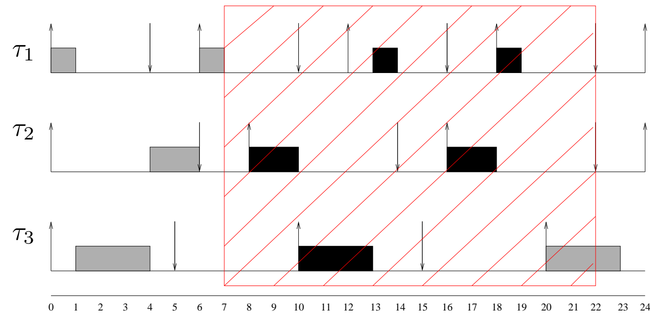

Figure 1.4: demand function interval [7,22]

For the previous example $t_2 - t_1 = 15$, therefore the condition $df(7, 22) = 9 < 15$ guarantees the schedulability of the task set in Table 1.3 but only for the inteval $[7, 22]$. In order to check the schedulability of the task set we have to verify the condition for a huge of intervals. This is unsustainable, hence a new test, characterized by a limited complexity, has to be found.

1.5 Demand Bound Function (DBF)

The Demand Bound Function (dbf), introduced by Baruah et al. in [16], and then used also by Jeffay and Stone [49] to handle the interrupt costs under EDF, represents the total amount of computation that must be executed in each interval of time when tasks are scheduled by EDF. For a generic periodic task τ_i activated at time $t = 0$, its $dbf_i(t)$ in any interval $[0, t]$, it is computed as

$$dbf_i(t) = \max\left\{0, \left(\left\lfloor \frac{T_i - D_i}{T_i} + 1 \right\rfloor C_i\right)\right\}. \qquad (1.18)$$

For the whole task set Γ composed by n tasks simultaneously activeted at $t = 0$, the dbf function is computed as

$$dbf_\Gamma(t) = \sum_{i=1}^{n} dbf_i(t). \qquad (1.19)$$

The processor demand for a set of n tasks in the interval [0,L] can be computed as

$$dbf(L) = \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \le L. \qquad (1.20)$$

Finally, a theorem to prove the schedulability of a give task set under EDF, by limiting the number of intervals in which computing the test.

Theorem 4 *A set if synchronous periodic tasks, with U ¡ 1, is schedulable by EDF if and only if:*

$$\forall L \le L^* \quad dbf(L) \le L \qquad (1.21)$$

where

$$L^* = \frac{U}{1 - U} \sum_{i=1}^{n} (T_i - D_i) \qquad (1.22)$$

In order to figure out the computation of the dbf in real context a simple example is proposed. Let's consider the previous task set in Table 1.3. For

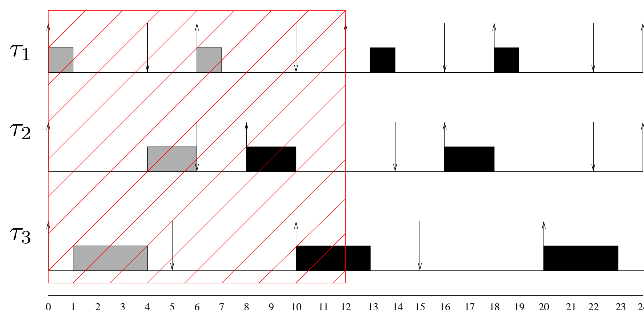

Figure 1.5: dbf and interesting deadlines in [0,12]

that case $U = 1/6 + 1/4 + 3/10 = 0.7167$, while $L = 12.64$. This means that we have to extend the analysis for all the deadlines till L^*. The set of interesting points (i.e deadlines) is $\{4, 5, 6, 10\}$ as shown in Figure 1.5

Therefore the values of the dbf computed in all intersting intervals are:

- $df(0, 4) = C_1 = 1 < 4$;
- $df(0, 5) = C_1 + C_3 = 4 < 5$;
- $df(0, 6) = C_1 + C_2 + C_3 = 6 \leq 6$;
- $df(0, 10) = 2C_1 + C_2 + C_3 = 7 < 10$;

Since for all $L < L^*$ we have that $dbf(L) \leq L$, the task set is schedulable under EDF.

1.6 Supply Bound Function (SBF)

For a periodic resource model (Θ,Π) where Π is a period ($\Pi > 0$) and Θ is a periodic allocation time ($0 < \Theta \leq \Pi$) as defined in [citazione!!!!]. A resource capacity U_Γ of a periodic resource $\Gamma(\Pi, \Theta)$ is defined as $\frac{\Theta}{\Pi}$. The periodic model (Θ,Π) has the following property:

$$supply_\Gamma = (k\Pi, (k+1)\Pi) = \Theta, \quad where \quad k = 0, 1, 2, ... \tag{1.23}$$

According with the periodic model Γ the supply bound function $sbf(t)$ in the time interval t is defined as:

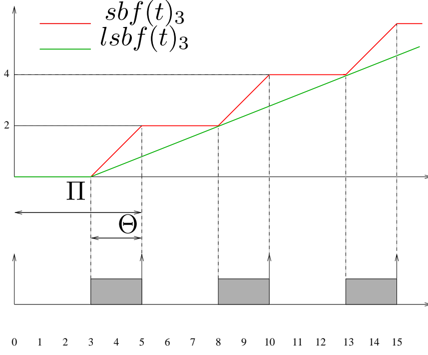

Figure 1.6: $sbf(t)$ and its linearization $lsbf(t)$

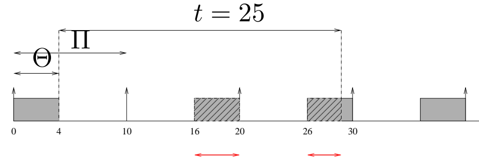

Figure 1.7: $sbf(t)$ for $t = 25$

$$sbf_\Gamma(t) = \begin{cases} t - (k+1)(\Pi - \Theta) & if t \in [(k+1)\Pi - 2\Theta, (k+1)\Pi - \Theta], \\ (k-1)\Theta & otherwise \end{cases}$$

where:

$$k = \max((t(\Pi\Theta))/\Pi, 1) \tag{1.24}$$

The sbf function can be also provided in a linear fashion by defining the $lsbf(t)$ (linear supply bound function) that represents a lower bound for the $sbf(t)$.

$$sbf_\Gamma(t) = \begin{cases} \frac{\Theta}{\Pi}(t - 2(\Pi - \Theta)) & if \quad t \geq 2(\Pi - \Theta), \\ 0 & otherwise \end{cases}$$

The functions $sbf(t)$ and $lsbf(t)$ are represented in Figure 1.6 for $\Gamma(5, 2)$ respectively in red and green.

Let's consider a further example to facilitate the comprension of the sbf function. For instance, $\Pi = 10$, $\Theta = 4$, and $t = 25$. In this case and according with 1.24 $k = 2$, the interval of t is $[22, 26]$. This means that $t \in [22, 26]$, in fact $t = 25$. The $sbf(25)$ for this example is equal to 7 as pointed out in Figure 1.7.

1.6.1 Schedulability analysis

According with the function defined in the previous chapter, under EDF the schedulability of a task set Γ within a reservation R is guaranteed if and only if:

$$\forall t dbf(t) \leq sbf(t) \tag{1.25}$$

For the case of fixed priority (FP) scheduling, a task set Γ is schedulable within a reservation R if and only if:

$$\forall i \quad \exists t \quad \in P_i \quad w(t) \leq sbf_R(t) \tag{1.26}$$

where P_i represents the set of points where the schedulability conditions has to be verified.

The 1.25 and 1.26 are necessary and sufficient conditions. By renouncing this feature, it is possible to derive two necessary condition by using the linearization of the sbf function.

Proposition 1 *(EDF schedulability). The schedulability of a task set Γ under EDF within a reservation R can be guaranteed if*

$$\forall t \quad dbf_\Gamma(t) \leq lsbf_R(t) \tag{1.27}$$

Proposition 2 *(FP schedulability). The schedulability of a task set Γ under FP within a reservation R can be guaranteed if*

$$\forall i \quad \exists t \quad \in P_i \quad w(t) \leq lsbf_R(t) \tag{1.28}$$

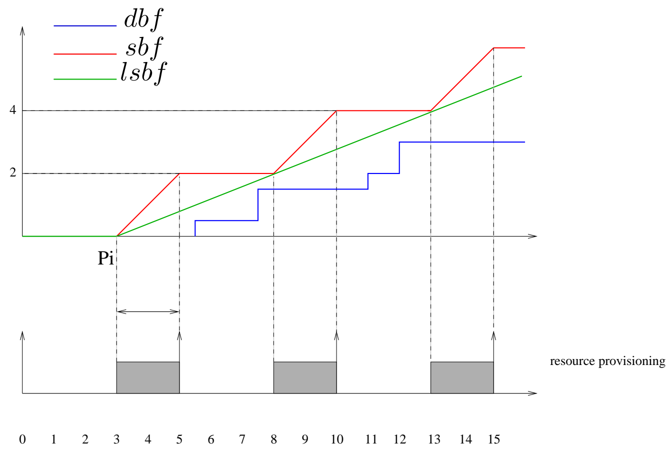

Figure 1.8: $sbf(t)$ for $t = 25$

In Figure 1.8 it is possible to appreciate the *demand bound function* $dbf(t)$, the *supply bound function* $sbf(t)$ and the *linear supply bound function* $lsbf$.

Chapter 2

Power management in Real Time Systems

Battery-operated systems have experienced a huge proliferation due to a pervasive presence of electronic devices in today's technology. In such systems, it is crucial that applications are developed to optimize resource usage, reduce the power consumption and increase lifetime. Moreover, in real-time embedded applications, an increasing number of systems could improve the overall performance by leveraging the advantages of power management. However, in time critical embedded systems, reducing the energy consumption may create overload conditions that can jeopardize the schedulability of the task set. Hence, the issue of reducing energy consumption must be considered as well as the timing constraints. In addition, many energy-aware scheduling methods introduce high variability in task execution times or even timing anomalies, hence they are not suitable for safety-critical systems. Finally, most of the system models proposed in the literature for carrying out the feasibility analysis are not very accurate for describing both the CPU power consumption and the task execution behavior. Such a simplified view of the system certainly allows achieving interesting theoretically results, but, at the same time, limits the applicability of the results, because most of the effects of power management are neglected and the proposed approaches are unrealizable with respect to the technologies provided by the existing hardware architectures. The Real Time community is extremely sensible to these open issues and more and more often these requirements are becoming mandatory in recent real applications, so that they have been addressed according with different strategies as described in the recent Predator Project [1] deliverable [26].

[1] Predator: Predator is a three-year focused-research project within the European Commission's 7th Framework Programme on Research, Technological Development and

The aim of this chapter is to collect the most significant approaches proposed in literature for managing power in embedded real-time systems, with the objective of reducing energy consumption without jeopardizing system predictability. In particular, a realistic power model is presented to evaluate the effectiveness of different approaches and a taxonomy is proposed to classify the most significant approaches available in the literature.

The notions exposed in the current chapter are necessary to introduce basic concepts, power models and the state of the art concerning power management of Real Time Systems. The reading of this chapter represents a preliminary step for the chapter three and four that represent both the real focus of the thesis.

2.1 Existing power models

Power dissipation is going to become an important design issue in a wide range of computer systems in the past decade. Power management with energy efficiency considerations is not only useful for mobile devices for prolonging their duration, but it is also helpful for server systems for the reduction of power bills. Dynamic power consumption due to switching activities and static power consumption due to the leakage current are two major sources of power consumption of a CMOS circuit. For micrometer-scale semiconductor technology, the dynamic power dominates the power consumption of a processor. However, for technology in the deep sub-micron (DSM) domain, the leakage power consumption is comparable to or even more than the dynamic power dissipation. Figure 2.1 shows the trend of the different power consumption components [50].

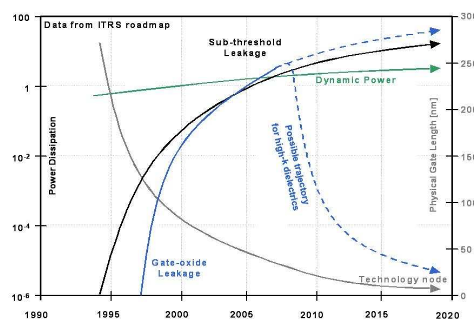

Figure 2.1: CMOS power consumption trend

Demonstration - http://www.predator-project.eu/.

The power consumption of a CMOS system is due to three components:

$$P = P_D + P_{SC} + P_{LK}. \tag{2.1}$$

where each single component represents:

P_D is the dynamic power, that is the power needed to load and unload the output capacitors of the gates. It is independent of the transistor size, but it depends on switching activities as in Figure 2.2;

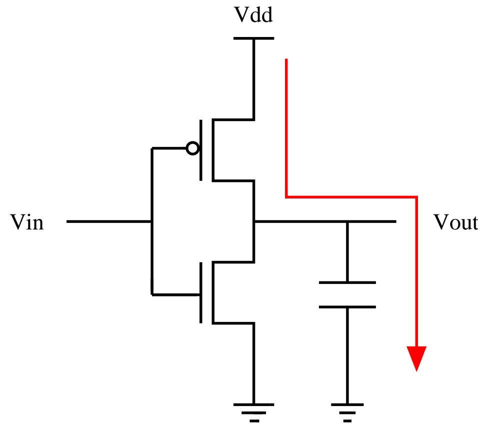

Figure 2.2: CMOS dynamic power consumption

P_{SC} is the power consumption during the gate switching, because in that precise moment, the power source is linked with the ground. Finite slope of the input signal causes a direct current flow from V_{DD} to GND for a short period of time during switching when both the NMOS and PMOS transistors are conducting, this scenario is represented in Figure 2.3;

P_{LK} is the leakage power that depends by the currents generated for physical reasons and it increases exponentially with the temperature. In Figure 2.4 both sub-threshold current (in red) and drain junction leakage (in blue).

The power consumption formula for each component is given by:

$$P_D = C_L \cdot V_{DD}^2 \cdot f; \tag{2.2}$$

$$P_{CS} = t_{SC} \cdot V_{DD} \cdot I_{Peak} \cdot f; \tag{2.3}$$

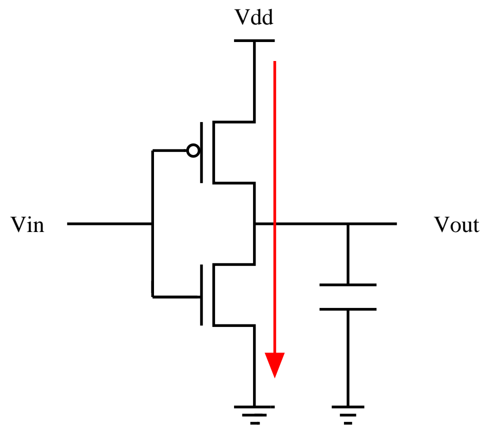

Figure 2.3: CMOS short circuit power consumption

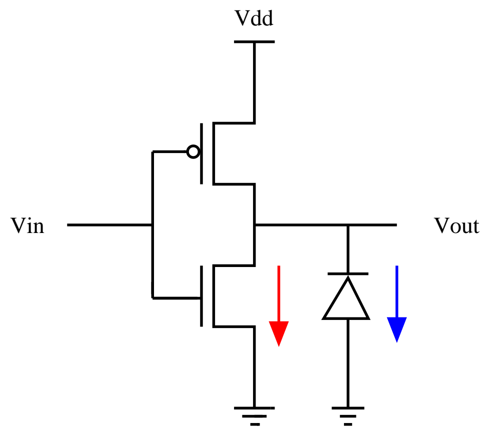

Figure 2.4: CMOS leakage power consumption

$$P_{LK} = V_{DD} \cdot I_{LK}; \tag{2.4}$$

where the list of symbols used is shown below:

V_{DD} is the supply voltage;

f is the clock frequency;

C_L is the equivalent load capacitors;

t_{SC} is the short circuit time during the gate switching;

I_{Peak} is the short circuit current during the gate switching;

I_{LK} is the leakage current;

Hence the complete formula commonly adopted is:

$$P = C_L \cdot V_{DD}^2 \cdot f + t_{SC} \cdot V_{DD} \cdot I_{Peak} \cdot f + V_{DD} \cdot I_{LK}; \tag{2.5}$$

However, reducing voltage naturally affects frequency. The highest frequency f_{max} under which a processor can operate is determined by the critical path τ (the longest path a signal can travel) of its circuitry, which is given by:

$$\tau = \frac{V_{DD}}{(V_{DD} - V_T)^2}; \tag{2.6}$$

where V_T is the threshold voltage and V_{DD} the input gate voltage. Hence, $f_{max} = 1/\tau$. The processor will cease to function if the voltage is lowered such that the propagation delay induced by it is too large for the desired operating frequency. Lowering supply voltage thus demands lower operating frequencies. However, the overall energy consumption of the system also depends on other components. Martin et al. [Mar01] derived the following model to describe the power consumption as a function of the processor speed, defined as a normalized frequency ($s = f/f_max$):

$$P(S) = K_3 \cdot S^3 + K_2 \cdot S^2 + K_1 \cdot S + K_0; \tag{2.7}$$

where K_3 is a coefficient related to the consumption of those components that vary both voltage and frequency; K_1 is a coefficient related to the hardware components that can only vary the clock frequency, whereas K_0 represents the power consumed by the components that are not affected by the processor speed. Finally, the second order term K_2 describes the non linearity of DC-DC regulators in the range of the output voltage.

When dynamic power consumption dominates the other components (due to switching and leakage), the K_3 coefficient is greater than the others and, power consumption can be approximates as:

$$P(S) = K_3 \cdot S^3; \tag{2.8}$$

2.2 Power aware scheduling

The different power-aware scheduling algorithms proposed in the literature can be classified according to the taxonomy illustrated in Figure 2.5.

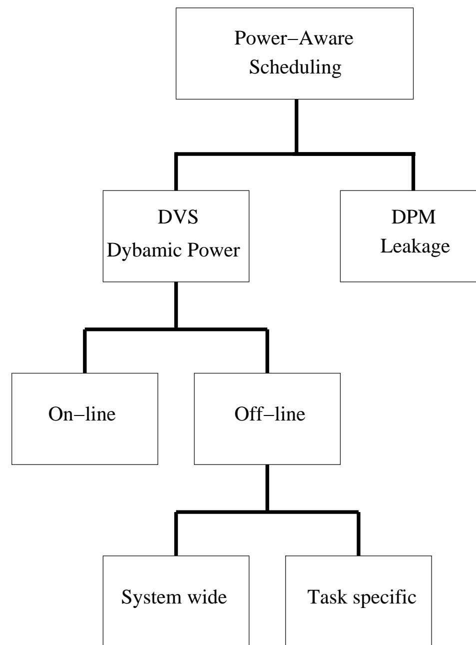

Figure 2.5: Taxonomy of power scheduling techniques

Two most part of power-aware algorithms can be distinguished based on the kind of power they try to reduce. The methods focusing on dynamic power use Dynamic Voltage Scaling (DVS) techniques, while those focusing on leakage-related consumption adopt Dynamic Peripheral Management (DPM) approaches.
Within the DVS family, algorithms can be distinguished between on line and offline. Offline algorithms compute the working speed using static values for parameters, like inter-arrival times and execution times, while on line methods re-compute the parameters during the jobs execution, taking advantage of possible early completions. A final discrimination can be done inside the offline algorithms, splitting them in system-wide and task specific. System-wide algorithms compute a single speed for all the tasks, whereas task specific algorithms may use a different speed for each task.

2.2.1 Dynamic Voltage/Frequency Scaling (DVS)

A widely adopted method to obtain power-aware systems is Dynamic Voltage Scheduling (DVS). In DVS techniques, the processor voltage supply can be reduced to save energy, because the power absorption depends on the third power of the input voltage supply level. However, decreasing the supply voltage also reduces the processor speed, hence proper strategies must be adopted to guarantee the timing constraints of real-time activities.

2.2.1.1 System-wide speed computation

The simplest approach to save power while meeting timing constraints is to reduce the processor speed so that all tasks reach the maximum processor utilization Umax allowed by the adopted scheduling algorithm. When tasks are periodic or sporadic, their schedulability is guaranteed if:

$$U(s) = \sum_{i=1}^{n} \frac{C_i(s)}{T_i} = \sum_{i=1}^{n} \frac{C_i}{s \cdot T_i} = \frac{U_p}{s} \leq U_{max}(A) \tag{2.9}$$

where the symbols in the formula represent:

C_i is the worst-case execution time (WCET) of task τ_i on a processor with unitary speed;

$C_i(s)$ is the WCET of task τ_i at the current speed (note that $C_i(s) = C_i/s$);

T_i is the period of task τ_i (or its minimum inter-arrival time between consecutive jobs);

$U(s)$ is the task set utilization when the processor runs at speed s;

U_p is the task set utilization at when running at unitary speed;

$U_max(A)$ is the maximum task set utilization to guarantee the feasibility under a given scheduling algorithm A.

therefore, the feasibility of a given taskset can be guaranteed if:

$$s^* \geq \frac{U_p}{U_{max}(A)}; \tag{2.10}$$

It follows that the lowest speed that guarantees feasibility and saves the highest energy is:

$$s = \frac{U_p}{U_{max}(A)}; \qquad (2.11)$$

For instance, if tasks are scheduled by Earliest Deadline First (EDF) [58], the utilization bound is Umax(EDF) = 1, hence the lowest processor speed is:

$$s^* = U_p. \qquad (2.12)$$

This approach works only theoretically, because real processors allow only a finite number of frequencies and not a continuous range. Then, a straightforward solution would be to select of the lower allowed frequency greater that the computed one. However, this solution is suboptimal it terms of energy consumption and can be improved by alternating between two frequencies. Such an approach has been investigated in [17], where the optimal speed s^* (when not available) is achieved by switching between the two closest discrete frequencies, in a PWM-mode. Figure 2.6 illustrates the schedules obtained by EDF without power management (a), against the two presented algorithms: the one that computes a single speed assuming a continuous frequency range (b) and the PWM approach for a discrete frequency range (c). The energy E consumed in a hyperperiod (also shown beside each diagram) is computed using the approximated power model, ($P = Ks_3$), assuming each task has the same power cost (K = 1 for all tasks).

2.2.1.2 Tasks with different power costs

Instead of setting a system-wide speed equal for all tasks, some algorithms computes a different speed for each task. The main issue that leads to such an approach is that not all tasks have the same power consumption, since they may use different devices and have different hardware requirements. This approach is adopted by Aydin et al. [AYD01A], who proposed an algorithm for computing the optimal speed for each task in a given set of periodic tasks, aiming at reducing the energy requirements.

Figure 2.7 compares the schedule obtained by EDF on the same task set by using a single speed computation (a) and using different speeds each task (b). In both cases, the energy consumption E is computed considering $K_1 = 1$ and $K_2 = 2$. As clear, power consumption can be further reduced by the algorithm that computes a different speed for each task.

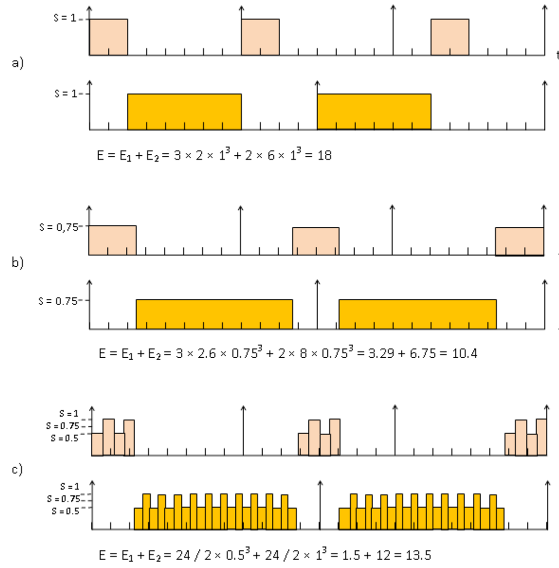

Figure 2.6: EDF: no power manag. (a), single speed (b) and PWM (c).

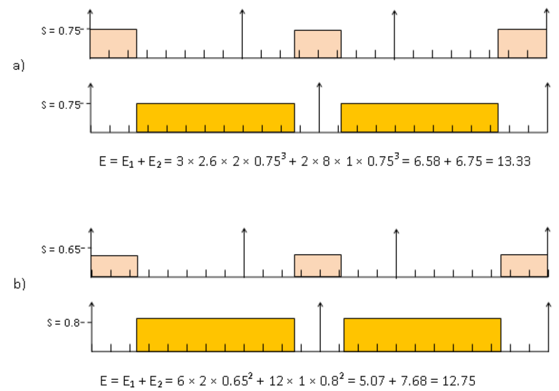

Figure 2.7: system-wide speed selection (a) and per-task speed selection.

2.2.1.3 Variable Computation Time

The solution presented above is optimal if all the jobs of each task always execute for their WCET. In particular, if the number of cycles required by each task is known in advance, it has been shown that using a constant speed during task execution would minimize the energy consumption, assuming a continuous speed processor. In fact, the convexity of the power/speed curve implies that maintaining a constant speed s is always better than switching between two different speeds across s. When the processor offers a limited

set of speeds, using the two speeds which are immediately neighbors to the optimal one minimizes the energy consumption.

However, considering that all tasks run for their WCETs is a very pessimistic assumption. If the computation time is not always equal to the WCET, it is not possible to compute the optimal constant speed because the actual number of cycles required by the current instance is unknown in advance. In this case, typical solutions [13,69,85] are based upon the idea of deferring some work, expecting that the current instance will request much less than its WCET.

The method consists in splitting the task execution into two parts, as shown in Figure 2.8. In the first part, the processor runs at a lower speed to reduce the energy consumed in the average case. In the second part, the processor runs at a higher speed in order to provide enough execution cycles even in the worst case. The idea is that, if a task tends to use much less than its WCET, the second part, which consumes more energy, may never be needed. If the probability density function of tasks execution time is known, Bini and Scordino [18] proposed a way to compute the optimal speeds to reduce power consumption.

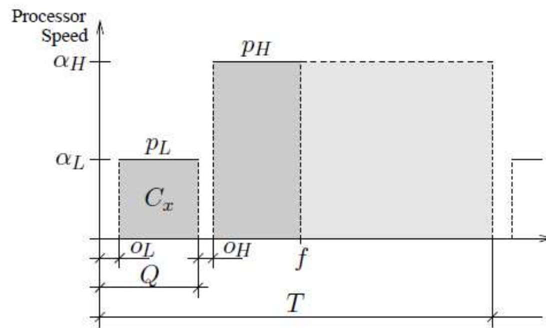

Figure 2.8: Two-speed energy management strategy. consumption.

Figure 2.9 illustrates the advantage of using the two-speed energy management scheme (b) with respect to the single speed solution (a). In this case, tasks execution times are smaller than their WCETs and power costs are considered to be the same for both tasks (K1 = K2 = 1).

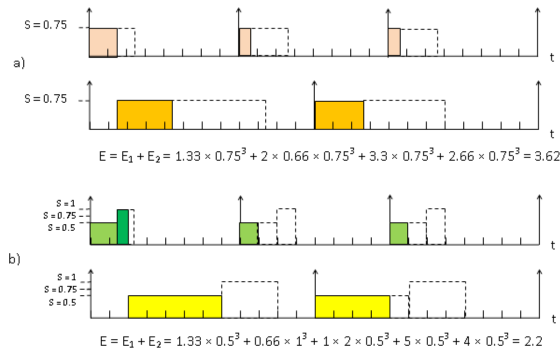

Figure 2.9: Constant speed schema (a) and a two-speed scheme (b).

2.2.1.4 Reclaiming of unused computation time

When a task instance runs for less than its WCET, the unused computation time can be reclaimed to further reduce the speed, so saving more energy. Such a speed reduction can affect either the remaining computations of the same task (intra-task reclaiming), or the execution of the other tasks (inter-task reclaiming). An intra-task reclaiming can be performed as proposed by AbouGhazaleh et al. [8], defining a set of power management points within the task code, so splitting the task into a number of regions. In this work, the authors presented an algorithm to position the power management points in order to minimize the energy consumption with the minimum overhead.

Inter-task reclaiming can be performed by on-line algorithms that use the residual slack at the end of each instance to further reduce the power consumption of the tasks waiting in the ready queue. Aydin et al. [12] proposed two algorithms: the Dynamic Reclaiming Algorithm (DRA) and the Aggressive Speed Adjustment (AGR). The first one attempts to allocate the maximum amount of unused processor time to the highest priority task in a greedy way. The second one aggressively reduces the speed of the running task under certain conditions to a level even lower that the one suggested by DRA. This speculative move might shift the task WCET completion time to a point that requires to increase the speed beyond the optimal one later to guarantee the feasibility of future tasks.

Another possibility to fully exploit inter-task reclaiming with discrete processor frequencies is to change the task periods whenever the application allows a certain degree of flexibility. A method based on elastic scheduling was proposed in [59] for computing the working speed and the tasks periods in order to reduce power consumption.

2.2.2 Dynamic Power Management (DPM)

Power dissipation has constrained the performance boosting of modern computer systems in the past decade. Dynamic power management (DPM) has been implemented in many systems to change the system (or device) state dynamically to reduce the power consumption. DPM techniques explore how to efficiently and effectively reduce the energy consumption to handle event streams with hard real-time or quality of service (QoS) requirements [11], [14], [78], [77]. Intuitively, the device can be switched to the sleep mode to reduce the power consumption when it is idle. Such a switching operation, however, has two concerns. On one hand, the sleep period should be long enough to compensate for the mode-switch overhead. On the other hand, to cope with possible bursts of task arrivals, the reserved time for serving the events must be sufficient to prevent deadline violations of tasks and overflow of the system backlog when activating the device again later on.

Device management has been considered first, adopting the so called event-driven (ED) approach, according to which a device is turned to sleep mode when there is no event in the ready queue, and is awaken for execution when an event comes to the system. A next step in ED systems is using the periodic power management scheme (PPM), where power management is done by statically analyzing the task scheduling streams S. Specifically, the periodic power management schemes first decide the period $T = T_{on} + T_{off}$ for power management, then switch the system to the standby mode for Ton time units, following by Toff time units in the sleep mode. The periodic power management schemes are driven by setting timers to turn the system to (and from) the sleep mode. Examples of such management schemes are presented in [45], where a comparison between periodic methods and event-driven ones is carried out.

The last evolution of DPM methods is represented by predictive algorithms. There exist on-line algorithms [44], [46] that predict future task arrival patterns and apply schedulability analysis to the prediction. Specifically, they try to be optimistic to handle tasks only when they really arrive. Such algorithms adaptively predict the next time to perform a mode switching by considering both historical and future task arrivals, and procrastinate the buffered and future events as late as possible without violating the timing and backlog constraints for the given task streams.

2.2.3 Achieving predictability and Scheduling Anomalies

The advantage of adapting the system speed to reduce consumption inserts an extra degree of freedom that needs to be taken into account in the analysis of the WCETs. Even with the most rigid scheduling algorithm, estimating

the effects of a frequency change on task execution times is not trivial. The first issue to be considered is related with the problem that task execution times do not scale linearly with the processor speed: access times to devices, waiting times for data from memories, and active waits in the code are not directly influenced by the frequency. Also, the access time to data inside the RAM is a function of the bus frequency, which could be different from the CPU frequency, depending on the specific architecture. Switching between frequencies is an action that requires a non negligible overhead in real architectures. Its duration depends on several factors and is a function of the starting and the finishing frequencies. During such intervals the CPU is unable to execute code, so introducing delays that could cause some tasks to miss their deadlines. Also, running at a lower speed increases the execution time of each task, increasing the probability of preemptions. In turn, this can increase the number of cache misses and related reloads, leading to bus accesses with increased execution time and energy cost. Neglecting these effects could lead to unpredictable deadline misses or to a power reduction smaller than expected, even in simple scenarios.
Figure 2.10 illustrates the effects of frequency scaling on two tasks scheduled by EDF. The first diagram show the schedule at full speed, while the second one shows the schedule when setting the speed equal to the task set utilization. In this case, however, the additional preemption increases the execution time due to the extra context switch overhead and Cache Related Preemption Delays (CRPDs) [38, 56, 64], leading to a deadline miss.
Even using DPM techniques, changing the order of task executions affect the cache misses and hence the real job execution times.
In the following section, a partially preemptive scheduling approach is proposed to limit the effects of the overhead introduced by speed variations and improve system predictability.

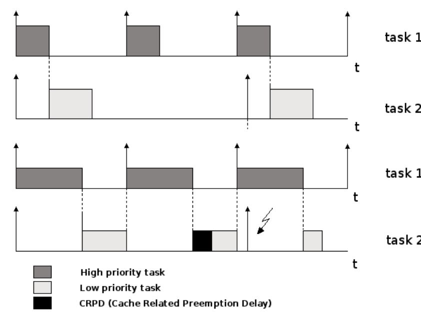

Figure 2.10: Effect of the extra preemption overhead caused by DVS.

This section presents some potential problems that may arise in a voltage-

controlled real-time system, identifying those mechanisms and situations that should be avoided when applying speed variations to time critical systems. Such problems prevent controlling the performance of a real-time application as a function of the processor speed, since a task could even increase its response time when executed at a higher speed. Typically, such scheduling anomalies arise when tasks share mutually exclusive resources or are handled by non-preemptive scheduling policies [25].

In the following figures, the processor speed is represented on the y-axis, so the higher the task execution box, the higher its execution speed. Figure 2.11 illustrates a simple example where two tasks, τ_1 and τ_2, share a common resource. Task τ_1 has a higher priority, arrives at time t = 2 and has a relative deadline $D_1 = 7$. Task τ_2, having lower priority, arrives at time t = 0 and has a relative deadline $D_2 = 23$. Suppose that, when the tasks are executed at a certain speed S_1 has a computation time $C_1 = 6$, (where 2 units of time are spent in the critical section), whereas τ_2 has a computation time $C_2 = 16$ (where 12 units of time are spent in the critical section). As shown in the first case of Figure 2.11, τ_1 arrives just before τ_2 enters its critical section, it is able to complete before its deadline, without experiencing any blocking. However, if the same task set is executed at a double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in the second case of Figure 2.11. This happens because, when τ_1 arrives, τ_2 already granted its resource, causing an extra blocking in the execution of τ_1, due to mutual exclusion.

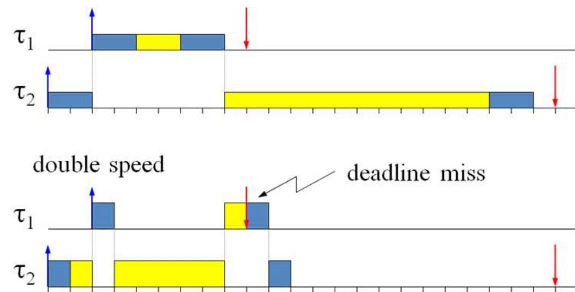

Figure 2.11: Scheduling anomaly in the presence of resource constraints.

Figure 2.12 illustrates another anomalous behavior occurring in a set of three real-time tasks, τ_1 τ_2 and τ_3, running in a non-preemptive fashion. Tasks are assigned a fixed priority proportional to their relative deadline, thus τ_1 is the task with the highest priority and τ_3 is the task with the lowest priority. As shown in the first case of Figure 2.12, when tasks are executed at speed S_1, τ_1 has a computation time $C_1 = 2$ and completes at time $t = 6$.

However, if the same task set is executed with double speed $S_2 = 2S_1$, τ_1 misses its deadline, as clearly illustrated in the second case of Figure 2.12. This happens because, when τ_1 arrives, τ_3 already started its execution and cannot be preempted (due to the non-preemptive mode).

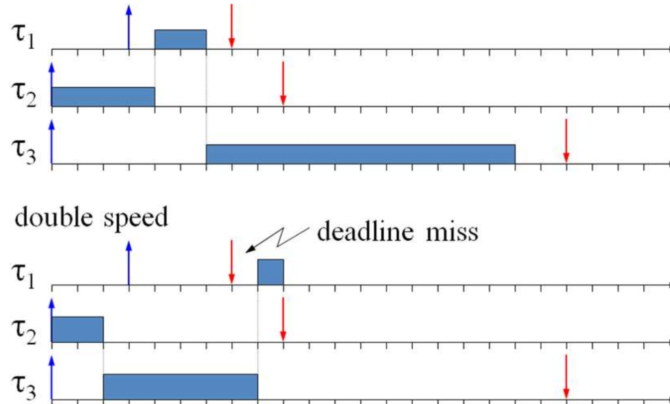

Figure 2.12: Scheduling anomaly in the presence of non-preemptive tasks.

It is worth observing that a set of non preemptive tasks can be considered as a special case of a set of tasks sharing a single resource (the processor) for their entire execution. According to this view, each task executes as if it were inside a big critical section with a length equal to the task computation time. Once a task starts executing, it behaves as it were locking a common semaphore, thus preventing all the other tasks from taking the processor.

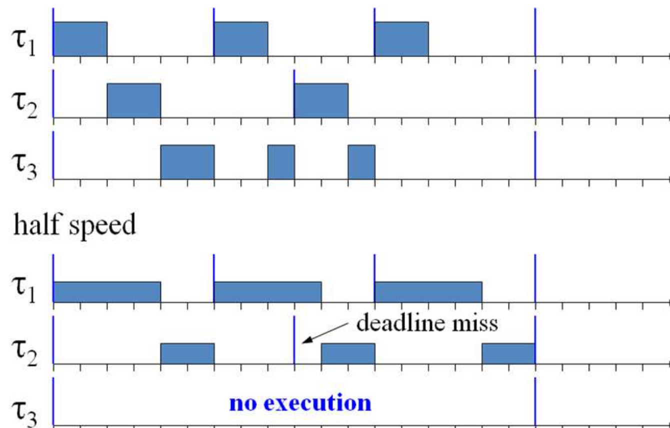

Figure 2.13: Effects of a permanent overload due to a speed reduction.

The example illustrated in Figure 2.13 shows the negative effects of a overload condition, caused by a speed reduction, in a set of periodic tasks.

The first case of Figure 2.13 shows the feasible schedule produced by the Rate Monotonic (RM) algorithm [58] when the processor runs at a given speed S_1, where the tasks have computation times $C_1 = 2$, $C_2 = 2$, and $C_3 = 4$, respectively.

The second case of Figure 2.13 shows the schedule obtained by RM when the processor speed is reduced by half, $S_2 = S_1/2$, so that all computation times are doubled. In this case, a speed reduction generates a permanent overload that causes τ_2 to miss its deadline and prevents τ_3 to execute. Note that a scheduler based on absolute deadlines, as Earliest Deadline First (EDF) [58], would not prevent τ_3 to execute and would react to overloads by delaying all tasks executions more evenly. A exhaustive comparison between RM and EDF for different scenarios can be found in [24].

Chapter 3

Power management in large-scale interconnected systems

Embedded systems cover a wide spectrum of application domains, such as consumer electronics, biomedical systems, surveillance, industrial automation, automotive, and avionics systems. In particular, the technology evolution of sensor and networking devices paved the way for plenty of new applications involving distributed computing systems, many of them deployed in wireless environments and exploiting the mobility and the ubiquity of components. In most cases, devices are battery operated, making energy-aware algorithms of paramount importance to prolong the system lifetime.

In each node of the system, at the processor level, two main mechanisms can be exploited to save energy: the Dynamic Voltage and Frequency Scaling (DVFS) and the Dynamic Power Management (DPM).

For DVFS processors, a higher supply voltage generally leads to both a higher execution speed/frequency and to a higher power consumption. On the other hand, DPM techniques are used to switch the processor off during long idle intervals, hence they tend to postpone tasks execution as long as possible still preserving the schedulability of the task set. At the network level, the energy consumption due to communication is usually managed by DPM techniques, although other mechanisms have been proposed in the literature, as the Dynamic Modulation Scaling (DMS) [73].

In micrometer CMOS technology, the dynamic power dissipation due to switching activities prevails against the static power dissipation caused by the leakage current. However, in most modern processors developed with sub-micron technology, the static power is comparable or even greater than the dynamic power [50, 51]. When the dynamic power is dominant, DVFS

techniques are used to execute an application at the minimum processor speed that guarantees meeting real-time constraints. Conversely, when static power is dominant, there exists a critical processor speed below which the energy wasted is greater than that consumed at the critical speed [30]. For this reason, some authors recently proposed energy-aware algorithms that combine DVFS and DPM techniques to improve energy saving [35, 84].

In distributed systems there is the need of taking into account processor and network bandwidth to guarantee performance requirements. In particular, in wireless distributed embedded systems, energy consumption and quality of service represent two crucial design objectives. Messages have to be transmitted within a deadline to guarantee the desired quality [54, 66], and the transmission itself represents an energy cost to be minimized. Although a lot of research has been done to reduce power consumption while guaranteeing real-time requirements, most papers focus either on task scheduling or network communication. However, a co-scheduling of task and messages would allow exploring more degrees of freedom and could lead to higher energy saving.

Finally, an effective approach has to be platform independent and easily portable to new hardware just by changing a small set of parameters, such as the energy consumption of the CPU in different working modes.

3.1 Related works

A lot of research papers presenting DVS algorithms for energy-aware real-time task scheduling have been published in the past years, such as [13, 52, 74]. Conversely, some other papers focuses on DPM techniques, see for instance [48], [47] and the related works therein. DVS scheduling algorithms, e.g., [12, 81, 83], tend to execute events as slowly as possible, without any violation of timing constraints: they trade processor speed with energy consumption. DPM algorithms are used to switch the processor off during long idle intervals [31, 48, 50].

Not many papers in the literature deal with tasks and packets energy-aware co-scheduling. Moreover, they focus on combining CPU DVS techniques for tasks and DPM approaches at the network level. To the best of our knowledge, no one considered DPM techniques for task scheduling.

In [65], the authors addressed energy saving issues for a system composed by a DVS capable CPU and a network interface supporting the DPM. They proposed two DVS based algorithms: one, Limited Look-Ahead EDF (LLE), favors energy saving at the CPU level and the other, Timeout Aware Scheduler (TAS) that favors energy saving at the network level. LLE tries to minimize the average power wasted by all tasks using a modified version of

the LaEDF algorithm [70]. Instead, TAS tries to maximize the sleep time of the network card by gathering the packet transmissions into bursts, exploiting LaEDF. The choice of which algorithm is better to use depends on the task set parameters and on the difference between the CPU and the network device, in terms of power consumption. Hence, the authors propose both off-line and on-line methods to select the best performing algorithm.

Poellabauer et al. [82] proposed an integrated resource management algorithm that considers both CPU and a bandwidth reservation protocol for the network interface, in wireless real-time systems. The aim of the proposed method is to guarantee task and message deadlines while reducing the power consumption. The resource management system is composed by two parts: a Task and Speed Scheduler (TSS) and a Packet Scheduler (PS). TTS is in charge of producing task scheduling and DVS selection. Instead, PS is in charge of producing packets queuing and delivering to the network interface. The input parameter of TTS is the next time-slot available for packet transmissions, which is provided by the PS. TSS uses a DVS scheduling technique, named Network Aware EDF (naEDF), based on LaEDF. The PS is based on a modified work-conserving EDF algorithm. The authors evaluate the performance of the algorithms by simulation. Moreover, the effectiveness of the algorithm is shown through a real implementation for the Linux kernel.

Sudha et al. [55] presented two slack allocation algorithms for energy saving based on both DVS and DMS techniques. The authors consider a single-hop wireless real-time embedded system, where each task node is composed by precedence constrained message passing sub-tasks. Furthermore, sub-tasks and messages are considered non-preemptable. Energy consumptions for both computation and communication are analyzed by a new metric, called normalized energy gain. The authors proposed two algorithms: the Gain based Static Scheduling (GSS) and the Distributed Slack Propagation (DSP). While the former is used off-line and computes the slack considering worst-case execution for each schedule entity (sub-task or message), the latter is used on-line to exploit the additional slack, available when tasks execute for less than the predicted worst-case computations. Notice that, while GSS is a centralized policy that consider all task and messages of the system, DSP is a distributed policy, independently executed at each node. This allows reducing both time overhead and energy waste due to message passing for global dynamic slack allocation. In this way, a dynamic slack generated in a node is only utilized for local tasks and messages.

3.1.1 Problem description

We consider a distributed real-time embedded system consisting of a set of wireless nodes. Each node executes a set of independent tasks that need

to exchange information with tasks running in other nodes. A node is modeled as a component $c = (\Gamma, S, M, B)$ that takes as input a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, a scheduling algorithm S, a message set $M = \{s_1, \ldots s_m\}$ and a transmission bandwidth B.

Tasks are scheduled by the node processor according to the given scheduling policy S, while messages are transmitted during the intervals in which the bandwidth B is made available by the adopted protocol. Notice that a node is not required to work during the remaining intervals, so it can be turned off to save energy.

The proposed analysis focuses on a bandwidth allocation protocol that provides a slotted bandwidth according to a Time Division Multiple Access (TDMA) scheme. To decouple task execution from the communication activity, all tasks in a node build packets and move them to a shared communication buffer in the processor memory. When the channel is available, packets are transferred from the communication buffer to the transceiver for the actual transmission.

As outputs, each component could provide a set of performance indexes, such as message delays, task response times, and the energy consumption. At the moment, only energy consumption is provided as output. The component interface is schematically illustrated in Figure 3.1.

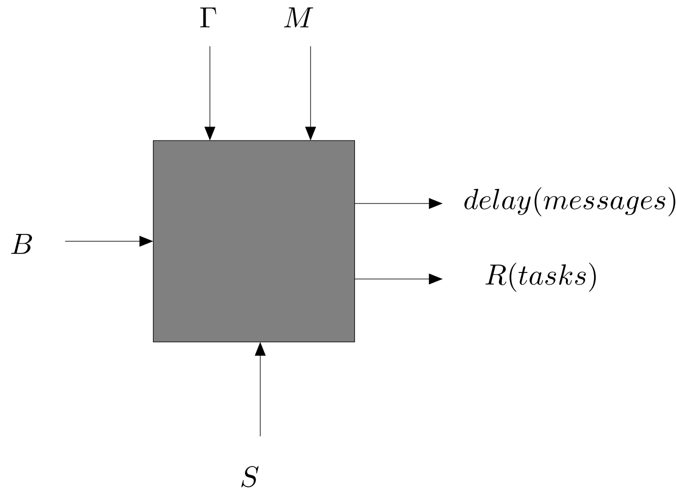

Figure 3.1: Node interface.

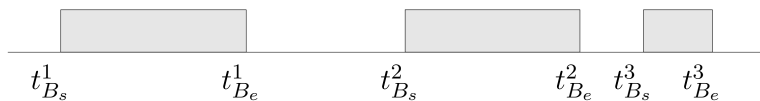

Figure 3.2: Bandwidth assignment.

3.1.2 Workload and Resource Models

An application Γ consists of a set of periodic tasks, where each task $\tau_i = (C_i, T_i, D_i)$ is characterized by a worst-case execution time C_i, a period T_i, and a relative deadline D_i. Each task τ_i produces a message stream $s_i = (m_i, M_i)$ characterized by a payload m_i and a deadline (relative to the task activation) M_i for the message transmission or reception.

In order to decouple the message production from the job execution we suppose that messages are generated at the job deadline. The produced messages are enqueued in a buffer and then transmitted as soon as the bandwidth becomes available. Assuming that packets ready to be transmitted are stored as soon as they are created and that the time for moving them is negligible, then message transmission does not affect task scheduling.

In each node, the computational resource (i.e., the processor) is assumed to be always available at any time t, hence it is modeled as a straight line $f(t) = t$. On the other hand, the communication bandwidth B is assigned by a bandwidth manager (running in a master node) in a slotted fashion. In general, the transmission bandwidth is modeled as set of disjointed slots $B = \{b^1, \dots b^r\}$, where each slot is described by a start time $t^i_{B_s}$ and an end time $t^i_{B_e}$. An example of slotted bandwidth assigned to a node is shown in Figure 3.2.

3.1.3 Power Model

Each node consists of a CPU (processing element) and a Transceiver (transmitting and receiving element). Each device can be in one of the following states:

- *active*. In this state, a device performs its job, executing tasks or handling messages. The power consumed in this state is denoted as P_a.
- *standby*. In this state, the device does not provide any service, but consumes a small amount of power P_s to be ready to become active within a short period of time.
- *sleep*. In this state, the device is completely turned off and consumes the least amount of power P_σ; however, it takes more time to switch to the active state.

For a processor that supports DVS management, the power consumed in active mode depends on the frequency at which the processor can execute. Such a frequency is assumed to vary in a range $[f_{min},\ f_{max}]$, while the processor execution speed s is defined as the normalized frequency $s =$

f/f_{max} and varies within $[s_{min}, s_{max}]$. In particular, for the processor in active mode the power consumption model adopted is derived by Martin et al. [61], which can be expressed as

$$P_a(f) = a_3 f^3 + a_2 f^2 + a_1 f + a_0 \quad (3.1)$$

where

- a_3 is the third order coefficient related to the consumption of the core sub-elements that vary both voltage and frequency;
- a_2 is the second order coefficient describing the non linearities of DC-DC regulators in the range of the output voltage;
- a_1 is the coefficient related to the hardware components that can only vary the clock frequency;
- a_0 represents the power consumed by the components that are not affected by the processor speed (like the leakage).

Switching from two operating modes takes a different amount of time and consumes a different amount of energy which depends on the specific modes, as shown in Figure 3.3. In particular, the following notation is used throughout the paper: $t_{a-\sigma}$ and $E_{a-\sigma}$ are the time and the energy required for active-sleep transition, while the active-standby transition is described by t_{a-s} and E_{a-s}. For all devices we have that $P_\sigma < P_s < P_a$ and $t_{s-a} < t_{\sigma-a}$.

In this paper, we assume also that switching between the standby mode and the active mode has negligible overhead, compared to the other switches, which is the same assumption made by other authors [80, 86].

A simplified power consumption model is adopted for the transceiver to concentrate on the interplay between DVS and DPM for the processor. The communication bandwidth is then considered as a constraint for serving the schedule that minimizes power consumption while guaranteeing a desired level of performance. In particular, a transceiver is assumed to be either in *on* (equivalent to the active state) or *off* (equivalent to the sleep state) mode only (not in *standby*). Whenever the transmission bandwidth is available the transceiver is considered in on mode; the power used to transmit and receive messages is assumed to be equal to P_{on}, that is: $P_{tx} = P_{rx} = P_{on}$. Whenever the transmission bandwidth is not available, the transceiver is assumed to be in off mode with a power consumption equal to P_{off}.

Table 3.1 summarizes all the allowed modes with their characteristics, while Figure 3.3 illustrates the mode transition diagram and the transition costs in terms of time and energy.

	Radio On	Radio Off
CPU Sleep	/	$P_\sigma + P_{off}$
CPU Standby	/	$P_s + P_{off}$
CPU On	$P_a(f) + P_{on}$	$P_a(f) + P_{off}$

Table 3.1: Power model: allowed power modes.

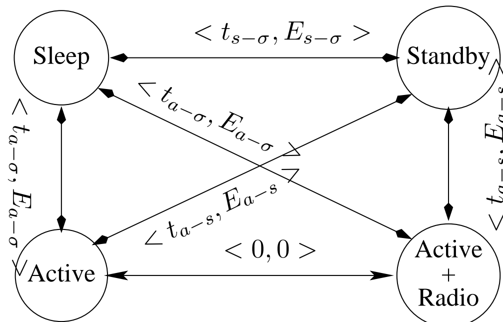

Figure 3.3: Power model: transition costs.

3.1.4 Energy Aware Scheduling (EAS)

Let consider the example shown in Figure 3.4, where three tasks are activated at time 3, 8 and 10 sec (*Activations* on the figure) with deadlines at 40, 50 and 55, respectively. There is also a pre-defined transmission slot in which the system is forced to transmit. Without such a constraint on transmission, tasks would have started as soon as they became active, running at the maximum allowed speed (dashed line). Using only DPM, task executions would be postponed as much as possible up to their deadlines, leading to the continuous line starting at $t = 25$, with slope equal to the maximum allowed execution speed. Notice however, that tasks could start earlier, at time $t = 15$, and run at a reduced speed to better exploit the transmission bandwidth (continuous line starting at $t = 15$sec).

Such an example motivates the need for combining DVS and DPM techniques to select the most appropriate delay and speed that reduces energy consumption, while coping with the available transmission bandwidth and guaranteeing the application timing constraints. The proposed algorithm is referred to as the Energy Aware Scheduler (EAS) [72].

The EAS algorithm is applied at a generic time instant t. First, it computes the interval $[t_{min}, t_{max}]$ for the next activation point t_a that satisfies DPM requirements and timing constraints. Second, the DPM is applied to compute the maximum t_{feas} at which the processor can start executing at its maximum speed s_{max}, keeping the task set schedulable. Third, task

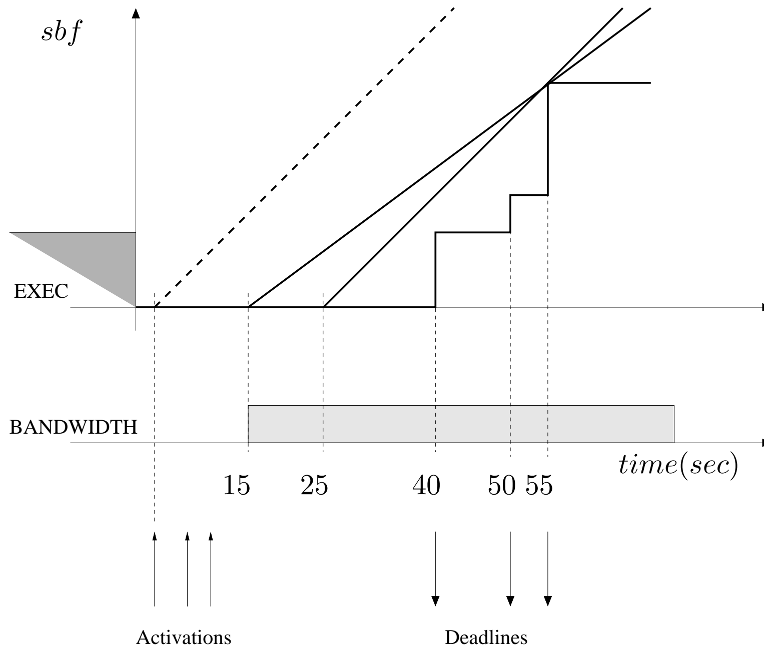

Figure 3.4: DVS and DPM: transmission bandwidth awareness.

executions are postponed as much as possible assuming execution at the maximum speed s_{max}, to approach the starting point t_{B_s} of the transmission bandwidth. The final step is to select the minimum processor speed s needed to keep the task set schedulable. To take message communication into account, the schedule is arranged to overlap with the bandwidth allocated slot. In this way, message transmission corresponds to task execution, allowing saving more energy.

The objective of the EAS algorithm is to compute the activation time t_a that minimizes energy consumption in the next task scheduling and message transmission slot. Any valid activation point t_a must take into account the feasibility bound t_{feas}, the start time of the bandwidth slot t_{B_s}, and the next activation time t_{act}[1]. Such dependencies define the interval $[t_{min}, t_{max}]$ for t_a. The value t_{max} is the minimum between t_{feas} and t_{B_s}. If the processor has no pending jobs, the value of t_{min} is set to the next activation time t_{act}, otherwise $t_{min} = t$. By the definition of the interval $[t_{min}, t_{max}]$, the actual selection of t_a is done by computing the time that minimizes the energy consumption from the current time t to the end of the evaluation period t_F. Algorithm 1 reports the sequence of steps of the EAS algorithm, while Figure 3.5 depicts the EAS application sequence and the result. Assumed

[1] t_{act} is the first task activation time after the actual time t.

task sets feasible under EDF, the feasibility of those task sets scheduled according the EAS is guaranteed by construction because at each step the feasibility is kept. In other words, the EAS applies the available slack to put in sleep the processor and it tries to reduce the CPU execution speed.

Due to the assumptions on the system model, the bandwidth is a constraints only for the energy saving problem and it does not affect the schedulability of the task sets.

Algorithm 1 Energy Aware Scheduling - EAS

```
procedure t | t ∉ B
    Compute the dbf(t);
    Compute sbf^{l*}(t) = sbf^l(t, f_max) and obtain t_feas;
    Calculate t_max = min{t_feas, t_{B_s}};
    if No pending jobs at t then
        t_min = t_act;
    else
        t_min = t;
    end if
    Find t_a ∈ [t_min, t_max] | min_{t_a} E(t_a);
    if t_a ≥ t + 2t_{a−σ} then
        Put the processor in sleep state in [t, t_a];
    else
        Put the processor in standby in [t, t_a];
    end if
    Compute the min frequency f_a or slope s_a guaranteeing feasibility.
end procedure
```

The system energy consumption $E(t_a)$ is computed as

$$E(t_a) = (t_a - t)P_{\sigma/s} + (t_F - t_a)P_a(s(t_a)f_{max}) + \\ +2E_{a-\sigma} + E_{radio}(t, t_F), \quad (3.2)$$

which is done according to the consumption models detailed in the previous section. In particular $E_{radio}(t, t_F)$ is the energy the transceiver consumes in $[t, t_F]$ as a function of the available bandwidth and $P_{\sigma/s}$ is the not-working power consumption, which is equal to P_σ if $t_a - t \geq 2t_{a-\sigma}$, and P_s otherwise.

As already said, the problem to be solved is to find t_a in the interval $[t_{min}, t_{max}]$ that minimizes energy consumption. That is,

$$t_a \mid min_{t_a \in [t_{min}, t_{max}]}\{E(t_a)\}. \quad (3.3)$$

In the next section, such a relationship will be deeply exploited by comparing the energy saving contributions from DVS and DPM.

Figure 3.5: EAS: three possible processor executions speeds.

3.1.4.1 Experimental Results

This section presents some simulation results achieved on the proposed EAS method. An event-driven scheduler and an energy controller simulator has been implemented in C language and interfaced to Gnuplot. The simulator receives a task set and a bandwidth assignment as inputs. The task set is executed with a chosen scheduling policy. The energy consumed to schedule tasks and to transmit messages is computed at each simulation run. A simulation run consists of scheduling one task set with the assigned bandwidth until the task set hyperperiod *hyp*. The power consumption $\frac{E}{hyp}$ in the hyperperiod is then considered.

The scheduling policies applied are:

- *EDF* with no energy considerations, meaning that the processor is assumed always active at the maximum frequency, even if tasks are not ready to execute.

- *pureDVS* on top of an EDF scheduling algorithm. Only speed scaling is applied off-line to guarantee feasibility and the processor speed is set to that value. On-line changes are not allowed.

- *pureDPM* where the task execution is postponed as much as possible and then scheduled by EDF. The execution is at the maximum processor speed.

- DVS and DPM are combined with EDF through the *EAS* algorithm.

A simulator infrastructure automatically generates a stream of tuples (U, n_t, B, n_B), where U denotes the utilization of the generic task set Γ, n_t the number of tasks, B the communication bandwidth (expressed as a percentage of the hyperperiod), and n_B the number of slots in which the bandwidth has been split. Both the task set utilization and the number of tasks are controlled by the task set assignment (U, n_t). The bandwidth assignment (B, n_B) allows to control both the total bandwidth and its distribution within the hyperperiod.

Given the total utilization factor U, individual task utilizations are generated according to a uniform distribution [21]. Each bandwidth slot is set in the hyperperiod with a randomly generated offset. To reduce the bias effect of both random generation procedures, 1000 different experiments are performed for each tuples (U, n_t, B, n_B) and the average is computed among the results obtained at each run.

Two different CPUs have been considered: the Microchip DsPic (DSPIC)[2] and the Texas Instruments (TI)[3], both using the CC2420 transceiver as communication device. Table 3.2 and Table 3.3 report the parameters that characterize the power model of the processors and the transceiver used in these tests, according to the models described in the system model section. Minimum and maximum frequencies of the CPUs are taken from the device data-sheets, whereas the coefficients $[a_0, a_1, a_2, a_3]$ comes form Equation 3.1.

CPU	P_s [mWatt]	f $[f_{min}, f_{max}]$ [Mhz]	$P_a(s)$ $[a_0, a_1, a_2, a_3]$ [mWatt]	P_σ [mWatt]	t_{sw} [sec]
TI	–	$[25, 200]$	$[7.7489, 17.5, 168.0, 0.0]$	0.12	0.00125
DSPIC	9.9	$[10, 40]$	$[25.93, 246.12, 5.6, 0.0]$	1.49	0.020

Table 3.2: Power profiles for processing devices.

Transceiver	P_s [mWatt]	P_a [mWatt]
CC2420	0.066	62.04

Table 3.3: CC2420 Transceiver power profile.

In a first simulation, we tested the power consumption of the CPUs as a function of the activation time. Figure 3.6 shows a general dependency of the power consumption from the model adopted for the processor. The

[2]DSPIC33FJ256MC710 microprocessor
[3]TMS320VC5509 Fixed-Point Digital Signal Processor

figure shows also that both CPUs are DVS sensitive, in the sense that both privilege DVS solutions than the pure DPM ones. Indeed, the DSPIC and the TI exhibit a lower energy at t_{min} than at t_{max} (respectively 160 and 195 in this case as one of the interval of analysis along the whole execution interval). This means that a pure DVS solution costs less than a pure DPM one. Moreover, the DSPIC shows a global minimum inside the interval, meaning that a combined policy is able to reduce energy consumption. The time value corresponding to the minimum is the t_a that has to be found.

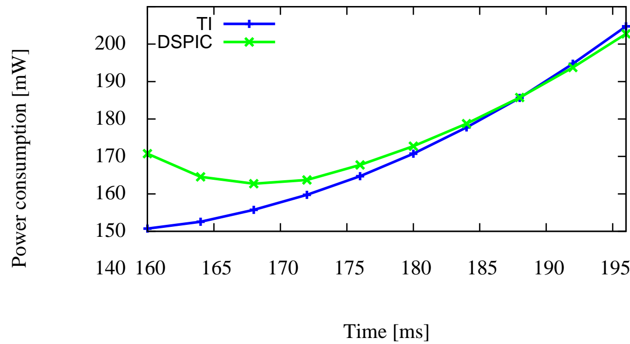

Figure 3.6: Power consumption: TI and DSPIC.

Figure 3.7 compares the two architectures, showing a higher energy consumption for the DSPIC. The power consumption has been averaged to the hyperperiod of each task set. Note that both the CPUs have a dependency on the utilization.

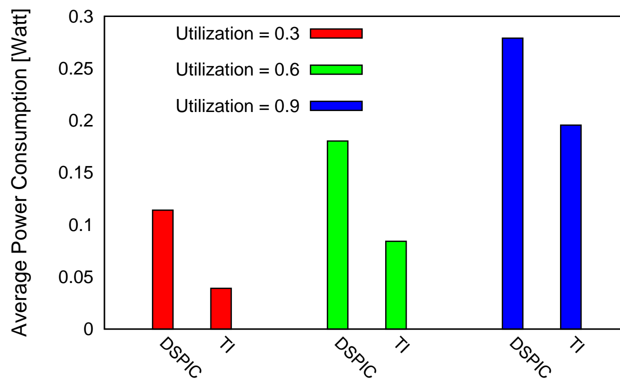

Figure 3.7: CPU comparison: $n_t = 4$, $B = 0.3$ and $n_B = 3$.

We also investigated the effects of the transmission bandwidth to the energy consumption of the system. The results are reported in Figure 3.8,

which illustrates the power consumption as a function of the bandwidth assignment. Note how the dependency is stronger with respect to the bandwidth amount, because the transmission cost increases when there is more bandwidth available; in fact, we assumed the CPU remains active while the bandwidth is available. Moreover we assumed to have messages available to be transmitted, so that the bandwidth is fully used for transmission with an increasing cost when the assigned bandwidth increases. On the other hand, the dependency with respect to the bandwidth allocation slots (how much B is split) is quite weak. This is because message deadlines were assumed to be large enough not to create a scheduling constraint.

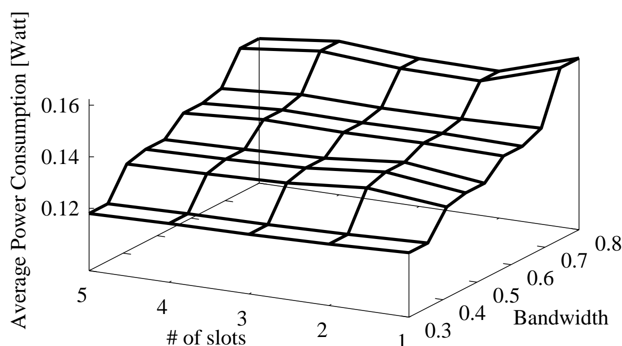

Figure 3.8: Average power consumption: $U = 0.3$, $n_t = 4$.

Figure 3.9 shows how the EAS policy is affected by the task set, in terms of U and n. Notice that the power consumption is significantly affected by the utilization but not much by the number of tasks.

Figure 3.10 compares the EAS policy with respect to the *pureDVS*. The results are quite similar, since the considered CPUs are both sensitive to DVS. Nevertheless, the EAS is able to exploit the DPM capabilities and the available bandwidth to reduce the power consumption in all the task set assignments, mainly when the processor is not heavily loaded (low utilization cases).

Finally, Figure 3.11 and Figure 3.12 compare the four scheduling policies (for TI and DSPIC, respectively), under the same B, n_B, and n_t conditions, but for different task set utilizations. Notice how the EAS policy outperforms the other policies, especially for low utilizations. For high utilization, EAS and *pureDVS* exhibit the same performance (but lower power consumption with respect to the *pureDPM* and EDF). This happens because, for high utilization there is no room for DPM improvements and only DVS is effective. Also note that, for very low utilizations, *pureDPM* provides better results than *pureDVS*. This is due to the fact that $U = 0.1$ would

require a speed lower than the TI minimal speed, hence *pureDVS* forces the CPU to have a speed greater than the utilization, providing more service than required.

To conclude, the EAS algorithm is proved to be effective with respect to the other policies because it looks for the minimum energy consumption in $[t_{min}, t_{max}]$ and in any possible condition. If the minimum is found in t_{min} or t_{max}, the combined method is equivalent to the pure DVS or the pure DPM, respectively. Most of the time, however, the minimum is found inside the $[t_{min}, t_{max}]$ interval, so that the EAS is able to further reduce energy consumption with respect to the pure versions.

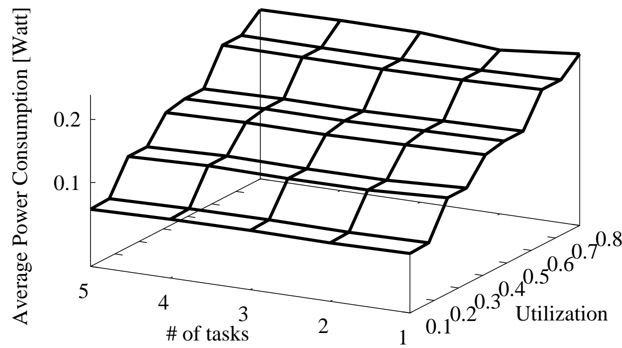

Figure 3.9: EAS with $B = 0.5$ and $n_B = 3$ and the TI processor.

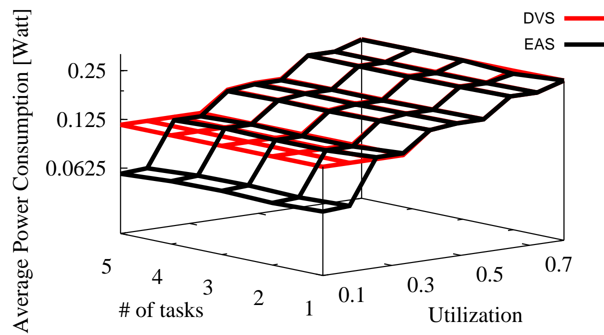

Figure 3.10: Comparison: $B = 0.5$ and $n_B = 3$ and the TI processor.

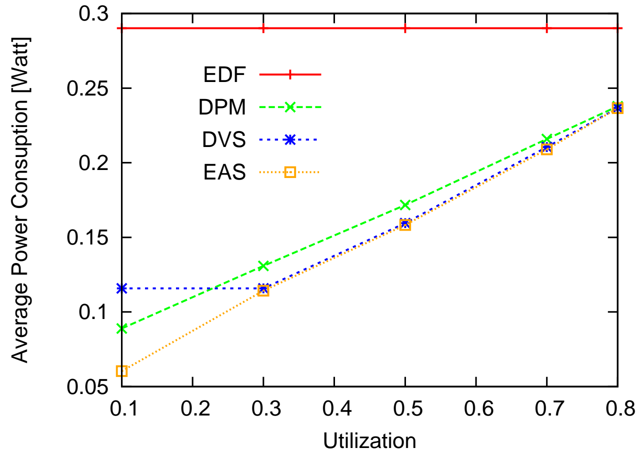

Figure 3.11: Comparison: $B = 0.5$, $n_B = 3$ and $n_t = 4$ (TI processor).

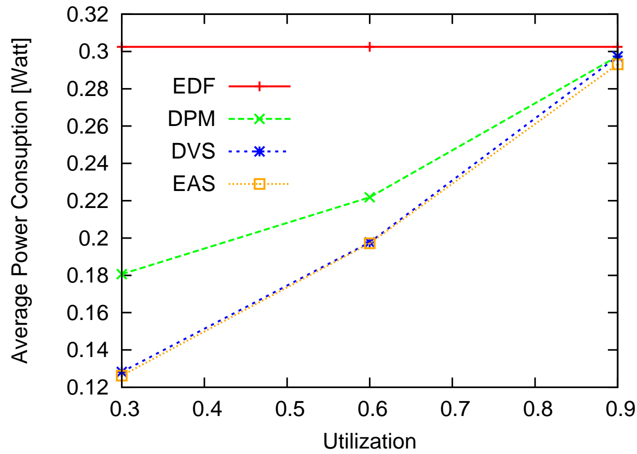

Figure 3.12: Comparison: $B = 0.5$, $n_B = 3$ and $n_t = 4$ (DSPIC processor).

3.1.5 Discrete Energy Aware Scheduling (DEAS)

This section illustrates the second solution namely Discrete Energy Aware Scheduling (DEAS) [60]. The proposed approach mixes at run-time DVFS and DPM techniques to reduce energy consumption while meeting all task deadlines, if there exists a feasible schedule. The combination of DVFS and DPM is done by forcing a CPU sleep interval followed by an active interval executed at a fixed frequency. Such a frequency is selected to minimize the energy (per unit of computation) between the current and the next invocation of the analysis. The j-th instance of the analysis is performed either at

the end of an active interval or at the end of a communication slot. The former instant represents the beginning of an idle period that can be prolonged further by the analysis, while the latter is selected to exploit the slack, if any, collected during the forced activity inside the slot.

The following terminology is used to identify particular timing instants.

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

$$t_{a_j} = \begin{cases} next_act(t), & \text{if no pending jobs at } t; \\ t, & \text{otherwise.} \end{cases} \tag{3.4}$$

  The index j referring to a particular instance will be omitted whenever not necessary.

- t_{w_i} denotes the latest time after t_{a_j} at which the processor can return active with frequency f_i and still guarantee the schedulability of the task set.
- t_{idle_i} denotes the first idle time after t_{w_i} assuming the processor is executing at frequency f_i.
- t_{e_i} denotes the effective time at which the processor can become idle considering the activity constraint inside bandwidth slots. Hence, if t_{idle_i} falls before b^s_k, t_{e_i} is set at t_{idle_i}; otherwise t_{e_i} is forced to occur at the end of the bandwidth slot, that is, $t_{e_i} = b^e_k$.

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

1. For each frequency f_i, the analysis derives the longest inactive interval δ_i exploitable in sleep state from t_a, such that the task set is still feasible when the CPU is turned active at $t_a + \delta_i$. A negative value of δ_i implies that the task set can not be schedulable at that frequency.

   δ_i is determined as the minimum among the inactive intervals computed for each deadline, that is

$$\delta_i(t) = \min_{d_j \in [t, t+L^*(f_i))} \left\{ d_j - \frac{dbf(t, d_j)}{f_i} - t \right\}. \tag{3.5}$$

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

$$t_{w_i} = min\{t_a + \delta_i(t_a), b_k^s\}. \tag{3.6}$$

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

$$\begin{aligned} t_{idle_i}^{s+1}(t_a) = & \sum_{\tau_j \text{ active}} \frac{c_j(t_a)}{f_i} + \\ & \sum_{j \in \Gamma} \left(\left\lfloor \frac{t_{idle_i}^s}{T_j} \right\rfloor - \left\lfloor \frac{t_a}{T_j} \right\rfloor \right) \frac{C_j}{f_i}. \end{aligned} \tag{3.7}$$

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

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

$$t_{e_i} = \begin{cases} t_{idle_i}, & t_{idle_i} < b_k^s; \\ b_k^e, & \text{otherwise.} \end{cases}$$

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

$$E_i(t_a, t_{w_i}, t_{e_i}) = (t_{w_i} - t_a)P_\sigma + (t_{e_i} - t_{w_i})P_{a_i}. \tag{3.8}$$

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

$$EPC_i(t) = \frac{E_i(t_a, t_{w_i}, t_{e_i})}{W_i(t_a, t_{e_i})}. \tag{3.9}$$

   A detailed analysis about the computation of W_i is carried out in section 3.1.6.

5. Among the possible frequencies that guarantee feasibility, the approach selects f^* featuring the minimum EPC_i. t_w^* and t_e^* denote the wake up time and the effective idle time resulting from the selected frequency f^*, respectively.

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

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

Figure 3.13 illustrates an example that clarifies the steps of the proposed approach. In the example, the CPU supports three different frequencies sorted in ascending order: f_1, f_2 and f_3.

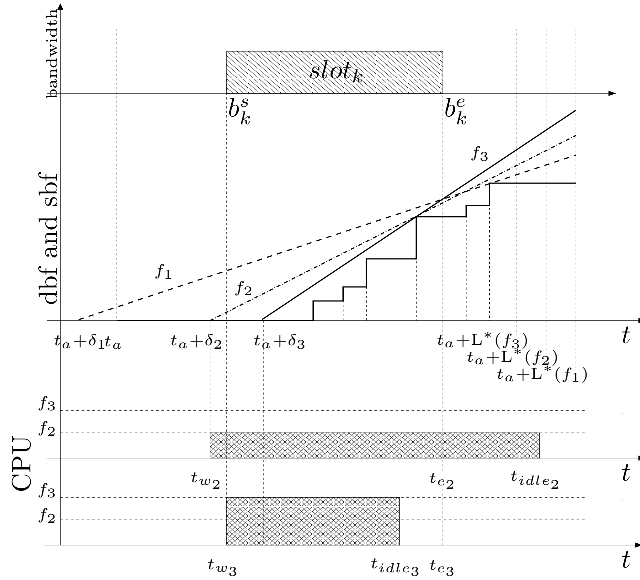

Figure 3.13: Example of the analysis' behaviour.

In the example, frequency f_1 leads to an unfeasible schedule, since δ_1 is negative, whereas f_2 and f_3 produce feasible solutions, since both δ_2 and δ_3 are positive.

Notice that, when considering frequency f_2, $t_a + \delta_2$ falls before b_k^s, hence $t_{w_2} = t_a + \delta_2$, whereas for f_3, t_{w_3} is set equal to the beginning of the slot b_k^s, as $t_a + \delta_3 \geq b_k^s$.

For both frequencies f_2 and f_3, t_{e_2} and t_{e_3} takes the value of b_k^e, as both t_{idle_2} and t_{idle_3} occur after b_k^s.

To choose between the two feasible frequencies (f_2 and f_3), the normalized energy consumption is computed. Such a value is intrinsically derived from the platform power model.

Algorithm 2 Discrete Energy Aware Scheduling - DEAS

procedure (t)

1: **Input**: $t : \forall k, t \notin [b_k^s, b_k^e)$
2: compute t_a according to Equation (3.4);
3: **for all** f_i **do**
4: compute $\delta_i(t_a)$ as in Equation (3.5);
5: **if** $\delta_i(t_a) < 0$ **then**
6: set f_i not feasible and **continue**;
7: **end if**
8: compute t_{w_i} according to Equation (3.6);
9: compute t_{e_i}, W_i as shown in section 3.1.6;
10: compute EPC_i according to Equation (3.9);
11: **end for**
12: compute f^* feasible that minimizes EPC_i;
13: set wake up time at t_w^*;
14: set CPU frequency to f^*;
15: **if** $t_w^* - t \geq t_{a\sigma} + t_{\sigma a}$ **then**
16: put the processor in sleep state;
17: **else**
18: put the processor in standby state;
19: **end if**

Equation 3.4, executed at line 2, has the complexity of an extraction from an ordered list of task activation times; that is, $O(1)$. On the other hand, the insertion complexity is $O(log_2(n))$, where n is the number of tasks. Given n and the maximum number of deadlines a single task can produce in the analysis interval, p, the maximum number of analysis points of the dbf is np. The upper bound of p is computed as the number of occurrences of the task with the shortest period in the analysis interval: $\left\lceil \frac{max_i L^*(f_i)}{min_i\{T_i\}} \right\rceil$. Supposing to arrange the active deadlines in a sorted list, with complexity $O(log_2(n))$ (as the active deadlines are always n) to keep the ordering, the computation of dbf, executed every time the algorithm is invoked, has a total complexity of $O(log_2(n)np)$. The computation of the δ_i, at line 4, involves a complexity $O(np)$. The computation performed at line 9 has a complexity of $O(nq)$, where q is defined as the maximum number of activations a task can generate in $[t_{a_j}, t_{a_j} + max_i L^*(f_i) + max_k\{b_k^e - b_k^s\}]$. The reason is that the algorithm analyzes all the activations, computing the

actual workload and any idle gap, as shown in section 3.1.6. The q upper bound is computed as $\left\lceil \frac{max_i L^*(f_i)+max_k\{b^e_k - b^s_k\}}{min_i\{T_i\}} \right\rceil$. The computation of the EPC_i has complexity $O(1)$. Hence, the for loop, executed at line 3, has a complexity of $O(nF(p+q))$, where F is the total number of available frequencies.

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

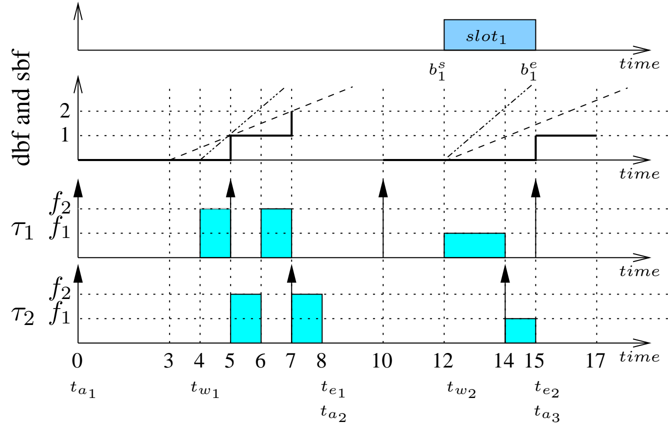

Figure 3.14: DEAS example.

The behavior of the DEAS algorithm is now illustrated using the example in Figure 3.14. The assigned bandwidth is composed by one slot in the interval $[b^s_1, b^e_1]$ equal to [12,15]. The CPU allows 2 frequencies equal to $f_1 = 5$ and $f_2 = 10$, and schedules a task set of 2 synchronous implicit periodic tasks with periods $T_1 = D_1 = 5$ and $T_2 = D_2 = 7$, and worst-case execution cycles $C_1 = 10$ e $C_2 = 10$. For the task set under analysis, we have $L^*(f_1) = 4$ and $L^*(f_2) = 2$. The result of the off-line computation for L_{max} is 7. The power consumptions in the active state are $P_{a_1} = 3$ and $P_{a_2} = 6$, while $P_\sigma = 1$ and $t_{a\sigma} + t_{\sigma a}$ is considered negligible for the sake of simplicity.

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

Executing at frequency f_1, the first idle time t_{idle_1} occurs at $t = 13$ because, from time $t_{w_1} = 3$, the busy period consists of three instances of τ_1 and two instances of τ_2, for a total execution of 10 units of time. Due to the bandwidth activity constraint, t_{e_1} is set to 15. Instead, running at frequency f_2, the next idle time t_{idle_2}, from time $t_{w_2} = 4$, occurs at time $t = 8$ and, since it falls before b^s_1, we have $t_{e_2} = 8$. Once the interval $[t_{w_i}, t_{e_i}]$

is determined, the algorithm computes $EPC_1 = 0.71$ and $EPC_2 = 0.70$, and therefore sets $f^* = f_2$.

Using f_2, the second invocation of the algorithm occurs at $t = 8$, causing the postponement of t_{a_2} to $t = 10$. Running at frequency f_2, the CPU can wake up at 13, but, for the activity constraint, t_{w_2} is set to $b_1^s = 12$. Consequently, $t_{idle_2} = 13$ and $t_{e_2} = b_1^e = 15$. Instead, using frequency f_1, the algorithm obtains $t_{w_1} = 12$, $t_{idle_1} = 18$, and $t_{e_1} = b_1^e = 15$.

In such a scenario, the energy consumptions are $EPC_1 = 0.73$ and $ECP_2 = 1$, and therefore the chosen frequency is $f^* = f_1$.

3.1.6 Workload Computation

This section presents the procedure *compute_te_W*, formally defined in Algorithm 3, that computes the effective idle time t_{e_i} and the effective workload W_i, concepts already introduced in Section 3.1.5. The algorithm, based on the current workload, computes next idle times t_{idle_i} till t_{e_i} is found, taking into account bandwidth constraints.
However, note that the procedure output is composed by t_{e_i} and W_i only. To reduce the DEAS algorithm complexity, such computations are integrated into a single routine. Figure 3.15 shows the effective workload of three key scenarios.

For each case, the effective workload is represented by the sum of the slashed areas. Such a value is expressed as number of machine cycles, so the working frequency must be considered.

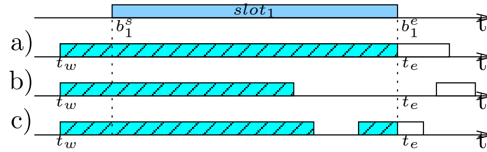

Figure 3.15: Examples of effective workload.

First, to determine t_{idle_i} the iterative approach is initialized as described in Section 3.1.5. Whenever $t^s_{idle_i}$, at a generic step s, crosses the end of the current bandwidth slot $t^{s-1}_{idle_i} < b^e_k \leq t^s_{idle_i}$ (cases a and c), the procedure stops setting $t_{e_i} = b^e_k$ and accounting in W_i the workload between the beginning of the current Busy Period and the end of the bandwidth slot b^e_k. Note that, if the recurrent relation converges ($t^{s-1}_{idle_i} = t^s_{idle_i}$) outside the bandwidth slot, no slot occurs during the analyzed Busy Period, hence $t_{e_i} = t^s_{idle_i}$ and W_i

is increased by $(t^s_{idle_i} - t_{w_i})f_i$. If $t^s_{idle_i}$ converges inside the bandwidth slot (cases b and c), W_i is incremented by $(t^s_{idle_i} - t_{w_i})f_i$.

Algorithm 3 Procedure to compute t_{e_i} and W_i

procedure *compute_te_W*

1: **input**: f_i, t_{w_i}
2: **output**: t_{e_i}, W_i
3: $W_i = 0$; $t_{start} = t_{w_i}$;
4: **loop**
5: $\quad t^0_{idle_i} = t_{start} + \sum_{\tau_j \text{ active}} \frac{c_j(t_{start})}{f_i}$;
6: $\quad$ **do**
7: $\quad\quad$ compute $t^s_{idle_i}$ according to Equation (3.7);
8: $\quad\quad$ **if** $t^{s-1}_{idle_i} < b^e_k \le t^s_{idle_i}$ **then**
9: $\quad\quad\quad t_{e_i} = b^e_k$;
10: $\quad\quad\quad W_i \mathrel{+}= (t_{e_i} - t_{start})f_i$;
11: $\quad\quad\quad$ **return**;
12: $\quad\quad$ **end if**
13: $\quad$ **while** $t^{s-1}_{idle_i} \neq t^s_{idle_i}$;
14: $\quad W_i \mathrel{+}= (t^s_{idle_i} - t_{start})f_i$;
15: $\quad$ **if** $t^s_{idle_i} \notin [b^s_k, b^e_k]$ **then**
16: $\quad\quad t_{e_i} = t^s_{idle_i}$;
17: $\quad\quad$ **return**;
18: $\quad$ **else**
19: $\quad\quad$ **if** $next_act(t^s_{idle_i}) \ge b^e_k$ **then**
20: $\quad\quad\quad t_{e_i} = b^e_k$;
21: $\quad\quad\quad$ **return**;
22: $\quad\quad$ **end if**
23: $\quad\quad t_{start} = next_act(t^s_{idle_i})$;
24: $\quad$ **end if**
25: **end loop**

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

3.1.6.1 Experimental Results

This section presents a set of experimental results that show the effectiveness of our approach with respect to other classical solutions. The results are obtained by simulation using a synthetic workload under three power

consumption profiles derived from real platforms according to the power consumption model described in power model section.

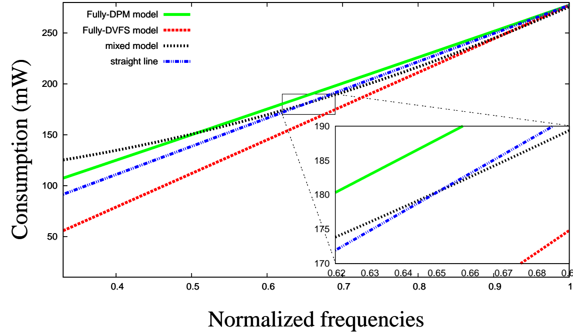

Figure 3.16: Speed-Power characterization for different platforms.

To test the behavior of our algorithm, three CPU consumption profiles (*power models*), shown in Figure 3.16, are introduced. Each profile is described by the power consumption coefficients $[a_3, a_2, a_1, a_0]$:

- *Fully-DPM* $[0.0, 5.6, 246.12, 25.93]$;
- *Fully-DVFS* $[0.0, 0.0, 330.62, -53.32]$;
- *Mixed* $[0.0, 150.55, 24.5, 100.78]$.

The fully-DPM model has been extracted from the Microchip dsPIC [4] datasheet, interpolating the typical consumptions. The other two models have been synthetically derived from the first one to achieve different but comparable behaviors, at the same time, with respect to the original. Fully-DPM and fully-DVFS represent opposite cases: in a fully-DVFS scenario, halving the speed (doubling the execution time) always implies a reduction of the energy consumption, while in fully-DPM cases, the consumption increases. A model is defined as DPM or DVFS according to its position with respect to the straight line.
Such a line represents a theoretical situation in which slowing down has the same energy consumption of executing at a different speed. The mixed model has a threshold frequency f_{th} at speed 0.65 meaning that its behavior is DVFS-like above f_{th} and DPM-like below.

The frequency range of the CPU used in the simulation is $[12.5, 40]$ MHz. The sleep state consumption P_σ is 1.49 mW and the wake up time takes

[4]DSPIC33FJ256MC710 microcontroller

about 20 ms. The standby state has a higher consumption P_s of 9.9 mW, but a shorter wake up time within 8 cycles. As for the fully-DPM model consumptions, such values were extracted from the dsPIC datasheet. All the simulations have been executed using a set of 8 evenly distributed frequencies.

For comparison purposes, four scheduling policies have been implemented in the simulator:

- *EDF* with no energy considerations, where the processor is assumed always active at the maximum frequency, even during idle intervals.
- *pureDVFS* on top of EDF, where the CPU runs with the minimal speed, computed off-line, that guarantees feasibility according to the task set. The actual speed is the lowest frequency greater than the minimal one.
- *pureDPM*, where, as soon as there is an idle time and no assigned bandwidth, the task execution is postponed as much as possible and then scheduled by EDF at the maximum speed.
- *DEAS*, the algorithm introduced in this paper.

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

Given the total utilization factor U, individual task utilizations are generated according to a uniform distribution [21].

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

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

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

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

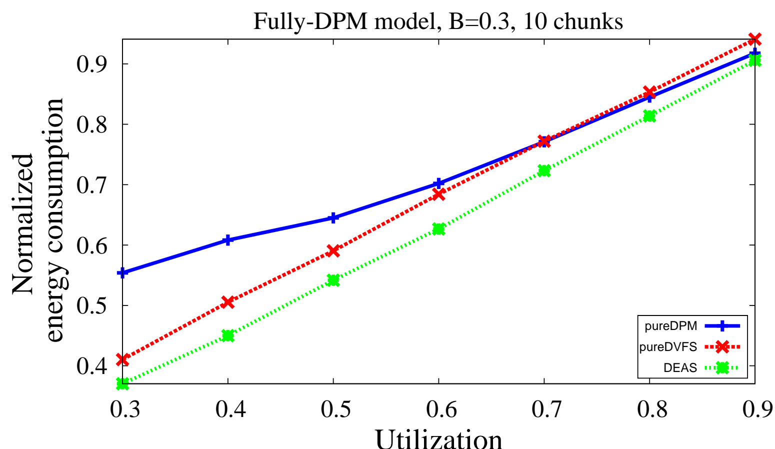

Figure 3.17: Consumptions with a fully-DPM power model.

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

As shown in Figure 3.17, due to the activity constraint posed by the bandwidth slots, DEAS outperforms pureDPM even in fully-DPM power models.

Instead, Figure 3.18 shows that in fully-DVFS power models such a constraint has no effect on pureDVFS: keeping the system active represents the default behavior and, with respect to the analyzed power model, the best solution. For this reason, DEAS and pureDVFS have similar performances.

Under a fully-DPM and in a mixed context, Figure 3.17 and Figure 3.19 shows that pureDVFS acts better than pureDPM for low U values, because the CPU can not be switched off inside bandwidth slots. Instead, for higher utilization values, the consumptions are similar.

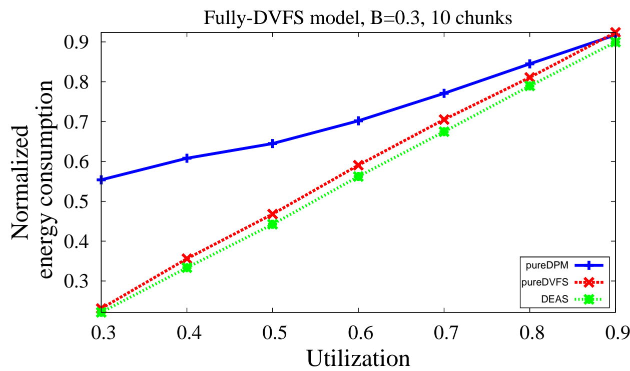

Figure 3.18: Consumptions with a fully-DVFS power model.

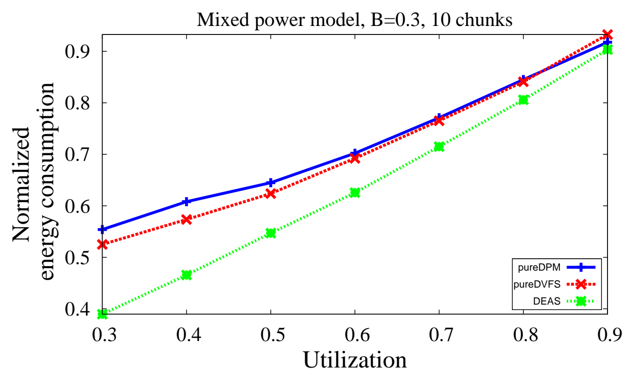

Figure 3.19: Consumptions with a Mixed power model.

Note that all the graphs show that DEAS is always able to select the right balance between DVFS and DPM depending on the specific characteristics of the architecture.

Chapter 4

Power management of multiprocessor systems

4.1 Introduction

The presence of multiprocessor systems is becoming relevant in recent computing architecture. Until now this trend has characterized big mainframes, servers and computer farms but recently even desktops and notebooks have been equipped with such hardware solutions. It is not far from truth believing that this family of computers represents the most widespread solution to support productive activities in our everyday lives.

Much more recently we are seeing how multiprocessor systems are migrating into embedded environment. In this domain power saving capabilities represents a key feature for such systems. For instance, if we consider a typical automotive power-train computing unit, operating very tightly with the car engine, it is easy to figure out that such systems usually works in a overheating environment. For these kind of systems, a policy that reduce power consumption of the CPU, and therefore working temperature, copes with the twice problem of improving system lifetime on one hand, an important aspect for battery-driven systems, and limiting system overheating on the other.

The automotive domain recently formalized the use and development of multiprocessor systems in new generation cars, most of these requirements are detailed in the AUTOSAR [68] standard and similar solution have been adopted by avionics industry, defense and consumer electronics.
Multiprocessor systems are also classified in different ways, in general they refer to Processor symmetry and Processor coupling concepts.

- **Processor symmetry** CPUs belonging to a multiprocessor system may be used for the same scope or may have be reserved for special purpose or even dedicated sub-activities. Furthermore they can have a different configuration of the Operating System, or in same case it lack. According with this description, multiprocessor systems that treat all CPUs in the same ways are commonly recognized to be symmetric multiprocessor systems (SMP), system characterized by the use of different CPUs configuration are considered asymmetric multiprocessor systems (ASMP).

- **Processor coupling** Multiprocessor systems that contain several CPUs connected together at chip level by the use of a shared communication bus is called Tightly-coupled multiprocessor systems. Multiprocessor systems whose CPUs are connected together by a high speed communication systems (often Gigabit Ethernet) are called Loosely-coupled multiprocessor systems and often refer to as cluster systems.

4.2 The case of Multiprocessor System-on-Chip (MPSoC)

Wayne Wolf and Grant Martin in [79] argue that MPSoCs constitute a unique branch of evolution in computer architecture, particularly multiprocessors, that is justified by the requirements on these systems: real-time, low-power, and multitasking applications. They present a short history of the MPSoCs as well as an analysis of the driving forces that motivate the design of these systems. They also argue that MPSoCs represent an important and distinct branch of multiprocessor, in fact they are not simply traditional multiprocessors shrunk to a single chip, but have been designed to fulfill the unique requirements of embedded applications. Furthermore the authors claim that MPSoCs form two important and distinct branches in the taxonomy of multiprocessors: homogeneous and heterogeneous multiprocessors. The importance and historical independence of these lines of multiprocessor development are not always appreciated in the microprocessor community.

4.2.1 History of MPSoCs

Wolf at All in [79] argues that MPSoCs represents a particular branch in the domain of multiprocessor systems, in fact the authors claim that they are not simply traditional multiprocessors shrunk to a single chip but have been designed to fulfill the unique requirements of embedded applications. The authors also claim that MPSoCs form two important and distinct branches in the taxonomy of multiprocessors: homogeneous and heterogeneous multiprocessors.

According with [79] this section shows a short brief about the story of MPSoCs, and then some examples of these architecture will be described with the purpose to provide how they are developed in the last years. The first MPSoC introduced in this section is the Lucent Daytona [9]. Daytona was designed for wireless base stations, in which identical signal processing is performed on a number of data channels. Daytona is a symmetric architecture with four CPUs connected together through a high-speed bus. The system was based on a enhanced SPARC V8 technology. where each CPU has an 8-KB 16-bank cache, each bank can be configured as instruction cache, data cache, or scratch pad. The chip was 200 mm^2 and ran at 100 MHz at 3.3 V in a 0.25-m based CMOS technology. The C-5 Network Processor [4] was designed for packet processing in networks. Packets are handled by processors grouped into four clusters of four units each. Three buses handle different types of traffic in the processor. The C-5 uses several additional processors, some of which dedicated. The main processor was a reduced instruction set computer (RISC). A third important class of MPSoC applications is multimedia processing. An early example of a multimedia processor is the Philips Viper Nexperia [36]. The Viper includes two CPUs: a MIPS and a Trimedia VLIW processor. A fourth important class of MPSoC applications is the cell phone processor. Early cell phone processors performed base- band operations, including both communication and multime- dia operations. The Texas Instruments (TI) OMAP architecture [5] has several implementations. The OMAP 5912 has two CPUs: an ARM9 and a TMS320C55x digital signal processor (DSP). The ARM is the master, and the DSP the slave. Another example of this category of system is the STMicroelectronics Nomadik [10], another MPSoC for cell phones. It uses an ARM926EJ as its host processor. The system also have two additional DSPs: one for video and one for audio. The ARM MPCore [42] is a homogeneous multiprocessor that also allows some heterogeneous configurations. The architecture can accommodate up to four CPUs. The Intel IXP2855 [6] is a network processor. Sixteen microengines are organized into two clusters. An XScale CPU serves as host processor. Two cryptography

accelerators perform cryptography functions. The Cell processor [53] has a PowerPC host and a set of eight processing elements known as synergistic processing elements. The Freescale MC8126 was designed for mobile base station processing. It is characterized by four Starcore SC140 VLIW processors and a shared memory.The Sandbridge Sandblaster processor [40, 41] can process four threads with full inter-locking for resource constraints. It also uses an ARM as a host processor. The Infineon Music processor is another example of a cell phone processor. It has an ARM9 host and a set of single- instruction multiple-data (SIMD) processors. The Cisco Silicon Packet Processor [37] is a example of the same category, it includes 192 configured extended Tensilica Xtensa cores. A Seiko-Epson inkjet printer Realoid SoC [67] incorporates seven heterogeneous processors: a NEC V850 control processor and six Tensilica Xtensa LX processors.
Finally, it is fundamental to cite that most of the recent platforms are based on programmable gate array (FPGA): Xilinx, Altera, and Actel. Some of these solution provide soft-based cores.

4.3 Simulation Infrastructure

Next generation Real-Time and Embedded devices is going to became multiprocessor systems, especially Multiprocessor System-on-Chip MPSoCs. Among these kind of systems there exist a particular class characterized by the following features and taken as system model for the next analysis:

- Several CPUs implementing very simple hardware architectures with no branch prediction, single issue and in-order pipeline but dedicated data and instruction caches;
- A highly predictable interconnection, such as a TDMA-based shared bus or NoC, which is able to provide the performance required by Real-Time applications.

Concerning software support for this class of architecture, the main solution adopted is partitioned. This means that each CPU has its own Operating System instance locally running. This paradigm is adopted in automotive and avionics domains, and it is defined by the corresponding AUTOSAR [68] and IMA [39] standards previously mentioned.

In [22] a functional model of the target architecture was developed to enable in-depth architectural exploration. Such architecture provide a multicore platform where all cores are 32-bit ARM-based with an associated

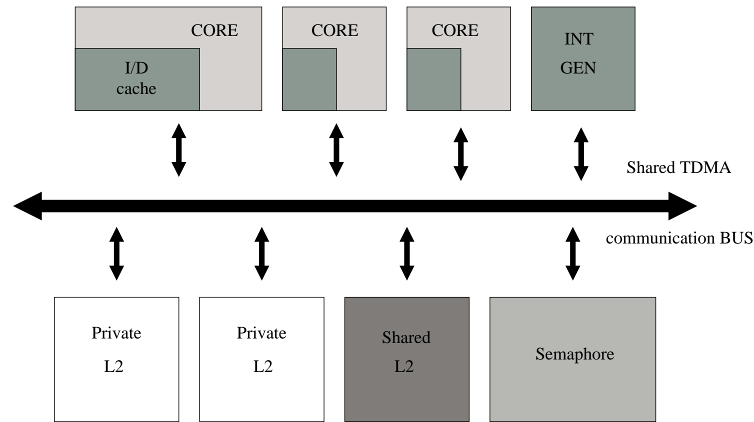

Figure 4.1: Hardware architecture

L1 cache. They are connected to a shared L2 memory via the shared bus. The L2 memory is segmented, it is characterized by a private portion associated to each core and a shared portion useful for communication or data exchange. An interrupt device is provided as well as a semaphore memory, a special memory device capable of test-and-set read operations. The latter is used for synchronizing concurrent accesses to shared resources, while the former provides the capability of efficiently propagating notifications and events in the system. The full architecture is shown in Figure4.1.

The communication bus is modeled at transactional level (TLM) and takes into account features of modern high-performance communication buses (such as [57] or ST StBus Protocol [32] [63], namely the capability of supporting burst interleaving, multiple outstanding transactions and split transfers. The bus model is packet-based, i.e. a "transaction" on the interconnect is composed by several packets. A functional TDMA arbiter is implemented. It loads the so-called Time Wheel (in literature it is also referred to as *Slot Table*) from a text file. The Time Wheel contains all the information on a single TDMA Round and unrolls over the time line, repeating infinitely during the entire simulation.

Concerning the software running on the simulated hardware platform, it is possible to run stand alone (i.e. without the support of an OS) or ERIKA [1] OS-based applications. ERIKA is an open-source (GPL2) multiprocessor real-time operating system (RTOS) kernel, implementing a collection of Application Programming Interfaces similar to those of OSEK/VDX standard [33] for automotive embedded controllers.
ERIKA is available for several hardware platforms and introduces innovative concepts, real-time mechanisms and programming features to support and

exploit the microcontrollers and multicore systems-on-a-chip. With multiprocessor hiding, it is possible to seamlessly migrate application code from a single processor to multiprocessors without changing the source code. Retargeting an application from single to multiprocessor architectures only requires minor modifications at the configuration files level, but allows retaining the source code. This requirement is covered by the OSEK standard by the OIL configuration file). The main ERIKA features related to this work are: task scheduling according to fixed and dynamic priorities; interrupt handling for urgent peripherals operation (interrupts always preempt task execution); resource sharing with Immediate Priority Ceiling protocol.

4.3.1 Software support to dynamic bus assignment

To cope with overload conditions ERIKA scheduling support has been extended by adding an implementation of the Elastic scheduling algorithm [28] where each task is considered as flexible as a spring, whose utilization can be modified by changing its period within a specified range. This software layer operates between the OS and user defined code. More specifically, each task is characterized by four parameters: a worst-case computation time C_i, a minimum period $T_{i_{min}}$ (considered as a nominal period), a maximum period $T_{i_{max}}$, and an elastic coefficient E_i. The elastic coefficient specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration: the greater E_i, the more elastic the task. In the following, T_i denotes the actual period of task τ_i, which is constrained to be in the range $[T_{i_{min}}, T_{i_{max}}]$. Moreover, $U_{i_{max}} = C_i/T_{i_{min}}$ and $U_{i_{min}} = C_i/T_{i_{max}}$ denote the maximum and minimum utilization of τ_i, whereas $U_{max} = \sum_{i=1}^{n} U_{i_{max}}$ and $U_{min} = \sum_{i=1}^{n} U_{i_{min}}$ denote the maximum and minimum utilization of the task set. The algorithm works on top of different scheduling algorithms with both static and dynamic priorities. For simplicity, tasks are scheduled by the Earliest Deadline First algorithm [58]. Hence, if $U_{max} <= U_d <= 1$, all tasks can be created at the minimum period $T_{i_{min}}$, otherwise the elastic algorithm is used to adapt the tasks periods to T_i such that $\sum \frac{C_i}{T_i} = Ud \leq 1$, where Ud is some desired utilization factor. It can easily be shown (see [28] for details) that a solution can always be found if $U_{min} \leq U_d$. If Γ_f is the set of tasks that reached their maximum period (i.e., minimum utilization) and Γ_v is the set of tasks whose utilization can still be compressed, then to achieve a desired utilization $U_d < U_{max}$ each task has to be compressed up to the following utilization:

$$\forall \tau_i \in \Gamma_v \quad U_i = U_{i_{max}} - (U_{v_{max}} - U_d + U_f)\frac{E_i}{E_v} \tag{4.1}$$

where

$$U_{v_{max}} = \sum_{\tau_i \in \Gamma_v} U_{i_{max}} \tag{4.2}$$

$$U_f = \sum_{\tau_i \in \Gamma_f} U_{i_{min}} \tag{4.3}$$

$$E_v = \sum_{\tau_i \in \Gamma_v} E_i. \tag{4.4}$$

If there exist tasks for which $U_i < U_{i_{min}}$, then the period of those tasks has to be fixed at its maximum value $T_{i_{max}}$ (so that $U_i = U_{i_{min}}$), sets Γ_f and Γ_v must be updated (hence, U_f and E_v recomputed), and equation (4.1) applied again to the tasks in Γ_v. If there exists a feasible solution, that is, if the desired utilization U_d is greater than or equal to the minimum possible utilization $U_{min} = \sum_{i=1}^{n} \frac{C_i}{T_{i_{max}}}$, the iterative process ends when each value computed by equation (4.1) is greater than or equal to its corresponding minimum $U_{i_{min}}$.

4.4 Adaptive TDMA bus allocation

The proposed algorithm [23] works as a bridge between hardware and software layers in order to allow an assignment of the bus which is aware of the core QoS requirements. The communication structure is presented in Figure 4.2. Due to its boundary position, the assigner could be implemented both in hardware or in software. Let's consider the latter situation.

The details of the algorithm are show in Figure 4.2. During system execution, a core may face a need for extra bus bandwidth, due for example to workload changes or to activation of sporadic tasks. Consequently the slave core make a request to the Master Core of the system for a certain (typically higher than the current) service level (R_i) for communication. The algorithm supports a discrete number of service levels (they are shown in Table 4.1), each service level corresponds a certain bus bandwidth percentage. Clearly, the relation between service levels and bus bandwidth depends on to the number of cores in the platform and it is calculated off-line. At predefined

0	ZERO	4	HIGH
1	MINIMUM	5	MAXIMUM
2	LOW	6	EXTREME
3	MIDDLE		

Table 4.1: Service Levels

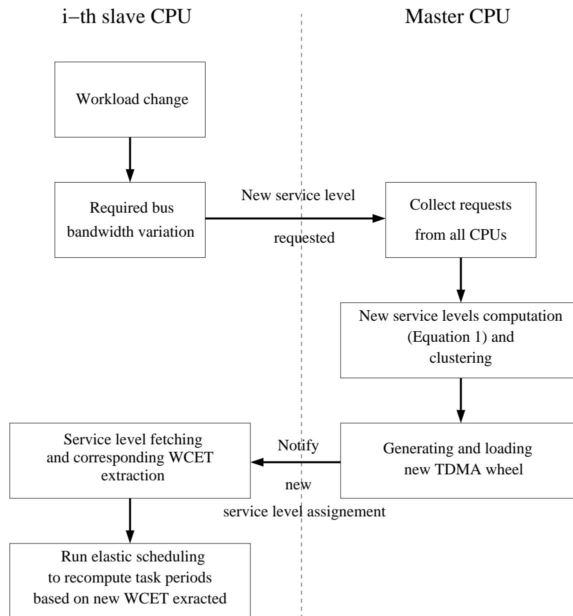

Figure 4.2: Algorithm diagram

instants, the Master fetches all new bus bandwidth requests coming from the other cores, mediates between them and recomputes the percentage (S_i) of the bus assigned to each core as

$$S_i = \frac{R_i}{\sum R_I} \tag{4.5}$$

and then generates a new Time Wheel. Clearly, the TDMA slots are set in order to assign the computed bus bandwidth to the cores. Our algorithm is not guaranteed to find the optimal solution but rather a fair trade-off betweeen all requests, being at the same time extremely efficient and lightweights. The actual service level may be different (i.e. lower) than the one requested if multiple requests happen at the same size since the algorithm mediates between all of them. Moreover, a core performing no request may see its service level changing as an effect of a new scheduling due to other cores requests. Then, the new Time Wheel is loaded in the Bus Arbiter and the new service levels are notified to the cores.

Since this change implies a variation in tasks execution times, task periods have to be recomputed according to the Elastic Scheduling algorithm previously described, using the WCETs of the task-set as input. It is important to remark that TDMA-wheel switching does not compromise the feasibility of a task-set running in a core characterized by a short TDMA

slot assignment. In fact the actual task parameters (i.e. elastic constants, T_{min}, T_{max} and the deadline equal to the period) are defined off-line, in such a way to guarantee a feasible solution to the algorithm in all possible scenarios.

The WCETs depend on the bus bandwidth assigned to the core, so they also have to be recomputed. The complexity of WCET analysis techniques makes unfeasible to do this at run-time, so they are computed offline and stored inside a lookup table (LUT) to make them available to the cores. This storage area has to fill the smallest possible space. This is obtained providing only WCETs for the limited number of service levels and imposing a quantization to the values obtained with Equation 4.5.
The LUT size is a tradeoff between the memory space used and the obtained bandwidth granularity. Increasing the number of levels allows the algorithm to better fit cores requests and leads to a solution with an higher quality of service. On the other hand, each extra level means an increasing in the algorithm overhead, more space for storing information and a higher computational effort for execution times calculation. A fair tradeoff already happens with a small number of levels. With the hardware used in the experiments that will be presented, A preliminar set of experiments, empirically demonstrates as 7 reppresents an adeguate value for the number of levels. After chosing the number of allowed bandwidth assignments (that is, the number of rows in the table), there are several options for dimensioning their values. The simplest is based on homogenousity: divide the valid bandwidth range by the number of elements. More sophisticated approaches minimize a defined metric: an example could be the aggregated bandwidth waste, i.e. is the sum of all quantization losses. In this work, a homogeneous bandwidth division has been adopted. Each row is composed by the WCETs of all tasks working with the selected bus bandwidth assignment. The WCET values can be obtained using a static code profiler and analysis tool such the one [43] developed by AbsInt.

Once the WCETs have been loaded, tasks periods can be accordingly adjusted to meet real-time requirements and tasks can be now scheduled. The overall approach gives two main benefits: first the bus TDMA allocation is QoS-aware and secondly the OS scheduler can take more accurate decisions based on the bounds given by the dynamic TDMA arbitration policy.

4.4.1 Task computation benefits

For the experimental setup let's consider a task-set composed by avionics tasks, automotive tasks and memory intensive access tasks. For the avionics case the Matlab U.S. Navy's F-14 Tomcat aircraft control task [62] has been adopted, it guarantee the aircraft to operate at a high angle of attack

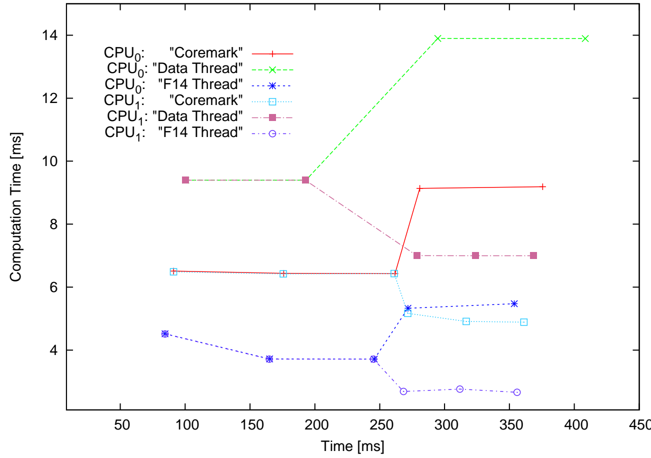

Figure 4.3: Task computation time variation

with minimal pilot workload; as automotive task the coremark [7] has been chosen, a well known and widely used benchmark in the domain of embedded systems; finally a task that performs mathematic operations such as summation and characterized by intensive memory access. Each task-set is composed by a combination of these tasks, EDF is chosen as scheduling policy and no precedence constraints nor critical section has been considered between tasks.

Figure 4.3 show the behavior of a system composed by 2 cores: CPU0 is the master core and CPU1 is the slave one. Three tasks run on each core. The y-axis reports the computation time of each task while the x-axis reports the current time. Approximatively between 100 and 200 millisecond a request of additional TDMA slot bandwidth is requested by CPU1. This request is equal to the HIGH level among the service levels available; CPU0 does not request for additional bandwidth. Such a request lead to a rebalancing of TDMA bus slots by the master core. Starting from this moment the computation time of each task running on CPU1 improves while the corresponding one on CPU0 get worse. The request, triggered by the third job of the avionics task running on CPU1, is made between 100 and 200 millisecond and the advantages for the CPU1 can be already appreciated in the fourth job for avionics and automotive tasks, and from the third job in case of the mathematical task.

Under these conditions, the task-set experiences a variable range of computation times: from 2 milliseconds for the avionic task up to 15 millisecond for the mathematical task. In order to catch the overhead introduced by the algorithm, a set of measurements have been performed on the avionic task.

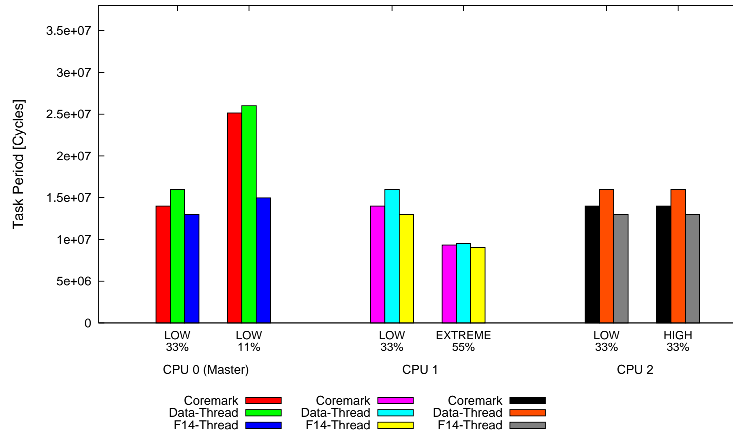

Figure 4.4: Task periods and service levels

The average overhead introduced is less than 5% of the computation time of the task itself. This overhead can be divided in a negligible (less than 5 microseconds) part spent for the OS context-switch, while the majority of it is equally spent by the elastic manager to collect cores requests and accomplish task period variations, and to update the TDMA-wheel reallocation, i.e writing of the new values in arbiter registers and triggering the table switch. Moreover, the code to accomplish these tasks could be further optimized: for instance, the calculation of the new task periods has no FPU support which could instead provide a further improvement of the overall performance. However, even this not optimized version of the code has an execution time which is less than 100 times the basic context switch.

4.4.2 Bus Access Time and Periods

In Figure 4.4 the variation of task periods has been show. This is the case for three cores and three tasks for each core. As usual, CPU0 represents the master core while CPU1 and CPU2 are the remaining slaves that compose the systems. The amplitude of the histogram bars indicates the periods of the tasks and they are collected in three clusters, one for each CPU. Inside each cluster it is possible to appreciate the value of each task period corresponding with the old (on the left), requested (on the right) and actual (in red) service level. The system starts with a fairly distributed level of service equal to LOW, the corresponding TDMA slot assignment is 33% for each CPUs. This scenario is represented by the first set of bars inside each cluster. The second set of bars inside each cluster shows that CPU1 and CPU2 ask for additional bus bandwidth, respectively an EXTREME

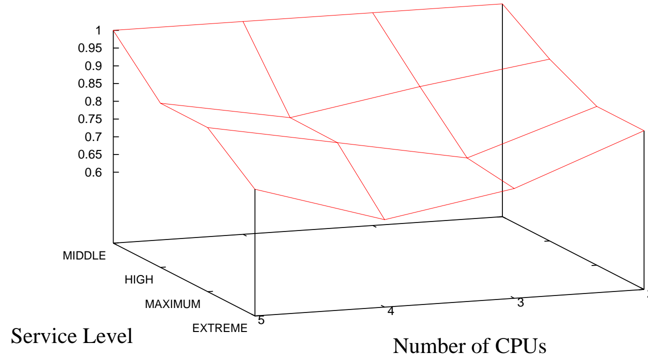

Figure 4.5: Buss access time and service levels

and a HIGH level of service, while CPU0 (master core) makes no requests. According with this set of requests, the master core assigns the Time Wheel the following way: MINIMUM (11%) for itself, HIGH (55%) for CPU1 that ask for the highest level of service and LOW (33%) for the CPU2. Note that with this particular combination of requests CPU2 is not able to improve its bandwidth and, despite of its request, it holds the initial percentage of TDMA bandwidth. This case has been deliberately chosen to highlight that requests for additional bandwidth must be considered as part of the whole set of demands coming from all CPUs.

Figure 4.5 provides an exhaustive representation of the system response in terms of bus access time as a function of the number of CPUs and service level requested. For each measure, the values are normalized over the case with the same number of CPUs and the service level equal to MIDDLE. As usual let evaluate a system characterized by a composite task-set and let collect the response of a single measured CPU, that ask for different service level, in a multiprocessor context with variable number of CPUs. The service level of the analyzed CPU starts from MIDDLE up to EXTREME, while the whole number of cores that compose the system varies from two up to five. The system starts with a fair bus assignment (MIDDLE service level) to the CPUs. The only CPU licensed to ask for different service level is the measured CPU, the remaining ones hold the initial service level (MIDDLE). The figure shows the improvements experienced by the measured CPU: the access time decrease if the number of CPUs or the service level requested increase. This trivial result is shown with the purpose of quantifying the advantage in terms of latency for the each bus access from the single CPU

point of view, compared to the case of a static and fair assignment: same bandwidth for each CPU.

4.4.3 Quality of Control index

In control applications the performance of a periodic control task is a function of the activation period. Increasing the task activation period leads to a performance degradation, which is typically measured through a Performance Index $J(T)$ [27, 75]. Often, instead of using the performance index, many algorithms use the difference $\Delta J(T)$ between the index and the value of the performance index $J*$ of the optimal control. Many control systems belong to a class in which the function expressing the degradation is monotonically decreasing, convex and can be approximated as

$$\Delta J(T_i) = \alpha_i e^{-\frac{\beta_i}{T_i}}$$

where the magnitude α_i and the decay rate β_i characterize the single task. The evaluation of the whole task set is computed as

$$\Delta J = \sum_{i=1}^{n} w_i \Delta J(T_i) = \sum_{i=1}^{n} w_i \alpha_i e^{-\frac{\beta_i}{T_i}}$$

where the w_i are used to characterize the relative importance of the tasks.

To have a common scale for all task sets, the Quality of Control index paper is expressed as

$$QoC = \frac{\Delta J_{nom}}{\Delta J}. \tag{4.6}$$

where ΔJ_{nom} is the value of the index calculated when tasks run at their nominal periods. A value of 1 means that all tasks are running with nominal periods.

All coefficients α_i and w_i are set to 1 for simplicity, while β_is are set to 20 in order to use the whole range [0,1] of the QoC index. Taking the previous example (shown also in Figure 4.4), the values of ΔJ for CPU0 changes from 2.4 to 2.6 due to the Time Wheel variation. In Table 4.2 are presented the value of QoC for different approaches computed for the same example and normalized over the difference between$J_{t_{max}}$ and $J_{t_{min}}$. where $QoC_{t_{min}}$ is the best possible QoC (ΔJ_{nom}) obtained with a set of CPUs each with a dedicated bus. This case represents the virtual upper bound, but it is not really experienced, because we are working in a multiprocessor environment with shared communication bus; QoC_{dyn} is obtained adopting our run-time algorithm; QoC_{fair} is the result of a fair TDMA scheduling, where all the slots have the same size. This is also the starting point in our

$\Delta QoC_{t_{min}}$	1.0000
ΔQoC_{dyn}	0.7067
ΔQoC_{fair}	0.6947
$\Delta QoC_{min_{bwd}}$	0.6357
$\Delta QoC_{t_{max}}$	0.5925

Table 4.2: QoC Indexes.

experiments, before the slot and task periods are modified. $QoC_{min_{bwd}}$ is a virtual QoC in case each core is assigned a low bandwidth (10% of the TDMA Round). This situation is typical of systems where each CPU must guarantee the timing constrains even with a small static slot assignment. Finally, $QoC_{t_{max}}$ is the QoC provided by the system if the longest allowed period is chosen for each task on every CPU. Table 4.2 shows that a system with the capability to dynamically adjust TDMA slots is able, starting by a fixed TDMA allocation, to have an improvement from 27% up to 31% of the QoC index. Moreover, the introduced overhead has negligible effect on the QoS (as prevoiusly said, in the average case it is around 5% of the computational time for the fastest task). On the contrary, a system characterized by a standard TDMA slot assignment is forced to operate with $QoC_{min_{bwd}}$, due to a lower bus-access-time.

This solution introduces an algorithm able to resize TDMA slots of communication bus accessed concurrently accordingly with the actual taskset requirements. The target architectures are RT MPSoCs where running tasks face unpredictable situations (external interrupts, interaction with users) and thus the standard off-line WCET analysis techniques are no longer efficient. This results in a loss of accuracy and consequently a loss of performance both of the TDMA bus scheduling and Elastic Scheduling, which cannot work at their best.

The novelty of the proposed architecture consists of a tight integration between the bus arbiter, that is in charge of managing shared bus allocation, and the elastic scheduling running on top of the OS system. TDMA Time Wheel and task periods are adjusted at run-time to meet the performance constraints. The algorithm is aware of QoS requirements of the given taskset, and it has been validate by adopting a real RT benchmarks platform able to run a typical embedded taskset.

4.4.4 Power management considerations

The hybrid solution described in this chapter was initially developed to manage overload in multiprocessor systems by combining hardware/software

strategies, but under certain conditions a similiar approach can be easily adapted to cope with different objectives.
For instance in some applications where MPSoCs are commonly used, power managment capabilities reppresent the first requirement and often there are some scenario where a subset of CPUs composing the MPSoC is not needed to be fully operative.
In this cases the usual power management strategies, such as DVS or DPM may be adopted together with a dinamic TDMA bus assignement. The previous chapter demonstrates how a correct TDMA bandwidth assignement aims to reduce computation time, the time saved (idle time) can be exploited to perform power saving strategies rather then incresing the sample rate (e.g. decresing the period T) like in the application previously described in the this chapter.

Let's consider a MPSoC where at a certain moment some CPUs are idle or are characterized by a small utilization factor U, while another CPUs belonging to the same MPSoC is experiencing an overload but it also need to reduce its power absorption. Under these conditions it is very difficult applying power management strategies since one of the systems is overloaded. For this case a static TDMA bandwidth assignement is not optimal, and a better solution can be found because the other CPUs are not overloaded.

According with this idea a set of experiments have shown that proving additional bandwidth to a CPU in overload condition can significantly provide a growing of the idle time for that specific CPU, therefore such idle time can be exploited as a chance to apply power management strategies, whose effects is as more effective as the idle time increses.
In order to cope with this new requirement, the algorithm described in [23] has been sligtly modified according with the schema depicted in Figure4.6

The algorithm is always the same except for the last step, in this case if the i-th CPU that requires additional bandwidth needs to apply power management strategies, the elastic scheduling step has to be skipped and the consequent idle time can be used to apply DVS or DPM. Therefore, according with Figure 4.6 and focusing on the final step, there exists two possibilities: in case of CPU needing to save energy the path chosen is the one denoted in red with label 1, otherwise the path is the usual that triggers the elastic scheduling algorithm and depicted in black wih label 2. The schedulability is always guaranteed for both paths, in fact if the schedulability of the CPU that obtains additional bandwidth was guaranteed before the new bandwidth allocation, it is still guaranteed after new bandwidth assigment, because the new utilization factor meet the inequality 4.7 under EDF, that is the scheduling policy adopted in this scenario.

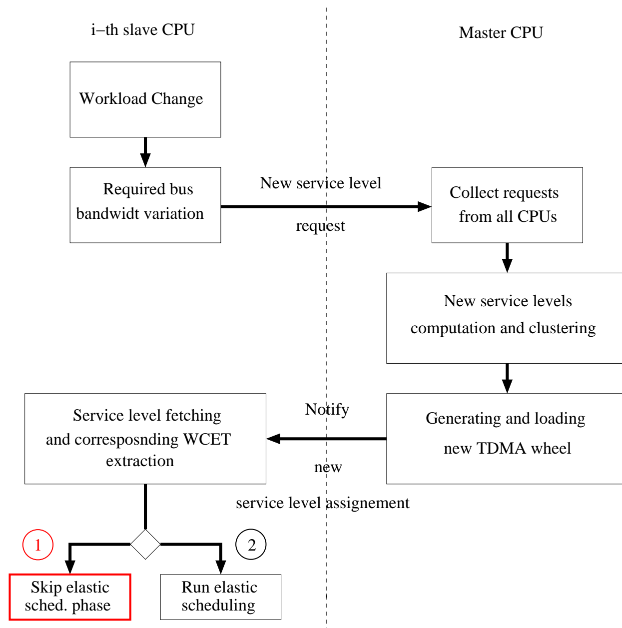

Figure 4.6: Power-aware TDMA bus allocation schema

$$U_{new} \leq U_{old} \leq 1 \tag{4.7}$$

The 4.7 is always verified since the computation time of each job decreases after the new TDMA bandwidth allocation.

For the remaining CPUs the schedulability is guaranteed anyway by the elastic scheduling manager. Since the bandwidth assigment for these CPUs is not advantageous, and therefore the computation time of the jobs increse, the elastic scheduling adapts the periods of the tasks to meet the new timing constraints caused by the increse of the computation time of each tasks. In order to compensate this effect, the period of each tasks is accordingly relaxed.

In order to appreciate the advantages achiveded in term of idle time by a system able to assign TDMA communication bandwidth according with this strategy, a set of tables describing different experiments is reported next. Each table reppresents a scenario where a single CPU (always the $CPU1$) experiences a need of additional bandwidth to reduce computation time and consequently having the chance to apply power saving stategiese, while the remaining CPUs reduce their activities, or goes into fully idle state.

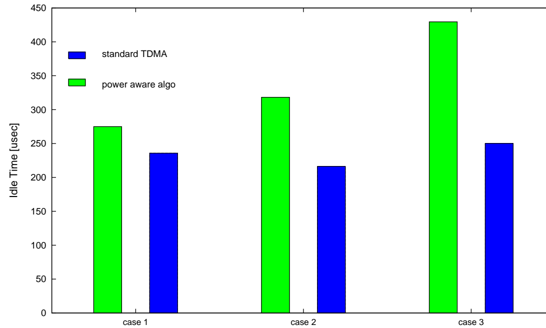

Figure 4.7: idle time caused by dynamic TDMA bandwidth assignement

The chosen task-set is always composed by the following three tasks: the F14 Tomcat control task, the Coremark and a numerical task performing a lot of bus transactions and mathematical computation. The hardware is always the emulated ARM7TDMA (MPARM), the operating system is Erika and the scheduler is EDF.

The Figure 4.7 shows three couples of bars that represent three different cases, each case describes a system whose feature change at run-time. The green bars of the histogram represents the idle time experienced by the $CPU1$ by applying this method, while the blue bars reppresents the idle time of the same CPU with the standard TDMA bandwidth assignement. For each case, a corresponding table 4.3, 4.4, 4.5 reports the initial and final status of the system in term of workload because we are interested in the system response corresponding to the modified conditions. Both final and initial status describe the system in two different instants, each case may be featured by a different number of CPUs, and different number of tasks may run or may be suppressed passing from the initial to the final status, it is also possible figure out the percentage of bandwidth assigned by the bus arbiter in both initial and final states. For each case in tables, the gap between the green bar and the blue one of the Figure 4.7, represents the amount of time this strategy is able to save (in term of idle time) with respect to the standard static approach. According with this description the three experiments analyzed are summarized in the following three tables.

	Initial Status		Final Status	
Bandwidth (%)	50%	50%	20%	80%
–	CPU0	CPU1	CPU0	CPU1
F14 Tomcat	×	×	suppressed	×
Coremark	×	×	suppressed	×
math	×	×	×	×

Table 4.3: case 1: two CPUs

	Initial Status				Final Status			
Bandwidth (%)	25%	25%	25%	25%	10%	70%	10%	10%
–	CPU0	CPU1	CPU2	CPU3	CPU0	CPU1	CPU2	CPU3
F14 Tomcat	×	×	×	×	suppressed	×	suppressed	suppressed
Coremark	×	×	×	×	suppressed	×	suppressed	suppressed
math	×	×	×	×	×	×	×	suppressed

Table 4.4: case 2: four CPUs

Initial Status					
Bandwidth (%)	20%	20%	20%	20%	20%
–	CPU0	CPU1	CPU2	CPU3	CPU4
F14 Tomcat	suppressed	×	suppressed	suppressed	suppressed
Coremark	suppressed	×	suppressed	×	suppressed
math	×	×	×	suppressed	suppressed
Final Status					
Bandwidth (%)	10%	70%	10%	10%	10%
–	CPU0	CPU1	CPU2	CPU3	CPU4
F14 Tomcat	suppressed	×	suppressed	suppressed	suppressed
Coremark	suppressed	×	suppressed	×	suppressed
math	×	×	×	suppressed	suppressed

Table 4.5: case 3: five CPUs

For instance, focusing on case number one, the system is composed by two CPUs, in the inital status the whole task-set runs on both CPUs, and the bus assignement is equal to 50% for both CPUs. In the final status the system reaches a new configuration where the $CPU1$ holds the initial configuration in term of running tasks, but it is also get an improved bandwidth assignement, since it passes from 50% to 80% of available bandwidth. The $CPU0$ insted, experiences a reduction in the number of running tasks, in particular in the final status the F14 Tomcat task and the Coremark have been suppressed in the $CPU1$. The bandwidth assigned is also changed, in fact it passes from 50% to 20%.
This example shows as a reduction of the activities for the $CPU0$ can be exploited as a chance to improve the performances of the $CPU1$. In fact the $CPU1$ is now able reduce computation time of its tasks since it can count on a improved bandwidth allocation. As consequence, the idle time intervals generated can be reclaimed to apply power management strategies and save energy.

Chapter 5

Conclusions

This work addresses the problem of reducing energy consumption in interconnected embedded systems with time and communication constraints. Based on system typologies this thesis tackled with two categories of problems.
The first one concerns with these systems that have to be power aware on one hand but they also have to guarantee communication services, because they are part of a larger interconnected and distributed system, where the communication bandwidth is a shared and limited resource.
The second typology of system is a interconnected system once again, but it operates on different scale as it is a MPSoC. Since these kind of microprocessor is going to become one of the mostly adopted solution in the domain of embedded systems too, the requirements are the same of the previous case but they have been addressed according with a different strategies.

For the first problem, the proposed solutions exploits both DPM and DVS techniques to reduce the energy consumption within each computational node. This goal is achieved by balancing the two main strategy previously mentioned and currently supported by modern hardware architecture, that depend on the specific CPU characteristics, actual workload, and bandwidth allocation. The method has been developed under realistic assumptions, such as discrete frequency levels, mode switch overhead, and communication constraints. Experimental results showed that the combined DPM/DVFS approach dominates each individual technique (pureDPM and pureDVFS) for all power models and any task set utilization for the radio interconnected system.

For the second problem, a new solution operating at two levels have been proposed. At the first level a flexible bus arbiter mediates among CPUs to

assign communication bandwidth according with the current needs of each CPU. At the second level the elastic scheduler reconfigures the task-set to meet real time constraints. This approach demonstrates as a dynamic communication bandwidth allocation provided at hardware level, combined with software solution for workload management like elastic scheduling, provides advantages for MPSoCs systems both in term of quality of control and power saving.

Appendix A

Erika

Erika Enterprise [1] is an open-source and Real Time Kernel implementing the OSEK/VDX API. ERIKA Enterprise is based on RT-Druid, which is a development environment distributed as a set of Eclipse plug-ins.

Erika Enterprise implements the standard OSEK/VDX conformance classes BCC1, BCC2, ECC1 and ECC2. Moreover, ERIKA provides other custom conformance classes named FP (Fixed priority), EDF (Earliest deadline first scheduling), and FRSH (an implementation of resource reservation protocols).

Erika Enterprise supports multicore the partitioning of tasks in a multicore system, this feature is available for several hardware architectures. Erika support the automatic code generation. This feature is provided by Scilab.

The main features of Erika Enterprise are:

- Real-time kernel, priority based, with stack sharing for RAM optimization.
- Minimal multithreading RTOS interface
- Scheduling parameters assigned at task activation and they will never change at run-time;
- Interface similar to the one proposed by the OSEK/VDX consortium for the OS, OIL, ORTI standards (the kernel has not been certified yet by the OSEK/VDX consortium)
- RTOS API for: Tasks, Events, Alarms, Resources, Application modes, Semaphores, Error handling.

- Support for conformance classes (FP, BCC1, BCC2, ECC1, ECC2, EDF) to match different application requirements;
- Support for preemptive and non-preemptive multitasking;
- Support for fixed priority scheduling and Preemption Thresholds;
- Support for Earliest Deadline First (EDF) scheduling;
- Support for stack sharing techniques, and one-shot task model to reduce the overall stack usage;
- Support for shared resources;
- Support for periodic activations using Alarms;
- Support for centralized Error Handling;
- Support for hook functions before and after each context switch.
- Crosstool License (GPL with Linking Exception)

The main features of RT-Druid are:

- Development environment based on the Eclipse IDE;
- Support for the OIL language for the specification of the RTOS configuration;
- Graphical configuration plugin to easily generate the OIL configuration file and to easily configure the RTOS parameters;
- Full integration with the Cygwin development environment to provide a Unix-style scripting environment;
- Apache ANT scripting support for code generation;
- RT-Druid graphical editor
- RT-Druid code generator per Erika Enterprise, including the code generator for Erika Enterprise Basic
- Support the ORTI standard for application debugging and tracing with lauterbach debuggers

Appendix B

MPARM

MPARM [2] is a multi-processor cycle-accurate architectural simulator. Its purpose is the system-level analysis of design tradeoffs in the usage of different processors, interconnects, memory hierarchies and other devices. MPARM output includes accurate profiling of system performance, execution traces, signal waveforms, and, for many modules, power estimation.

The processor model used for the applications described in this work is a ARM7TDMI processor. The processor features are provided by the SWARM engine, a Software ARM, that is a software simulation of a ARM7 processor written in C++. SWARM [3] represents the core of the MPARM. It supports several memory models:

- Memories;
- scratchpad memories (as cache replacements, as local buffers and as point-to-point interprocessor queues);
- Snoop devices (to provide cacheability in multiprocessor shared-memory environments);
- Synchronization devices (semaphores, inter-processor interrupt devices);
- DMA controllers;
- "Smart memories" (memories with built-in DMA engines);
- Frequency scaling devices (to provide runtime frequency and voltage scaling; includes dual-frequency FIFOs to attach components to each other);
- FFT engine.

At the interconnection level, MPARM provides:

- AMBA AHB. Various models are available, written both in pure SystemC and (under STMicroelectronics NDA) developed with the OCCN libraries. AHB to
- AHB bridges are available. Multilayer (crossbar) components are available;
- STBus (under STMicroelectronics NDA) interconnect. Type 3 nodes are supported, in various topologies;
- AMBA AXI;
- xpipes NoC. This NoC was developed in-house. Porting is almost done, with final debugging and statistics collection going on.

There are several ports available for MPARM:

- RTEMS;
- Linux;
- Erika.

MPARM has been adopted by several Universities and Industrial players.

Bibliography

[1] Erika enterprise: Open source rtos for single- and multi-core applications. www.evidence.eu.com.

[2] Mparm: a multi-processor cycle-accurate architectural simulator. http://www-micrel.deis.unibo.it/sitonew/research/mparm.html.

[3] Swarm: a modular simulation of an arm 7 processor. http://www.cl.cam.ac.uk/ mwd24/phd/swarm.html.

[4] C-5 network processor architecture guide, May 2001. http://www.freescale.com/.

[5] Omap5912 multimedia processor device overview and architecture reference guide, Mar 2004. http://www.ti.com/.

[6] Intel ixp2855 network processor, 2005. http://www.intel.com.

[7] CoreMark a EEMBC banchmark. http://www.coremark.org.

[8] N. AbouGhazaleh, D. Moss, B. Childers, and R. Melhem. Toward the placement of power manegement points in real time applications. In *Proc. of the Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, Barcelona, Spain, 2001.

[9] B. Ackland, A. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J. H. ONeill, J. Othmer, E. Sackinger, K. J. Singh, J. Sweet, C. J. Terman, , and J. Williams. A single-chip, 1.6-billion, 16-b mac/s multiprocessor dsp. In *IEEE J. Solid-State Circuits*, pages 412–424, March 2000.

[10] A. Artieri, V. DAlto, R. Chesson, M. Hopkins, and M. C. Rossi. Nomadikopen multimedia platform for next generation mobile devices. 2003.

[11] J. Augustine, S. Irani, and C. Swamy. Optimal power-down strategies. *In FOCS*, pages 530–539, 2004.

[12] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 95–105, Washington DC, USA, 2001.

[13] H. Aydin, R. Melhem, D. Mossé, and P. Mejia Alvarez. Power-aware scheduling for periodic real-time tasks. volume 53, pages 584–600, Washington, DC, USA, May 2004.

[14] P. Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. *In SODA*, pages 364–367, 2006.

[15] Sanjoy K. Baruah, Rodney R. Howell, and Louis Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[16] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *In Proc. of the 11th Real-Time Systems Symposium*, pages 182–190, Orlando, Florida, USA, 1990. IEEE Computer Society Press.

[17] E. Bini, G. Buttazzo, and G. Lipari. Speed modulation in energy-aware real-time systems. In *IEEE Proc. of the 17th Euromicro Conference on Real-Time Systems*, Palma de Mallorca, July 2005.

[18] E. Bini and C. Scordino. Optimal two-level speed assignment for real-time systems. *International Journal of Embedded Systems*, 4(2):101–111, July 2009.

[19] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *ECRTS'01, IEEE Computer Society*, pages 59–66, Delft, The Netherlands, 2001.

[20] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52:933–942, 2003.

[21] Enrico Bini and Giorgio C. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, June 2004.

[22] P. Burgio, M. Ruggiero, and L. Benini. Simulating future automotive systems, march 2010. http://www-micrel.deis.unibo.it/burgio/pub/docs/TLMBUSTechReport.pdf.

[23] Paolo Burgio, Martino Ruggiero, Francesco Esposito, Mauro Marinoni, Giorgio Buttazzo, and Luca Benini. Adaptive tdma bus allocation and elastic scheduling: a unified approach for enhancing robustness in multi-core rt systems. In *Proceedings Of The 28th Ieee International Conference On Computer Design - ICCD10*, Amsterdam, the Netherlands, Oct. 2010.

[24] G. Buttazzo. Rate monotonic vs. edf: Judgment day. volume 29, pages 5–26, Jenuary 2005.

[25] G. Buttazzo. Achieving scalability in real-time systems. *IEEE Computer*, 4(1):54–59, 2006.

[26] G. Buttazzo, M. Marinoni, and F. Esposito. Deliverable n 5.5 integrating scheduling and dvs management, Jenuary 2010. http://www.predator-project.eu/.

[27] G. Buttazzo, M. Velasco, P. Marti, and G. Fohler. Managing quality-of-control performance under overload conditions. In *Proc. 16th IEEE Euromicro Conference on Real-Time System*. IEEE Computer Society Press, 2004.

[28] G C. Buttazzo, G Lipari, M. Caccamo, and Abeni L. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51:289–302, 2002.

[29] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 1998.

[30] Jian-Jia Chen and Tei-Wei Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *ICCAD '07: Proc. of the 2009 International Conference on Computer-Aided Design*, pages 289–294, New York, NY, USA, 2007. ACM.

[31] Jian-Jia Chen and Tei-Wei Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *International Conference on Computer-Aided Design (ICCAD)*, pages 289–294, 2007.

[32] G. A. Clouard, G. Mastroroceo, F. Carbognani, and A. Perrin. F. Stbus communication system concepts and definitions, 2002. http://www.st.com/stonline/products/literature/um/14178.pdf.

[33] OSEK/VDX consortium. Osek/vdx standard. http://www.osek-vdx.org.

[34] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. pages 807–813, 1974.

[35] Vinay Devadas and Hakan Aydin. On the interplay of dynamic volatage scaling and dynamic power management in real-time embedded applications. In *EMSOFT'08: Proc. of the 8th ACM Conference on Embedded Systems Software*, pages 99–108, New York, NY, USA, 2008. ACM.

[36] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor soc for advanced set-top box and digital tv systems. 18(5):21–31, Sept 2001.

[37] W. Eatherton. The push of network processing to the top of the pyramid. Oct. 2005.

[38] C.-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), 1998.

[39] Ren L.C. Eveleens. Integrated modular avionics development guidance and certification considerations. National Aerospace Laboratory NLR, 1006BM Amsterdam Netherlands.

[40] J. Glossner, D. Iancu, J. Lu, E. Hokenek, and M. Moudgill. A software-defined communications baseband design. 41(1):120–128, May 2003.

[41] J. Glossner, M. Moudgill, D. Iancu, G. Nacer, S. Jinturkar, S. Stanley, M. Samori, T. Raja, and M. Schulte. The sandbridge sandbridge convergence platform. 2005. http://www.sandbridgetech.com.

[42] J. Goodacre and A. N. Sloss. Parallelism and the arm instruction set architecture. *Computer*, 38:42–50, Jul 2005.

[43] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis, 2004. http://www.absint.de/aiT_WCET.pdf.

[44] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo. Adaptive dynamic power management for hard real-time systems. *In the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 23–32, 2009.

[45] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo. Periodic power management schemes for real-time event streams. *In the 48th IEEE Conf. on Decision and Control (CDC)*, pages 6224–6231, 2009.

[46] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo. Adaptive power management for real-time event streams. *In the 15th IEEE Conf. on Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[47] Kai Huang, Luca Santinelli, Jian-Jai Chen, Lotar Thiele, and Giorgio Buttazzo. Adaptive dynamic power management for hard real-time systems. In *RTSS'09: Proc. of the 30th IEEE Real-Time Systems Symposium*, Washington, DC, USA, 2009. IEEE Computer Society.

[48] Kai Huang, Luca Santinelli, Jian-Jai Chen, Lotar Thiele, and Giorgio Buttazzo. Periodic power management schemes for real-time event streams. In *CDC'09: Proc. of the 48th IEEE Conference on Decision and Control*, pages 6224 – 6231, Shangai, China, 2009. IEEE Computer Society.

[49] Kevin Jeffay and Donald L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, pages 212–221, December 1993.

[50] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proc. of the 41st ACM/IEEE Design Automation Conference (DAC)*, pages 275–280, 2004.

[51] Kim, Todd Austin, Jie S. Hu, and Mary Jane. Leakage current: Moore's law meets static power, 2003.

[52] W. Kim, D. Shin, H.S. Yun, J. Kim, and S.L. Min. Performance camparision of dynamic voltage scaling algorithms for hard real-time systems. In *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 219–228, San Jose, California, September 2002.

[53] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. 26(3):10–23, Jun 2006.

[54] Anis Koubâa, Mário Alves, Eduardo Tovar, and André Cunha. An implicit gts allocation mechanism in ieee 802.15.4 for time-sensitive wireless sensor networks: theory and practice. *Real-Time Syst.*, 39(1-3):169–204, 2008.

[55] G. Sudha Anil Kumar and G. Manimaran. Energy-aware scheduling of real-time tasks in wireless networked embedded systems. In *RTSS '07: Proc. of the 28th IEEE International Real-Time Systems Symposium*, pages 15–24, Washington, DC, USA, 2007. IEEE Computer Society.

[56] C.-G. Lee, K. Lee, and J. H. et al. Bounding cache-related preemption delay for real-time systems. In *IEEE Transactions on software engineering*, volume 27, November 2001.

[57] ARM Limited. Simulating future automotive systems, 2003. http://www.arm.com/products/solutions/AMBA3AXI.html.

[58] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[59] M. Marinoni and G. Buttazzo. Elastic dvs management in processors with discrete voltage/frequency modes. In *IEEE Transactions on Industrial Informatics*, volume 3, pages 95–105, 2007.

[60] Mauro Marinoni, Mario Bambagini, Francesco Prosperi, Francesco Esposito, Gianluca Franchino, Luca Santinelli, and Giorgio Buttazzo. Platform-aware bandwidth-oriented energy management algorithm for real-time embedded systems. In *Proc. of the 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA 2011)*, Toulouse, France, September 2011.

[61] T. Martin and D. Siewiorek. Non-ideal battery and main memory effects on cpu speed-setting for low power. *IEEE Transactions on VLSI Systems*, 9(1):29–34, 2001.

[62] Matworks. Designing an f-14 high angle of attack pitch mode control. http://www.mathworks.com.

[63] S. Microelectonics. St microelectronics, tech. rep, 2002. http://www.st.com/stonline/products/technologies/soc/stbus.htm.

[64] H. S. Negi T. Mitra and A. Roychoudhury. Accurate estimation of cache-related preemption delay. *In CODES+ISSS03*, October 2003.

[65] Bren Mochocki, Dinesh Rajan, Xiaobo Sharon Hu, Christian Poellabauer, Kathleen Otten, and Thidapat Chantem. Network-aware dynamic voltage and frequency scaling. In *RTAS '07: Proc. of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 215–224, Washington, DC, USA, 2007. IEEE Computer Society.

[66] Christian Nastasi, Mauro Marinoni, Luca Santinelli, Paolo Pagano, Giuseppe Lipari, and Gianluca Franchino. Baccarat: a dynamic realtime bandwidth allocationpolicy for ieee 802.15.4. In *Proc. of IEEE*

Percom 2010, International Workshop on Sensor Networks and Systems for Pervasive Computing (PerSeNS 2010), Mannheim, Germany, 2010.

[67] S. Otsuka. Design of a printer controller soc using a multiprocessor configuration. Oct. 2006.

[68] AUTOSAR partnership. Autosar (automotive open system architecture), 2003. http://www.autosar.org/.

[69] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102, Banff, Canada, October 2001.

[70] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 89–102, Washington, DC, USA, 2008. IEEE Computer Society.

[71] Luca Santinelli. *Adaptive Resource Reservation*. PhilosophiæDoctor (Ph.D.) thesis in Embedded Systems, Retis Lab. - Scuola Superiore Sant'Anna di Pisa, Piazza Martiri della Libertà, 33 - 56127 Pisa (Italy), 2010.

[72] Luca Santinelli, Mauro Marinoni, Francesco Prosperi, Francesco Esposito, Gianluca Franchino, and Giorgio Buttazzo. Energy-aware packet and task co-scheduling for embedded systems. In *Proc. of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 279–288, Scottsdale, Arizona, USA, 2010.

[73] Curt Schurgers, Vijay Raghunathan, and Mani B. Srivastave. Modulation scaling for real-time energy aware packet scheduling. In *Global Telecommunications Conferance (GLOBECOMM 01)*, pages 3653–3657, San Antonio, Texas (USA), 2001.

[74] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *Proc. 24th IEEE Real-Time Systems Symposium*, pages 40–51, Cancun, Mexico, December 2003.

[75] D. Seto, J. Lehoczky, L. Sha, and K. Shin. On task schedulability in real-time control systems. In *Proc. 17th IEEE Real-Time Systems Symposium*, pages 13–21. IEEE Computer Society Press, 1996.

[76] Marco Spuri, Giorgio Buttazzo, and Fabrizio Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, pages 210–219, Pisa, Italy, December 1995.

[77] Swaminathan and K. Chakrabarty. Pruning-based, energy-optimal, deterministic i/o device scheduling for hard real-time systems. *ACM Trans. Embedded Comput. Syst.*, 4(1):141–167, 2005.

[78] V. Swaminathan and C. Chakrabarti. Energy-conscious, deterministic i/o device scheduling in hard real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(7):847–858, Oct 2003.

[79] Wayne Wolf and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 27(10), Oct 2008.

[80] Chuan-Yue Yang, Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. System-level energy-efficiency for real-time tasks. In *the 10th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, pages 266–273, 2007.

[81] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proc. of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 374–382, 1995.

[82] Jun Yi, Christian Poellabauer, Xiaobo Sharon Hu, Jeff Simmer, and Liqiang Zhang. Energy-conscious co-scheduling of tasks and packets in wireless real-time environments. In *RTAS '09: Proc. of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 265–274, Washington, DC, USA, 2009. IEEE Computer Society.

[83] Yumin Zhang, Xiaobo Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proc. of the 39th ACM/IEEE Design Automation Conference (DAC)*, pages 183–188, 2002.

[84] Baoxian Zhao and Hakan Aydin. Minimizing expected energy consumption through optimal integration of dvs and dpm. In *ICCAD '09: Proc. of the 2009 International Conference on Computer-Aided Design*, pages 449–456, New York, NY, USA, 2009. ACM.

[85] Y. Zhu and F. Mueller. Feedback edf scheduling of realtime tasks exploiting dynamic voltage scaling. *Real-Time Systems Journal*, 3(1):33–63, December 2005.

[86] Jianli Zhuo and Chaitali Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Proc. of the 42nd ACM/IEEE Design Automation Conference(DAC)*, pages 628–631, 2005.