Energy-Centric Scheduling for Real-Time Systems

Gruian, Flavius

2002

Energy-Centric Scheduling for Real-Time Systems

Flavius Gruian

Doctoral dissertation, 2002

Department of Computer Science
Lund Institute of Technology
Lund University

This thesis is submitted to the Board of Research: FIME — Physics, Informatics, Mathematics and Electrical Engineering — at Lund Institute of Technology (LTH), Lund University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering.

Document built with:
LaTeX 2_ε, gnuplot 3.7.2, FreeHand 10, Xfig 3.2 patchlevel 2, and MS Excel.

Version: November 16, 2002

ISBN 91-628-5494-1
ISSN 1404-1219
Dissertation 15, 2002
LU-CS-DISS:2002-2

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: Flavius.Gruian@cs.lth.se
WWW: http://www.cs.lth.se/home/Flavius_Gruian

ABSTRACT

ENERGY CONSUMPTION is today an important design issue for all kinds of digital systems, and essential for the battery operated ones. An important fraction of this energy is dissipated on the processors running the application software. To reduce this energy consumption, one may, for instance, lower the processor clock frequency and supply voltage. This, however, might lead to a performance degradation of the whole system. In real-time systems, the crucial issue is timing, which is directly dependent on the system speed. Real-time scheduling and energy efficiency are therefore tightly connected issues, being addressed together in this work.

Several scheduling approaches for low energy are described in the thesis, most targeting variable speed processor architectures. At task level, a novel speed scheduling algorithm for tasks with probabilistic execution pattern is introduced and compared to an already existing compile-time approach. For task graphs, a list-scheduling based algorithm with an energy-sensitive priority is proposed. For task sets, off-line methods for computing the task maximum required speeds are described, both for rate-monotonic and earliest deadline first scheduling. Also, a run-time speed optimization policy based on slack re-distribution is proposed for rate-monotonic scheduling. Next, an energy-efficient extension of the earliest deadline first priority assignment policy is proposed, aimed at tasks with probabilistic execution time. Finally, scheduling is examined in conjunction with assignment of tasks to processors, as parts of various low energy design flows. For some of the algorithms given in the thesis, energy measurements were carried out on a real hardware platform containing a variable speed processor. The results confirm the validity of the initial assumptions and models used throughout the thesis. These experiments also show the efficiency of the newly introduced scheduling methods.

ACKNOWLEDGMENTS

Foremost, I would like to thank my supervisor, Krzysztof Kuchcinski, for his permanent support and guidance, and for his steady confidence in my work. Special thanks to my co-supervisor, Petru Eles, whose method and attitude towards research often served as a model for myself, ever since the time of my undergraduate studies. I am also very grateful to my other co-supervisor, Per Larsson-Edefors, for his invaluable help in the early stages of this thesis.

During my years as a PhD-student, I had the privilege of working in two successful departments, that both provided a pleasant and friendly environment. Many thanks to all the people in the Department of Computer and Information Science at Linköping University, where I started on the graduate studies path. My best regards also to the people in the Department of Computer Science at Lund University, where I finally completed this thesis. The present ESDLAB members and former CADLAB colleagues, and now friends, have played an important role throughout my studies.

On the more practical side, many thanks to Intel, especially to Bill E. Brown and Atila Alvandpour, for providing not only the hardware platform for the experiments, but also prompt feedback and helpful suggestions. Also, I very much appreciate the help given by Martin Nilsson with the experimental setup and measurements. Many thanks to all the engineers in the technical group, for providing a trustworthy and efficiently working system. My appreciation to the administrative staff, for their excellent handling of all kinds of practical problems.

I would also like to mention that this work was supported initially by WITAS — The Wallenberg laboratory for research on Information Technology and Autonomous Systems — and subsequently by ARTES – A network for Real-Time research and graduate Education in Sweden.

I owe a lot to my friends from all over the world, who were of more help than they know it. My friends back home believed in me and supported me from the beginning; without them I would have never started this journey. The friends I made while in Linköping were of great help during my stay there, and continue to bring joy into my life. Special thanks to Paul, who took the time to read this thesis and to provide insightful feedback. My newest friends from Lund do a good job in reminding me that there are other things in life than writing theses. Thank you all. Finally, to someone very special: this might have nothing to do with the curtains, but thank you for every single moment we spent together...

None of this would have happened without the full support of my family. My deepest gratitude and love to my parents, Mili and Liviu, who always wanted the best for me.

Flavius Gruian
Lund, November 16, 2002

CONTENTS

CHAPTER 1

INTRODUCTION

THIS FIRST CHAPTER starts by presenting the motivation behind the research work making the subject of this thesis, and continues with a succinct and general problem formulation. We then point out the contributions of the thesis to the real-time, design automation, and low power/energy areas. Finally, an overview of our work is given, briefly describing the structure of the thesis.

1.1 Motivation

Energy consumption reduction is becoming nowadays an issue reflected in most aspects of our lives. For digital systems, energy efficiency is an acute problem appearing from the high computational demands in all sorts of applications. The obvious driving force behind addressing energy consumption in digital systems is, at the first glance, the development of portable communication and computation. The consumers demand better performance and more functionality from the hand-held devices, but this also means higher power and energy consumption. Battery life is one of the most important parameters for such devices, directly influencing the system size and weight. At the same time, although battery technology is also developing, its progress is rather slow and cannot keep up with the demands of the modern digital systems. At a deeper scrutiny, there are many areas that would benefit from design methods targeting energy efficiency:

- Space applications is an area where weight has a great impact on cost, because of the limited load of the carrier rockets and shuttles. Energy-efficient design methods yield smaller solar panels and batteries for all kinds of satellites and probes. For example, the electrical power sys-

tem for the *2001 Mars Odyssey* spacecraft accounts for 17% (65kg) of the total mass, planned to operate for 4.6 years. Reducing the energy demands through appropriate methods one could, for instance, add more data acquisition instruments on board. As another example, the energy available in a *Deep Space 2* probe would yield only about two hours talk time on an *Ericsson 628* mobile phone.

- As mentioned before, portable devices, such as PDAs, mobile phones, laptops need to use the battery energy as efficiently as possible. The direct impact of energy-centric design techniques is on system size and/or cost. The indirect impact is the environmental effect of using less batteries (toxic waste) for the same functionality.
- Medical implants within the human body also require energy-efficient designs. Pacemakers and different kinds of regulators would ideally consume only the energy provided by the muscles. There exist already prototypes of simple artificial retinas functioning only based on the energy provided by the incident light.
- Lower energy consumption also means lower average power consumption. Consequently, lower power consumption means less dissipated heat. Finally, heat dissipation directly influences the packaging and cooling solutions for integrated circuits. Energy efficiency brings thus a bonus for all types of digital systems, in terms of lower cost packages and cooling.

With all these applications, there is little doubt that energy efficiency is an important optimization goal in digital system design. Furthermore, note that most of these are in fact time critical systems. This is why the current thesis addresses energy efficiency in the context of real-time systems.

1.2 Thesis Objectives

Timeliness and energy efficiency are often seen as conflicting goals. When designing a real-time system, the first concern is usually time, leaving energy efficiency as a hopeful consequence of empiric decisions. Yet, with the right methods both goals can be achieved. Energy-efficient architectures may be selected while still meeting the timing constraints. Furthermore, with the advent of variable speed processors, scheduling acquired the new dimension of processor speed. Classic real-time scheduling techniques can now be adapted to address both timing and energy through efficient selection of processing speed. The goal of the current thesis is to provide solutions to some of these problems. Briefly, the answer and the *leitmotif* of the thesis lies in trading off speed for energy, whenever timing requirements allow it.

1.3 Contributions

This thesis bridges the gap between the design of real-time and energy-efficient systems by proposing several approaches that address timing and energy consumption in a unified manner. Their energy-efficient solutions, their successful use of stochastic information, and their use of the increasingly popular variable speed processors, makes the described methods unique, to our knowledge. Specifically, the thesis presents:

- A new task level scheduling strategy for tasks with probabilistic execution pattern entitled **Stochastic Scheduling**, first introduced in [Gru01a]. The technique is compared to several alternative approaches, including *compiler-assisted scheduling*. Some practical results on a real platform, showing the potential of our method, are also presented (Chapter 4).

- A new task graph static scheduling approach, based on list scheduling with an energy-centric priority function. The algorithm, entitled **LEneS**, was initially introduced in [GK01]. A less successful, but much simpler scheduling technique, **LS-PS**, is also described and compared to LEneS (Sections 5.4 and 5.3).

- Methods for computing **maximum required processing speeds** for task sets scheduled via the classic *Earliest Deadline First* and *Rate-Monotonic* approaches. The simplicity and efficiency of these methods, partially mentioned in [Gru01b, Gru01a], make them the basic real-time energy reduction methods on variable speed processors (Section 5.5).

- An extension of the classic *Rate-Monotonic Scheduling* (RMS) for including **run-time slack distribution**. This method, first introduced in [Gru01a], is proven to yield the same timing behavior as the classic RMS, but with significant reduction in energy consumption for sets of tasks with probabilistic execution pattern (Section 5.6).

- A new ordering method for sets of tasks with probabilistic execution times, named **Uncertainty-Based Scheduling**. Initially introduced in [Gru01b], this method uses stochastic information in an aggressive manner to lower the energy consumption without affecting the real-time behavior, as proven by practical results on a real platform (Section 5.7).

- Analysis of several system-level design flows for low energy, directed at task graphs, on both fixed and variable speed processor architectures. For fixed speed processors we describe a novel approach for unified binding and scheduling, based on constraint programming, as introduced in

[GK99]. For variable speed processor architectures, we describe two approaches, initially introduced in [Gru00b], that are based on the Simulated Annealing heuristic and our own scheduling methods for low energy (Sections 6.2 and 6.3).

1.4 Thesis Layout

The thesis consists of four parts. The first, preliminary part, contains the theoretical background necessary for understanding our work (Chapter 2) and the models used throughout the thesis (Chapter 3). The subsequent parts focus on describing our own research.

The second part focuses on task level scheduling for variable speed processor architectures (Chapter 4).

The third part presents a large variety of task group level scheduling techniques (Chapter 5). These techniques address both task graphs and task sets. Both static and run-time methods are described for tasks with fixed or probabilistic execution pattern.

The fourth part addresses scheduling in the context of system-level design flow (Chapter 6). Several design flows for low energy are described, both for fixed and variable speed processor architectures.

Chapter 7 concludes the thesis, by presenting a summary and possible future trends in this research area.

CHAPTER 2

BACKGROUND

THIS CHAPTER SETS THE FRAME for our work by reviewing first some basic notions from micro-electronics, real-time systems, and design automation for system-level synthesis. In the second part, we present some of the relevant research related to our own, pointing out the significant differences.

2.1 Power, Energy, and Delay

In order to design power and energy-efficient systems, one has to understand first the physical phenomena that lead to power dissipation or energy consumption. Since today most digital circuits are implemented using CMOS[1] devices, it is important to examine the relations between power, energy, and signal delay in such devices. Although we focus here on CMOS circuits, the power-energy-delay dependencies could be generalized for any technology. Moreover, the methods and techniques we present in this thesis are at an abstraction level that is rather independent of technology. These conclusions are fundamental for our hardware models used in our work.

2.1.1 Power Consumption in CMOS Circuits

The issues discussed in the current section are valid for any CMOS circuit. For the sake of simplicity, consider the CMOS inverter of Figure 2.1. The power dissipated on this inverter can be decomposed into two basic types, static and dynamic [RP96]:

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

[1] CMOS: Complementary Metal Oxide Semiconductor

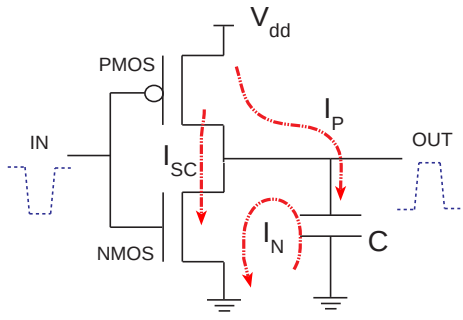


Figure 2.1: CMOS inverter

In the ideal case, CMOS circuits do not dissipate static power, since in steady state there is no open path from source to ground. In reality, there are always leakage currents through the MOS transistors, currents depending on the threshold voltage and on the technological process. These currents yield the static component of the CMOS power consumption. Although the static power is today about two orders of magnitude smaller than the total power, according to [Bor99] the typical chip's leakage power increases about 5 times each generation, and will soon become a significant portion of the total power. Fortunately, by reducing the die temperature one can substantially lower the leakage power. In other words, reducing the dynamic power component and employing better cooling techniques will be even more critical in advanced deep-sub-micron technologies.

The dynamic component of the CMOS power is dissipated during the transient behavior, i.e. during switching between logic levels. For the same CMOS inverter depicted above, if the input switches from one logic level to the opposite, at some moment both the NMOS and PMOS transistors will be open, thus allowing a short circuit current (I_{SC}) to appear between source and ground. With a careful design of the transition edges, this dynamic subcomponent can be kept below 10-15% of the total power consumption.

Most of the power is, thus, consumed by charging and discharging the output capacitance. Consider, for the moment, that the input executes one full cycle from high logic-level to low logic-level and then back to high. During a high-to-low transition, an amount of energy equal to CV_{dd}^2 is drained from V_{dd} through I_p, part of which is stored in the output capacitance while the rest is dissipated on the PMOS transistor. During the reverse, low-to-high transition, the output capacitance is discharged through I_N. Thus, during one cycle a total energy equal to CV_{dd}^2 is consumed. Note that the power consumption

directly depends on the switching activity of the signals involved. In this context, let us define the switching activity, α, as the number of high-to-low transitions during one predefined period. Since we are discussing synchronous circuits, involving a periodic signal (a clock) with frequency f, we can choose the predefined period mentioned earlier, as the clock period. The effective frequency of switches in this case is given by the product: αf. In CMOS circuits, this component of power dissipation accounts for at least 85-90% of the total power consumption [RP96].

From all the considerations made above, we can approximate the power dissipated on a CMOS circuit node using the following formula:

$$P_{CMOS} \approx P_{dynamic} \sim \alpha f C V_{dd}^2 \tag{2.2}$$

This means that the power consumption in a CMOS circuit is proportional to the switching activity, capacitive load, clock frequency, and the square of the supply voltage. All the power and energy reduction techniques try to minimize one or more of these factors. Unfortunately, they are all coupled in some manner. For example the circuit delay Δ, which sets the clock frequency, depends on the supply voltage:

$$\frac{1}{f} \sim \Delta \sim \frac{V_{dd}}{(V_{dd} - V_t)^\gamma} \tag{2.3}$$

where V_t is the threshold voltage and γ is the saturation velocity index. For a sufficiently small V_t we can rewrite the relation between clock frequency and supply voltage as:

$$f \sim V_{dd}^{(\gamma-1)} \tag{2.4}$$

Dependencies, although less obvious, exist between the other factors in the power formula. This means that power minimization is an issue that has to be treated carefully, as detailed below.

Supply voltage (V_{dd}) reduction appears to be the most promising, because of its quadratic dependency to power. A decrease in voltage by a factor of two yields a decrease in power by a factor of four. Unfortunately, the possibility to reduce the supply voltage is limited by several factors, such as design performance and compatibility. When the supply voltage approaches the threshold voltage, the circuit delays become large (Figure 2.2), since the output capacitance is charged and discharged slowly. This sets the lower limit for the supply voltage around $2 \times V_t$ [RP96]. The degradation in performance could be attenuated by working at lower threshold voltages, but this gives rise to other problems. When the threshold voltage becomes very small, the leakage currents through the MOS transistors increase, leading to an increase of static power consumption. Supply voltage reduction works best when combined with techniques which speed up the design, such as parallelization and

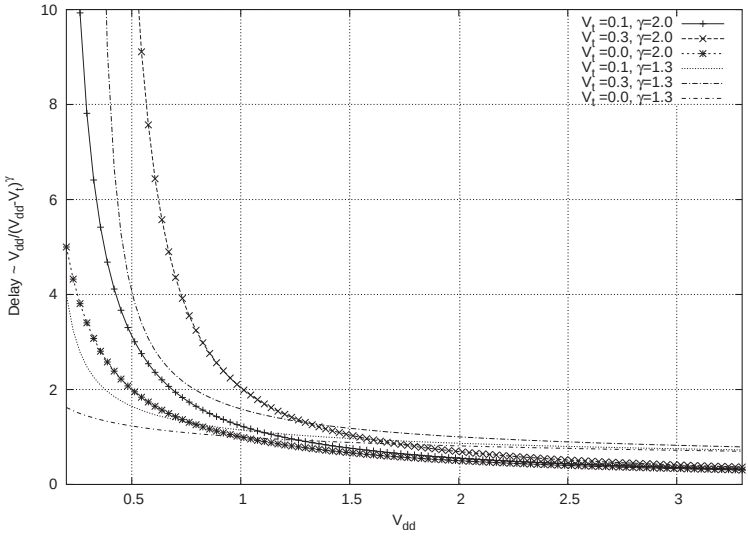


Figure 2.2: The dependency between circuit delay and supply voltage for different threshold voltages V_t and velocity saturation factors γ.

pipelining. The gain in speed can be converted into a decrease in power and energy, by reducing the supply voltage. Furthermore, reducing the supply voltage is a method that can be used for both dynamic and static CMOS circuits.

Lowering the **clock frequency** (f) has direct influence on the design performance, and it is usually combined with speed-up techniques and supply voltage manipulation. Stopping the clock for some circuit parts can be viewed as "component shut-down" at high levels of abstraction. This approach will only work for static CMOS components, since dynamic components loose information if the clock signal is turned off.

Decreasing the **capacitive load** (C) is yet another way to diminish the power consumption. The physical capacitance in a CMOS circuit consists in fact of two capacitances [RP96]: device and interconnect parasitic capacitance. In the past, the device capacitance has dominated over the interconnect parasitic capacitance but, with the process of technology down-scaling, the interconnect capacitance becomes more and more important, and has to be taken into consideration. Physical capacitance can be kept low by reducing the number of gates, using shorter wires and smaller devices. On the other hand, the reduction of device size reduces the current drive of the transistor, making the circuit to operate slower. This performance loss might prevent V_{dd} reduction, which is a more efficient technique for power minimization.

The **switching activity** (α) is much influenced by the data being processed (except for logic styles with pre-charged nodes [RP96]). From Equation 2.2, if a CMOS circuit does not change its state, it virtually does not consume any power. The signal with the highest activity in a circuit is, by far, the clock. Therefore, several power reduction methods focus on lowering the number of clocked nodes in a CMOS circuit. For certain logic styles, an important source of switching activity is glitching, or unwanted and spurious transitions that occur before a node settles down to its final steady-state value.

The switching activity is directly influenced by the choice of number representation in a design. For example, considering a signal oscillating around zero, using a 2's complement representation imposes the switching of most of the bits whenever passing through zero. In this case, a Sign-Magnitude representation could be more suitable. For counters and address buses, a Grey code representation is the best choice. Of course, one should consider also the implications in the functional units design and/or the conversion circuits, if needed, when choosing the number representation.

The work presented in this thesis is based mainly on supply voltage scaling coupled with clock frequency minimization. Switching activity or capacitive load reduction are only indirectly addressed, in the chapter dedicated to system-level synthesis. This comes from the fact that our scheduling problem assumes that the work performed by each task is already decided. More precisely the capacitive load and number of switches is fixed at the moment the tasks are scheduled. Furthermore, the low level aspects of power minimization are somewhat encapsulated in the power versus speed (clock frequency and supply voltage) dependency specific for each processor.

2.1.2 Energy Consumption in CMOS Circuits

As stated in the thesis title, we focus on energy rather than power consumption. Although low power and energy efficiency are often perceived as overlapping goals, there are certain differences when designing for one or the other. Formally, the energy consumed by a system is the amount of power used during a certain period of time:

$$E = \int_0^t P(t)\mathrm{d}t \tag{2.5}$$

Every computation, simple or complex, requires a specific interval of time to be completed. The energy consumption decreases if the time required to perform the operation decreases and/or the power consumption decreases. Thus, compared to the pure power consumption minimization problem, energy reduction includes the time aspect. A technique that would lower the power, but at the same moment increase the computational time, might even lead to

an increase in energy consumption. For example, one could halve the power consumption by only halving the clock frequency in Equation 2.2. At the same time the overall computational time required to perform the same operation would double, leading to no effect on energy consumption. On the other hand, the supply voltage forces an upper limit on the clock frequency. For this reason supply voltage and clock frequency scaling are addressed in conjunction. Note that often lower energy consumption means slower systems. Real-time scheduling and energy minimization are therefore closely related problems, that should be tackled in conjunction for best results.

2.2 Real-Time Systems

Real-time systems are considered to be those types of systems which have to respond to certain stimuli within a finite and specified delay [BW01]. In other words, the correctness such systems depends not only on the logical result of the computations, but also on the time at which the results are produced [SSRB98]. For *hard* real-time systems, it is imperative that responses occur within the specified deadline, any exception leading to a total failure of the system. In *soft* real-time systems response times are important, but the system will still function correctly if some deadlines are occasionally missed.

Although the work presented in this thesis focuses on hard real-time systems, certain algorithms can be easily adapted for soft real-time systems, leading to even greater reductions in energy consumption while keeping a reasonable Quality of Service. Furthermore, we only consider the timing aspects characterizing real-time systems, without involving other specific features concerning, for instance, data sampling, computational accuracy, control efficiency, concurrency, reliability or safety. We also consider that, in principle, if basic real-time scheduling algorithms can be augmented with low energy techniques, then the extensions or enhancements of these algorithms for more complex real-time applications can employ the same low energy techniques. In fact, our energy-efficient methods can be applied once the real-time system timing behavior has been determined, without significantly affecting the design decisions beyond the choice of energy-aware components (such as variable speed processors).

Building upon the properties of the real-time applications, scheduling techniques exhibit several important characteristics [SSRB98, Axe97]:

- **Flexibility**, or the ability of the run-time scheduler to adapt to changes in the task set,
- **Predictability**, or the ability to analyze the run-time behavior by, for instance, estimating the task response time and verifying the timing constraints, and

- **Complexity**, or the volume of computation required to take scheduling decisions. One can distinguish between off-line complexity, when optimality is an issue, or at run-time complexity, when the scheduling overhead is important.

Following these properties, real-time scheduling algorithms can be distinguished into [Bur91, BW01]:

- **Static or dynamic**. A static approach takes all scheduling decisions in advance and requires prior knowledge about the properties of the system, but yields little run-time overhead. For instance, in the case of a fixed set of purely periodic, fixed execution time tasks, it is possible to lay out a complete schedule beforehand, using a *cyclic executive* approach. Yet, there are certain drawbacks with this method. For example, building the executive is NP-hard in the general case and sporadic tasks are difficult to handle [Loc92]. A dynamic scheduling method determines the schedules at run-time, being therefore more flexible. Fixed-priority scheduling (FPS) assigns static priorities to tasks, but schedules them at run-time. Rate-monotonic scheduling (RMS) is from the FPS class, since priorities are assigned according to task rates (fixed). By comparison, in earliest deadline first (EDF) scheduling even the priorities are computed at run-time, the highest priority tasks being those with the nearest deadline. Occasionally, in the real-time community, the scheduling approaches are referred to as static and dynamic depending on the decision regarding task priorities. In this sense FPS is called static and EDF, dynamic.

- **Preemptive or non-preemptive**. In a preemptive scheme certain low priority tasks may be suspended if higher priority tasks need to be executed. Alternatively, in non-preemptive approaches, once started each task finishes its execution without interruption from others. Clearly, preemptive schemes are more flexible, but they also introduce certain time overhead due to the context switches. Intermediate approaches also exist, such as *deferred preemption* schemes, where preemption cannot occur during certain critical time intervals.

- **Centralized or distributed**. A uni-processor system or a multi-processor system with shared memory, where the inter-processor communication time is negligible, are typical examples of centralized systems. In distributed systems, communications take considerable time that have to be accounted for, during feasibility analysis and scheduling.

The work presented in this thesis covers a wide range of scheduling techniques. For variable speed processors, the new dimension introduced by speed selection makes even the scheduling of individual tasks non-trivial, especially

for tasks with probabilistic execution time. At task group level, we look at both static and dynamic scheduling strategies for low energy. In particular, we address static cyclic scheduling for task graphs, FPS in the form of RMS or deadline-monotonic scheduling, and the EDF scheme. Preemption or non-preemption is assumed according the underlying real-time scheduling models. For instance, preemption is required in RMS, but not in some cases of EDF (single-rate, common arrival). In one case we advocate the use of preemption, just to lower the energy consumption when a non-preemptive strategy would also fulfill the real-time constraints (see Section 5.7). In all cases, preemption time overhead is considered to be negligible. We focus mainly on the uni-processor case, yet some of our scheduling methods are designed for heterogeneous architectures (see Sections 5.3, 5.4 and Chapter 6).

2.3 System Synthesis

For best results, the energy consumption issue should be addressed throughout the design cycle of a system. Therefore, it is important to stress the place of our methods in this whole design cycle. Although the thesis focuses mainly on task scheduling, in Chapter 6, we step back and look at how scheduling relates to the system architecture and task mapping. These are essential decisions that have to be taken during *system-level synthesis*, which is only a part of the typical design process, as detailed in the following.

Digital systems can be specified at different levels of abstraction. A traditional view of these levels and the relations between them is captured by the *Y-chart*, introduced in [GK83]. The most commonly used levels of abstraction are the physical, the logic, the register-transfer (RT), the behavioral, and the system level. At system level, the digital system is specified in terms of interacting, concurrent processes, which can be implemented either as hardware components, or in software. The support architecture is given in terms of boards, processors, memories, and buses. The hardware part of a system-level specification can be decomposed in several behavioral specifications, consisting of a sequential algorithm, or a single process. At RT level, the specification is further refined to a description in terms of operations between variables. At this point, the support architecture is composed of ALUs[2], multiplexers, and registers. At logic level, digital hardware is represented in terms of boolean functions implementable by gates, latches, flip-flops, etc. Finally, at physical level, also referred to as circuit level, the hardware is described in terms of transistors, resistors or even silicon areas. Since there is no universally accepted description for these abstraction levels, different research and industry groups might have different views on this matter. It is also hard to

[2]ALU: Arithmetic and Logic Unit

define clear cuts between any two adjacent levels. Very often, the digital system specification consists of several connected specifications, given at various levels of abstraction.

The complexity of certain systems can be dealt with by using high abstraction level descriptions. For a given specification, moving down from system to physical levels, more and more information is added. The specification becomes more detailed, larger and, therefore, harder to handle. We refer to this refinement process as *synthesis* [MLD92], although the term is often used to describe the translation from a behavioral representation of the design into a structural one [GDWL92, DeM94]. In particular, system-level synthesis deals with the formulation of the basic architecture of the implementation [EKP98a]. The most important decisions during the system-level synthesis step, are the allocation of the set of physical processors and the mapping of processes (or tasks) onto this processors while optimizing or fulfilling certain design parameters. For real-time applications, the timing constraints are an essential design parameter. Although, system partitioning, hardware/software co-design, and interconnect-structure design are a few fundamental issues related to system-level synthesis [EKP98a] that affect the energy consumption, we will not address them explicitly in this thesis. More precisely, assuming an already partitioned system, our architecture selection and task mapping approach choose specific task implementations, which might happen to be software or hardware solutions, and certain communication channels, based on the final effect on energy and timing, being oblivious to the lower level implementation choices. Furthermore, our scheduling methods only deal with ordering and speed selection (when necessary), without significantly altering the task implementations in order to affect the internal timing or energy consumption.

2.4 Related Research

In the last few years, the research surrounding low power and low energy systems has become a flourishing area. Initially, most work focused on low power design at the lower abstraction levels. Gradually, as design automation started to be used at higher and higher abstraction levels, energy consumption became an important design metric, overshadowing power consumption at system-level. Finally, the already matured real-time area and low power and low energy design met first in the form of *dynamic power management* and then into *dynamic voltage scaling*, as variable voltage processors made their appearance. Relatively recent surveys of this area can be found in [Jha01, BD99, Ped01].

A large spectrum of compile-time techniques for low energy software also appeared. A few early reviews of such techniques can be found in [TMW94a,

TMW94b, MOI+97]. Some of these are memory energy reduction through efficient data access [PD99, DKV+02], switching reduction through register relabeling and instruction scheduling [Shi01, LLM+01, CC01, PKVI01], or various algorithmic transformations [CBD01, SBD01].

Note that all these techniques are orthogonal with speed scheduling algorithms. Once the tasks have been compiled to yield the lowest energy consumption with the compile-time techniques, task level and task group level speed scheduling techniques can be applied, in conjunction with power management, to additionally reduce the system energy consumption.

2.4.1 Task Level Scheduling

At a first glance, deciding a task level speed distribution might appear as a trivial problem, implicitly solved at task group level. In fact, on fixed speed processors, it is not at all a problem. On the other hand, for variable speed processors, task level scheduling becomes an interesting issue because:

- real processors can only run at a limited, discrete range of speeds, most likely not including the ideal speed for the task in question, and
- an overwhelming majority of tasks exhibit variable (probabilistic) execution pattern, allowing for slowdown at run-time.

Thus, running the task always at only one speed, the one which guarantees meeting the deadline even for the worst case, is not the best solution from the energy consumption point of view. For this reason, several research groups have recognized and addressed the task level speed scheduling problem.

For applications involving variable processing rate, the available time for a given task may vary from instance to instance. Re-evaluating the speed for every instance can, thus, save energy. **Chandrakasan et al. (1996)** present in [CGX96] such an adaptive scheduling method, designed for digital signal processing (DSP) applications, such as MPEG. **Pering et al. (1998)** evaluate in [PBB98] several adaptive speed scheduling methods for DSP applications. The majority of these kind of methods are soft real-time, more concerned with the quality of service, than with meeting hard deadlines. The present thesis focuses on hard real-time systems.

Ishihara and Yasuura (1998) have proven in [IY98] that a task can optimally run at any virtual speed, by running it in two phases at two different real speeds. These speeds are the ones bounding the virtual speed. The authors have also shown that the number and distribution of real available speeds has a direct impact on the energy efficiency of such a schedule. In fact, this dual-phase execution model seems to remain the standard for tasks with fixed execution pattern.

For tasks with variable execution pattern, the speed scheduling problem is even more interesting. **Lee and Sakurai (2000)** describe in [LS00b, LS00a] a method for run-time adjustment of processor speed. The task is divided in regions corresponding to time slots of equal length. At the end of each region's execution, a re-evaluation of the speed is performed in the following way. If the elapsed execution time after a certain number of regions is smaller than the alloted number of time slots, the execution speed can be lowered by an easily determined factor. A similar approach is presented by **Shin et al. (2001)** in [SKS01] as a part of a wider strategy, including task group level scheduling. Another task level scheduling method is described by **Mossé et al. (2000)** in [MACM00], where the regions and their corresponding time slots may have different sizes, while the slack (the difference between the elapsed and the worst case time at a certain moment) is distributed to regions according to various strategies. A more advanced approach, entitled *intra-task scheduling*, is described by **Shin, Kim et al. (2001)** in [SKL01]. In this method, speed re-calculation is performed after control decision affecting the execution path, at the beginning of representative basic blocks. A somewhat similar approach is taken by **Hsu and Kremer (2001)** in [HK01, HK02], where the task is profiled on basic-block level regions. In that work, a single or multiple regions are selected for speed scaling, based on their CPU and memory load. The method introduces a small performance degradation, acceptable only for soft real-time systems. The scheduling methods presented in this paragraph can all be classified as compiler-assisted approaches, since the task must be profiled and pre-processed off-line by, for instance, adding speed switching code snippets in essential points. Briefly, the algorithms must be aware of the internal structure of the task.

In parallel, a different class of scheduling method is emerging, which do not require knowledge about the internal structure of the task. These algorithms can be employed directly at run-time, without the need of a special pre-processing step. Besides our own *stochastic scheduling* method, the only other algorithm from the same class is *PACE* by **Lorch and Smith (2001)**. Presented in [LS01, Lor01], *PACE* was originally designed for soft real-time systems, but its pre-deadline part is based on the same idea as our own algorithm. Namely, it uses the probability distribution function of the task execution pattern to derive optimal schedules in the long run. The method of computing the distribution of the workload over the available processor speeds in *PACE* is slightly different from our own *stochastic scheduling*, but their results are also promising.

2.4.2 Task Group Level Scheduling

One of the first attempts to an energy-sensitive scheduling strategy at task group level, appears to be the method of **Weiser et al. (1994)** presented in

[WWDS94]. Assuming a variable voltage processor, that work examines several speed scaling heuristics, all based on processor load. An extension of that work was presented by **Govil et al. (1995)** in [GCW95]. A more formal analysis of the minimum-energy scheduling problem, for periodic independent tasks on a variable speed processor, is presented by **Yao et al. (1995)** in [YDS95]. By examining the processor load on specific intervals, the authors describe an off-line optimal algorithm and two on-line heuristics for speed scheduling on top of the classical EDF. Furthermore, lower bounds for the on-line heuristics are determined in that work. Remaining one of the most important in the field, [YDS95] triggered the interest for a more formal approach to energy-efficient scheduling.

Hong et al. (1998) describe in [HPS98] a speed scheduling method for periodic and sporadic tasks, as an extension to EDF, also based on the processor load and worst case behavior. In [HKQ+98], the same authors present a non-preemptive speed scheduling strategy for sets of independent tasks with arbitrary arrivals and deadlines. The method is based on a heuristic task ordering step followed by a speed selection step. **Qu et al. (1999)** address in [QKPS99] energy reduction in pipelines, using variable speed stages selected depending on the latency constraints and data size.

Rate-monotonic scheduling (RMS) is first addressed in the context of energy-efficient scheduling by **Lee and Krishna (1999)** in [LK99]. Using a dual-speed processor model, the authors evaluate a speed scaling algorithm based on RMS, composed both of an off-line step and an on-line strategy. Furthermore, the authors investigate the energy efficiency of their method for tasks with probabilistic execution pattern. In [SC99], **Shin et al. (1999)** also present an extension of RMS, this time for multiple speeds processor. Based on tracking the next arrival times, their method adjusts the speeds for the running tasks, whenever there are no tasks waiting. The authors extend the method in [SKS01], to include task level speed scheduling based on compile-time *application slicing*. In another publication [SCS00], the same authors give an algorithm for computing a common maximal required speed for a task set, scheduled in with RMS. Our own RM-MRS method, presented in Section 5.5.2 improves this approach by identifying individual speeds for each task in the set. **Jejurikar and Gupta (2002)** further extend this method in [JG02], for handling task synchronization.

Okuma et al. (1999) introduce in [OIY98], and improve later in [OIY01], an EDF-based two phase scheduling algorithm that assigns speeds to tasks with different arrivals, deadlines, and capacitive load. In the off-line phase, each task is assigned a slot for execution and consequently a maximal speed, problem solved using ILP[3]. At run-time, if a task finishes early, the unused processor time is greedily assigned to the immediately next task instance,

[3] ILP: Integer Linear Programming

lowering its off-line determined speed. On the same line, **Swaminathan and Chakrabarty (2001)** describe in [SC01b] both a MILP[4] approach and a heuristic for determining energy-efficient speed settings for EDF policy. Their approach assumes a dual-speed processor, but takes into account the speed switching latency and energy overhead.

Manzak and Chakrabarti (2000) present in [MC00] an off-line scheduling algorithm for tasks with arbitrary arrivals and deadlines, which again includes switching activity (or capacitive load as in [OIY01]). That approach is based on a heuristic that starts from a feasible schedule and gradually decreases the speed of certain tasks until no feasible schedule can be obtained. The same authors describe in [MC01] a speed selection algorithm based on a similar method, including periodic tasks this time. Their method tries to assign the same lowest possible speed for all tasks, but can still lead to unused processor times. Based on [MC00] and [YDS95], **Pouwelse et al. (2001)** describe in [PLS01] a low complexity sub-optimal scheduling method, implemented on a StrongARM with variable voltage platform. Their *energy priority scheduler* tries to level the speed schedule, by increasing the workload of low speed regions, through occasionally splitting tasks. **Aydin et al. (2001)** present in [AMMMA01] an off-line polynomial-time algorithm for determining the optimal speeds for tasks with fixed execution time, but with different power characteristics, scheduled using EDF.

Quan and Hu (2001) adapt in [QH01, QH02] the algorithm presented by Yao [YDS95] to tasks with fixed priorities. Their new approach still focuses on off-line scheduling for tasks with fixed execution pattern.

Pillai and Shin (2001) describe in [PS01] a few techniques for speed scheduling of periodic task sets, together with experimental results on an AMD K6-2+/ Linux platform. Their off-line method computes a single maximal required speed, based on utilization. Their *cycle conserving* on-line approach is similar to our slack distribution strategy presented in [Gru01a], to which the authors refer in their paper. Unlike in [Gru01a], no formal analysis of the feasibility of their method is presented. Another method they present, *look-ahead DVS* for EDF, tries to defer as much work as possible to as late as possible, while still keeping the deadline, in the hope that tasks will finish earlier than the worst case. Although it does not uses stochastic data for the tasks, their approach is based on a similar idea we base our task level stochastic scheduling method.

Kim et al. (2002) present in [KKM02] a run-time slack distribution strategy for EDF, based on an slack estimation heuristic that considers both accumulated and expected slack.

With the plethora of speed scheduling algorithms, a common evaluation platform becomes essential for the designer to detect the best method in a

[4] MILP: Mixed Integer Linear Programming

specific case. **Shin et al. (2002)** present such an environment, *SimDVS*, in [SKJ+02, KSY+02], together with an performance evaluation of several dynamic voltage scaling algorithms, including some of our own.

Note that few of the approaches mentioned above take into account the run-time possibilities for speed scaling yield by the dynamic slack generated in the system [LK99, SC99, PS01, OIY01, KKM02]. This slack appears from tasks finishing execution early, from the variation of processor demand, etc. Even fewer techniques distribute this slack to the best advantage of the whole system [AMMMA01]. Most of these approaches were developed in fact in parallel with our own slack distribution strategy for RMS. Furthermore, none of these task group level techniques use the stochastic information related to task execution pattern to derive more efficient orders, as in our uncertainty based scheduling approach.

The work mentioned until now focuses on task sets of independent tasks. Scheduling task graphs for energy efficiency was also addressed in the context of distributed systems containing variable speed processors. **Luo and Jha (2000)** describe in [LJ00] a joint scheduling method for periodic task graphs and aperiodic tasks that allows for energy-efficient speed selection at run-time. The static phase dedicated to task graph scheduling, distributes the tasks loosely between arrival and deadline, such that the inter-task slack can accommodate aperiodic tasks or allow for speed reduction at run-time. The same authors improve their methods in [LJ02], with a critical-path analysis based static scheduling method, task execution order refinement, and a improved run-time speed recalculation algorithm. Since several researchers have addressed also architecture selection in conjunction with task graph scheduling, we cover these in the next section.

2.4.3 System-Level Synthesis

For architectures with fixed speed processors, the problem of energy-efficient mapping and scheduling task graphs is somewhat similar to low power behavioral synthesis with multiple supply voltages, as addressed for example in [CP97, SR99, SC98, MC02, KB99, Mar00].

At system-level, the problem of energy consumption on processors with fixed speed was first addressed as a natural continuation of cost minimization or performance oriented co-synthesis. **Kirovski and Potkonjak (1997)** describe in [KP97] a system-level power minimization approach for hard real-time tasks, based on load balancing combined with supply voltage reduction. **Dave et al. (1999)** present in [DLJ99], an energy-aware co-synthesis system, COSYN-LP, based on an energy-conscious clustering technique for minimizing inter-processor communication. **Yang et al. (2001)** present in [YWM+01] a combined static-dynamic scheduling algorithm for task graphs on system containing a fixed speed processor system. The static phase finds a set of

schedules with different lengths and energy consumption, that may have different tasks allocated to processors with different speeds. The dynamic phase chooses the best schedule from the off-line set, based on the required processing rate. Thus, the allocation of tasks to processors and the schedule may change at run-time. **Dick (2002)** presents in [Dic02] several system-level synthesis algorithms targeting, among other objectives, low power consumption. The most representative in the context of our work are perhaps MOGAC [DJ97] and its enhancement, EMOGAC, for designing heterogeneous distributed systems based on a genetic algorithm (GA) for assignment and list-scheduling.

System-level synthesis using variable speed processors only recently made its way into the research community, our work from [Gru00b] being among the first addressing this subject. **Schmitz and Al-Hashimi (2001)** investigate the same problem in [SAH01, SAHE02], using GAs to optimize mapping and scheduling of task graphs. The GA optimizing the schedule it is nested in another GA, optimizing the mapping. In the scheduling GA heuristic, list-scheduling and voltage scheduling are performed consecutively, as two separate steps. In [SAHE02] the authors compare their scheduling strategy to our own LEneS method. **Zhang et al. (2002)** present in [ZHC02] an approach based on a two steps, the first being a combined assignment and ordering while the second being voltage selection for each task in the task graph. The first step is solved by a custom heuristics that balances the load on the processors. The voltage selection step is formulated and solved as an ILP problem.

There are many other approaches to reduce the energy consumption during system-level synthesis, that are based on memory size reduction, memory access minimization, algorithmic transformations, pipelining, etc. We only mentioned here the methods that are in the same class with the ones described in this thesis.

CHAPTER 3

MODELS

THROUGHOUT THE THESIS several different models are used for describing the system under design. The present chapter introduces these models. First we introduce the models used to describe the behavior of the system. Task and task group models are addressed at this point. Then we present the models describing the hardware resources available, such as processors and communication channels.

3.1 Task Model

In the beginning of the system-level design and synthesis, before anything is decided about the implementation, a *task* models a certain part of the functionality of the system. Here we introduce a task as a computational process, yet this model is extensible to communications, as pointed out later.

Definition 3.1. A **task** is a sequential process, a single thread of execution, describing a part of the system behavior.

Furthermore, our task model has the following characteristics:

1. it may communicate with the outside world (other tasks) only before or after its execution, not during execution;
2. it may be preempted at any point, unless the hardware resource imposes some restrictions in this sense.

Although the first point appears to be a restriction, it can be easily fulfilled, since any actual process that needs to exchange information during execution

can be decomposed into a sequence of processes interrupted by communications. When we are talking about a task assigned to a certain processor, *task* refers to an executable specially adapted (compiled, interpreted) to that processor. An *executable task* has additional characteristics:

3. it consists of atomic computational steps, namely clock cycles;

4. preemption can occur between any two consecutive clock cycles, but not during a clock cycle.

The number of clock cycles required by a task depends on the processor executing that task. Note also that point 4 is actually a refinement of the 2^{nd} characteristic of a task.

Occasionally we use the term *task instance* (or *job*) to describe a process, running executable. For tasks with multiple possible execution paths this is an important distinction since two instances of the same task may exhibit different behavior. The interesting distinction for us is difference between the number of clock cycles required by the two instances. Later on we address both tasks using the same number of clock cycles for each instance and tasks with instances that use different number of clock cycles. The latter model is more realistic for tasks whose execution depends on the input data.

3.1.1 Task attributes

Different scheduling methods describe tasks using abstract models of various complexity. A task is usually represented by a n-tuple of values (or attributes). Examining the attributes of a task, one can distinguish between *requirements-related* and *implementation-related* attributes.

The system specification usually contain requirements that refer to *deadlines*, task *period*, and maybe *dependencies* between tasks. All the scheduling methods described in this thesis were designed in principle to work for tasks with *hard deadlines* (see Section 2.2). Yet, most of the scheduling methods we present can be modified for *soft* or *firm deadlines*, as mentioned in specific sections. Since some of our methods were designed for *periodic* tasks, another task attribute used is the task period. Occasionally, deadlines or periods are specified for groups of (usually dependent) tasks instead of for each and every task separately. This is usually the case for systems described as task graphs (see Section 3.2.1).

Other task attributes, such as the computational demand, execution time or execution pattern, are strongly influenced by implementation. The processor type, its operating clock frequency, power consumption, the compiler, and the specific algorithms used are all implementation choices. Once the assignment of tasks to processors is decided, and the tasks brought to an executable format (i.e. compiled and linked) we can talk about the *execution pattern* of

that task. In its simplest form, the execution pattern can refer only to the number of clock cycles needed in the worst case (WCE). Most of the scheduling algorithms in the literature work only with timing information exhibited by the worst case execution. To achieve energy-efficient schedules, we actually make use of more extensive information about the task execution. In our case the proper definition for the execution pattern is the following.

Definition 3.2. The **execution pattern** of an executable task is a probability distribution function, $\eta(x)$, describing the probability that the task will execute for exactly x clock cycles.

Note that $\eta(x) = 0$ if $x \notin [\text{BCE}, \text{WCE}]$, where BCE refers to clock cycles for the best case execution and WCE refers to the clock cycles for the worst case execution. Occasionally, even more information is required about the task. Some scheduling algorithms [SKL01] use a flow graph representation [ASU86] for the task. Such an algorithm will be briefly discussed in Section 4.3.2.

Although by *task* we usually refer to a computational process, communications can be modeled in a similar way. A reasonable assumption regarding communications is that tasks executed on the same processor can communicate instantly, in zero time, using for example shared memory. Inter-processor communication on the other hand takes time, resources (i.e. the bus), and consumes power. Moreover, we assume that communications do not take processor time (i.e. are handled by off-chip controllers and buffers). Our scheduling algorithms can thus treat computational tasks and communications in a similar manner, as long as the assignment step takes care of correctly binding tasks to processors and communications to channels (buses). The energy reduction techniques for communication channels are different than those for processors, but they turn up to be typical speed vs. energy trade-offs after all, as briefly described in Section 3.4.

Finally, in our algorithms a task τ_i is represented by the following n-tuple:

$$\tau_i =< D_i, T_i, \eta_i(x) > \tag{3.1}$$

where D_i is the deadline, T_i the period, and $\eta_i(x)$ the execution pattern of task τ_i. Some of the scheduling algorithms presented in this thesis use only specific characteristics of the execution pattern instead of the full function:

- best case number of clock cycles, $\text{BCE}_i = max\{x\}_{\eta_i(x)>0}$
- worst case number of clock cycles, $\text{WCE}_i = min\{x\}_{\eta_i(x)>0}$
- mean number of clock cycles, $\overline{X}_i = \sum_x \eta_i(x)x$

Note that for tasks with fixed execution pattern (single execution path) $\text{BCE}_i = \text{WCE}_i = C_i$, where C_i is the number of clock cycles required by the task.

3.2 Task Group Models

Usually the system under design is more complex than a single task, and needs to be represented by a group of tasks. The algorithms presented in this thesis, work either on task graphs or task sets, each representation exhibiting its own advantages and drawbacks, as described below.

3.2.1 Task Graphs

Task graphs are frequently used in hardware design automation, being similar to Data Flow Graphs (DFG) or Control Data Flow Graphs (CDFG) [DeM94]. Task graphs emphasize the dependencies between tasks and are suited for describing fairly static and possibly hierarchical structures.

Definition 3.3. A **task graph** is a directed a-cyclic graph $\Gamma = (N, V)$ where each node is a task $\tau_i \in N$ and each edge $(\tau_i, \tau_j) \in V$ specifies that task τ_i must finish its execution before τ_j starts executing.

Conditional execution can be modeled in several ways. One may for example assign guards to some of the edges. Tasks which have guarded incident edges are executed not only when all its predecessors finished their execution but also only if the guard condition is *true*. In this case, each combination of conditions can be considered as representing a distinct non-conditional task graph. Scheduling a conditional task graph can be reduced then to scheduling several non-conditional task graphs and combining the results, as for example in [EKP+98b]. Alternatively, dealing with conditional paths in constraint programming based modeling is a straightforward extension of a non-conditional task graph scheduling, as shown in [Kuc99, SGK99, SGK00, KW01]. Another way to handle conditional execution is to move conditions inside tasks, obtaining tasks with variable execution time, depending on the conditions. The conditional task graph is then modeled as a non-conditional task graph made of tasks with variable execution time. Our static scheduling strategies can handle such task graphs, except the tasks will always be assumed to exhibit their worst case execution pattern. For the reasons described above we will not explicitly use the conditional task graph model in this thesis.

The nodes in the task graph described above can model both computations and communications. In general, the task graph can be viewed as a bipartite graph, where every edge connects a computational node with a communication node. Since we assume instantaneous communication for computational tasks assigned to the same processor, our task graphs can contain, after assignment, edges that connect two computations.

Since the task graph usually describes a repetitive process, an implicit loop is assumed over the whole graph. An iteration starts by executing first those tasks that do not depend on other tasks and proceeds executing the rest

of the tasks once their predecessors finished. The iteration finishes as soon as all reachable tasks finish their execution. Timing constraints resulting from requirements at this level usually refer to the time required for executing one whole iteration. In our model, only when the current iteration finishes, a new task graph instance can begin executing. Correlating this with equation 3.1 containing the n-tuple describing a task, we can say that all tasks will have the same period and deadline, $\forall i \in N,\ T_i = D_i = \mathrm{GraphExecutionDelay}$. By contrast, pipelining techniques use task graph execution models where iterations overlap [Kuc99, SGK00, KW01]. Although our techniques can be adapted to pipeline scheduling, this thesis does not explicitly address pipeline scheduling.

A-cyclic task graphs can also model internal loops, in a hierarchical manner. Each loop can be modeled by a node which is itself an a-cyclic sub-graph. Together with the implicit loop assumed for each a-cyclic graph, this hierarchical modeling can capture nested loops over sub-groups of tasks as in Figure 3.1.

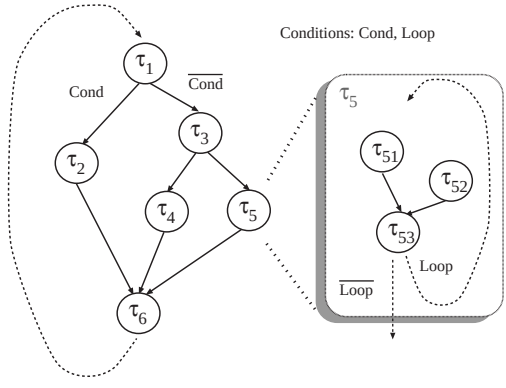


Figure 3.1: A task graph with conditions and loops. The depicted graph is hierarchical, with τ_5 modeling a sub-graph forming a loop. *Cond* controls the execution to take either the τ_2 or τ_3 path. *Loop* controls the repetitive execution of the τ_5 sub-graph.

3.2.2 Enhanced Task Graphs

The scheduling algorithm presented in Section 5.4 uses an Enhanced Task Graph (ETG) model, derived from classic task graphs. Since the resources used are variable speed processors, the execution time of a task can vary depending on the processor clock frequency. Therefore, each task, is more accu-

rately modeled by two important events: start of its execution and the end of its execution. Figure 3.2 details examples of a task graph and its corresponding ETG. In the task graph, depicted on the left, tasks are represented by the circles annotated by pairs of values. A pair consists of the execution time of the task and the identifier of the processor executing the task. The black disks represent communications annotated with the duration and the identifier of the bus/link used for that specific communication. The arcs define the partial order of task execution, which is imposed by the various data dependencies. The ETG is obtained from the initial task graph by substituting each node with a pair of nodes: a start node (the circles), marking the beginning of the execution of that node, and an end node (the grey disks), marking the completion of the task. The execution times of the tasks are now assigned to the internal edges. In our current model, only computational tasks are subject to change their execution delay, while the communication delays remain fixed. The thick edges in the ETG represent the fixed delays. The other edges depict modifiable delays, and the associated numbers define their minimal values. The information regarding the assignment of tasks to processors is also transferred to the ETG.

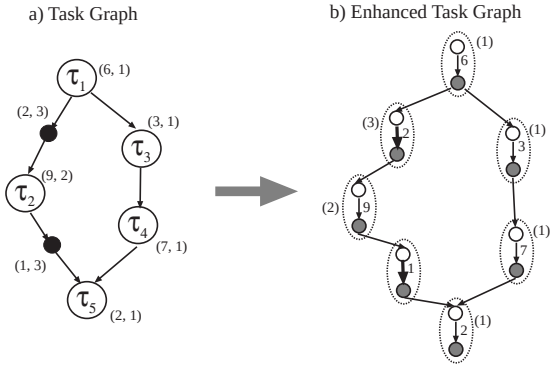


Figure 3.2: A Task Graph and its corresponding Enhanced Task Graph

3.2.3 Task Sets

The task set model emphasizes the timing requirements for each task. They are more suited for modeling dynamic systems employing priority based runtime scheduling. In our task set model, there is no data or control dependency between tasks, so there is no communication taking place between the tasks. Moreover, our scheduling algorithms focus on the case when the tasks run on

a single processor. Tasks are still represented using the triple from equation 3.1, with the mention that for any task the deadline is not larger than the period. Occasionally our algorithms use more restricted models (i.e. deadlines equal to periods), as explicitly stated in the corresponding sections.

An important characteristic of a task set is the processor *utilization* imposed by the task set. Since we are dealing with variable speed processors, the utilization is always defined for the reference speed, that is usually the maximum processor speed. Furthermore, since the tasks have variable execution pattern, we distinguish between *worst case utilization* and *actual utilization*. With the notations introduced in Section 3.1.1 we give the following definitions:

Definition 3.4. Given a task set of N tasks $\{\tau_i = (D_i, T_i, \eta_i(x))\}_{i=1...N}$ the **worst case utilization** is computed as:

$$U_{\text{WCE}} = \frac{1}{f_{ref}} \sum_{i=1}^{N} \frac{\text{WCE}_i}{T_i} \tag{3.2}$$

where f_{ref} is the reference clock frequency. The **actual utilization** over H hyper-periods is computed as:

$$U(H) = \frac{1}{f_{ref}} \frac{1}{H} \sum_{h=1}^{H} \sum_{i=1}^{N} \sum_{j=1}^{\lceil T_i/T_H \rceil} \frac{X_i^{hj}}{T_i} \tag{3.3}$$

where X_i^{hj} is the number of cycles executed by the j^{th} task instance of task τ_i in hyper-period h. T_H is the duration of the hyper-period, computed as the least common multiple of all task periods, $lcm\left(\{T_i\}_{i=1...N}\right)$.

Note that the actual utilization can be accurately computed only after execution, when all the execution patterns for all task instances are known. For a sufficiently long interval of time, an estimate of the actual utilization is the expected value of utilization $\overline{U}$:

$$\overline{U} = U(t)\big|_{t\to\infty} \approx \frac{1}{f_{ref}} \sum_{i=1}^{N} \frac{\overline{X_i}}{T_i} \tag{3.4}$$

3.3 Processor Models

A basic assumption for modeling resources is that *a task can exclusively use only one resource at a certain time*. This can perfectly model for example tasks implemented as software running on a processor or ASICs implementing a

single task. Escaping our model are the cases where several tasks are implemented on a single ASIC, executing in parallel, sharing functional units. Yet, since our scheduling techniques focus mainly on architectures composed exclusively of processors running software tasks, the above mentioned assumption is reasonable enough.

Other assumptions are that the *processors are synchronous designs* (for which it makes sense to talk about clock frequency) and given a certain clock frequency, *all clock cycles require the same amount of energy*, irrespective of the computational requests. This means that the power consumption of a processor for a given clock speed remains almost constant and independent on the running task.

In practice the power difference between various instruction can vary considerably ([TMW94b]). Yet, complex tasks, as encountered in this design phase, consist of rather heterogeneous mixes of instructions, meaning that the variation in power consumption over a relatively small number of clock cycles is statistically insignificant [RJ98]. We make one exception from this rule for the NOP instruction, which is assumed to consume only around 20% of the normal (average) power [SC99]. A processor constantly executing NOPs will thus use only 20% of the energy it would use by doing some useful work.

The *constant power per clock cycle* assumption given above may also be inaccurate at task group level, if the tasks exhibit very different computational requirements. For example an intensive data processing tasks (such as shuffling data in memory) may consume more power than a control task. Although our algorithms do not explicitly distinguish between such tasks, they can be adjusted to accommodate these cases by assigning different weights to tasks accordingly.

3.3.1 Fixed Speed Processors

In the first part of Chapter 6 we address architecture selection for minimal energy, using fixed speed processors as resources. This type of resource models processors operating at constant clock frequency and voltage, thus dissipating a constant amount of power. They also have at least two power modes: ACTIVE, when they execute instructions, consequently consuming significant power, and SLEEP, when instruction processing is stopped and the power consumption is much lower. The processor can, thus, enter the SLEEP state whenever it is idle. Power modes are a common feature today for all types of processors, and especially embedded processors (see Example 3.1). Efficiently selecting power modes for resources, at run-time, is an area usually referred to as Dynamic Power Management (DPM). The Advanced Configuration and Power Interface (ACPI) specification is a joint effort by Compaq, Intel, Microsoft, Phoenix, and Toshiba towards a standard API supporting

DPM [Com02]. ACPI 2.0 is implemented in most Windows™ operating systems and in the Linux kernel version 2.4. However, DPM is not the focus of this thesis.

Example 3.1 (Intel® 80200 Power Modes):
The Intel® 80200 processor [Int01] has three[1] power modes: ACTIVE, IDLE, and SLEEP. The ACTIVE mode is the normal operation mode. The other two are low power modes, selectable by programming CP14, register 7. In the low power modes, parts of the processor are shut down to achieve a low power consumption. In IDLE mode, the processor stops fetching and executing instructions but keeps its Phase-Lock Logic (PLL) running and maintains its architectural state. Interrupts re-instate the ACTIVE mode almost instantly. In SLEEP mode, the PLL is stopped and the architectural state lost. Re-entering the ACTIVE mode from SLEEP takes considerably more time, since it requires a full reset sequence and restoring the architectural state. The following table summarizes power modes' data for an 80200 running at 733MHz and 1.5V:

Power Mode	ACTIVE	IDLE	SLEEP
PLL	On	On	Off
Architectural State	kept	kept	lost
Latency to ACTIVE	-	~ 1 clock cycle	> 2000 clock cycles
Core current (Icc)	720mA	190mA	N/A

For real processors, switching between power modes exhibits a certain latency and consumes additional energy. Our algorithms do not consider power mode switching latency and energy overhead explicitly. However, if these are significant compared to the task delay and energy, the scheduling algorithms can be extended to include the mode switching overhead.

3.3.2 Variable Speed Processors

Most of the scheduling techniques described in this thesis refer to architectures using variable speed processors. For these type of processor the clock frequency and supply voltage can be adjusted at run-time. Scheduling for these processors requires not only deciding on the time moments for certain events but also deciding on the clock frequency and supply voltage for each time interval. Therefore, the techniques addressing variable speed processors are usually known as *Dynamic Voltage Scaling* (DVS) techniques.

The ideal model of a variable speed processor is able to run at a continuous range of clock frequencies and voltages. Moreover, since the goal is to

[1] A fourth power mode, DROWSY, initially present in the 80200 description is currently despecified.

consume as little energy as possible, for a given clock frequency there is an unique optimal supply voltage. This is the lowest voltage for which the circuit delay still permits the given clock frequency (see Section 2.1). The (optimal) supply voltage, power consumption and cycle energy are therefore uniquely determined by the clock frequency. In the following, instead of using the absolute clock frequency as a basis to describe the processor clock and supply settings, we will use the term *processor speed*. The processor speed is the relative value of the clock frequency f compared to a reference clock frequency f_{ref}, which is usually also the maximal clock frequency:

$$s_f = \frac{f}{f_{ref}} \tag{3.5}$$

A processor running at half speed will thus have the clock frequency half the reference frequency, with all the resulting consequences in terms of supply voltage, power and energy consumption. Using the equations introduced in 2.1, the voltage and power consumption at frequency f can be written in terms of their reference values:

$$V_f = V_{ref} s_f^{\frac{1}{\gamma-1}} \quad P_f = P_{ref} s_f^{\alpha} \tag{3.6}$$

where γ, the saturation velocity index, is approximated by 2.0 in the classical MOSFET models. More accurate models [RP96] show that γ is closer to 1.3, yet this does not affect the dependency types, which remain non-linear. Since $\alpha = 1 + \frac{2}{\gamma-1}$, power consumption is a convex function of speed. In fact, since tasks execute clock cycles, it makes more sense to talk about clock cycle energy for a certain frequency, e_f, than to talk about power consumption at a frequency, P_f:

$$\begin{aligned} e_f &= \int_a^{a+\frac{1}{f}} P_f \mathrm{d}t = P_{ref} \int_a^{a+\frac{1}{f}} s_f^{\alpha} \mathrm{d}t = \\ &= e_{ref} f_{ref} s_f^{\alpha} \frac{1}{f} = e_{ref} s_f^{\alpha-1} = \\ &= e_{ref} s_f^{\frac{2}{\gamma-1}} \end{aligned} \tag{3.7}$$

For $\gamma = 2$, the clock cycle energy depends quadratically on speed. This is the commonly used model in DVS research and also the model we use in this work. As proven by our experimental results presented later on, the model is accurate enough for our algorithms to perform efficiently on a real platform. Finally, the energy of a task that executes a certain number cycles N_f at each frequency f taken from a set F can be computed as:

$$E_{ideal} = \sum_{f \in F} N_f e_f = e_{ref} \sum_{f \in F} N_f s_f^2 \tag{3.8}$$

We denoted the energy in equation 3.8 by E_{ideal} since it does not include the effects of speed switching on energy. The ideal model of the variable speed processor can switch between clock frequencies and supply voltages without any time or energy overhead.

A more realistic model of a variable speed processor has to address two real phenomena:

- the range of available speeds is limited and discrete
- switching speeds has a time and energy overhead

We will now look at these problems in more detail.

Discrete Range of Speeds

In practice, the range of available speeds on a processor can only be discrete (see Section 3.3.2 for examples). This comes from the fact that the core clock frequency is generated internally by a Phase-Locked Loop (PLL) or Delay Loop Logic (DLL) using an external, fixed frequency clock. The internally generated frequency is a multiple of the external frequency. The supply voltage follows then the steps imposed by the available clock frequency steps.

Even on a discrete range of speeds, one can simulate a continuous range of speeds for long enough tasks. The virtual clock frequency can be obtained by running different parts of the task at different real clock frequencies. To simulate a desired frequency f_v, it is enough to use two real frequencies, one higher $f_H > f_v$ and one lower $f_L < f_v$. A task requiring N clock cycles will then run N_H clock cycles at f_H, and $N_L = N - N_H$ clock cycles at f_L. To know exactly how many clock cycles to run at each real frequency, it is enough to make sure that the time covered by running the N clock cycles at f_v is equal to the time covered by running N_H at f_H plus N_L at f_L. In other words, we have to solve the following equation in N_H:

$$\frac{N}{f_v} = \frac{N_H}{f_H} + \frac{N - N_H}{f_L} \tag{3.9}$$

Finally, if we take into account the fact that the number of clock cycles has to be an integer, we obtain the following solution for equation 3.9:

$$N_H = \left\lceil N \frac{1/f_v - 1/f_L}{1/f_H - 1/f_L} \right\rceil \quad N_L = N - N_H \tag{3.10}$$

Note that the virtual frequency obtained using this splitting may be slightly higher than the desired virtual frequency. This difference might be significant only for very short tasks, using a small number of clock cycles and for a large discrepancy between the real frequencies. If switching between the two

frequencies takes a relatively important interval of time $t_{H \to L}$, one may take this into calculation. Equation 3.9 becomes then:

$$\frac{N}{f_v} = \frac{N_H}{f_H} + \frac{N_L}{f_L} + t_{H \to L} \tag{3.11}$$

From the energy consumption point of view, as shown in [IY98], it is optimal if one chooses f_H and f_L to be the closest bounding frequencies for f_v. Using this execution model, for a discrete range of speeds, the real energy function becomes then a linear approximation of the ideal energy function. Between any two adjacent real frequencies, the energy varies in a linear manner (see Figure 3.3). The more real speeds are available, the better the approximation of the ideal energy function.

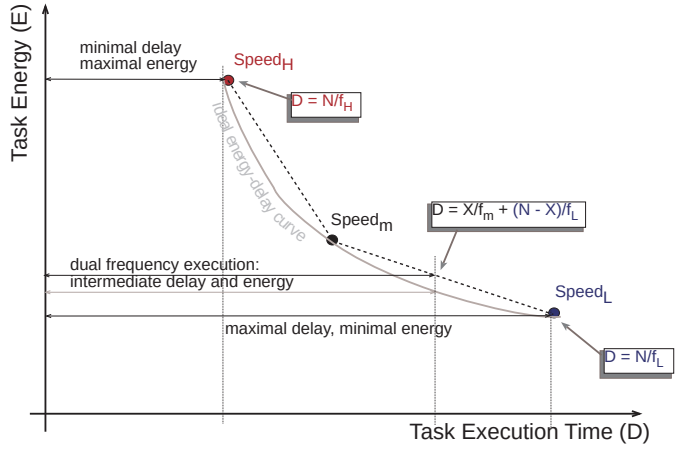


Figure 3.3: Energy-delay dependency for a three-speed processor, $Speed_L, Speed_m, Speed_H$. Intermediate speeds are possible by executing parts of the tasks at the available speeds.

Speed Switching Overhead

In modern processors, the clock signal accounts for a large part of the power consumption ([RP96] reports up to 40% of the processor power is consumed by the clock). To reduce jitter, noise and power consumption, the high speed core clock signals are today generated on-chip, using Phase-Locked Loops (PLL) or Delay Loop Logic (DLL). An external slow, and thus low power, clock signal is used by the on-chip PLL/DLL to generate the fast core clock. Changing the frequency of the PLL output signal has certain latency, since the loop has

to adjust to the new frequency. This means usually that during the time in which the PLL re-locks, the processor has to stall (see Example 3.2). So there is a certain time overhead when switching between speeds.

The voltage supply design may also contribute to the speed switching overhead. This happens for the architectures where the processor must stall until the supply voltage stabilizes. Of course, if both the supply voltage and clock frequency change simultaneously, only the slowest of the two operations will affect the switch latency. Yet, many processors are designed such that they can keep executing instructions at constant rate while the voltage switches between two levels (assuming also the lowest voltage allows the working clock frequency).

Example 3.2 (Intel 80200 Speed Switching Overhead):
Intel 80200 is based on the XScale architecture and has an ARM core. Its core clock frequency can be changed at run-time by writing in a certain control register. Once a value has been written there, the processor stalls, and the DLL re-locks on the new frequency. The latency of this operation is around 2000 clock cycles as reported in [Int01]. In Figure 3.4, we show the profile of a speed switch for the 80200. First the clock frequency is changed to a lower frequency. The power consumption approaches zero for the duration of the DLL re-locking on the new frequency. Then the voltage is adjusted accordingly. (see Appendix B for a detailed description of the system used) While the voltage changes, the processor executes instructions, so the latency is given by the DLL re-lock interval. From the figure, this can vary around $30\mu s \ldots 35\mu s$.

Slightly improved designs, that can change their clock frequency in one clock cycle, can be imagined. For example one could use two PLLs/DLLs, one generating the operating clock, and the other adjusting to a new frequency. When a new frequency is requested, the secondary PLL will begin re-locking on the new frequency, while the processor will continue executing using the primary PLL clock. As soon as the secondary PLL re-locked, the two PLLs swap roles. Thus, the processor would not stall while the clock switches to the new frequency. Note that this solution still exhibits a certain latency, since the processor will continue using the old frequency for some time.

Alternatively, one may use three PLLs, locked on consecutive frequencies. The one operating the processor would always be the one locked on the middle frequency (say PLL_2). Whenever a speed-up or a slow-down is required, the processor can instantly switch to using one of the other PLLs (say PLL_3). Then the third PLL (PLL_1) would need to re-lock on a new frequency, such that PLL_3 will be the one generating the middle frequency, while PLL_1 and PLL_2 generate the bounding frequencies. Although the speed change is instant in this case, there would still be a limitation on the speed switch frequency. The time between two speed switches has to be at least the time required by a

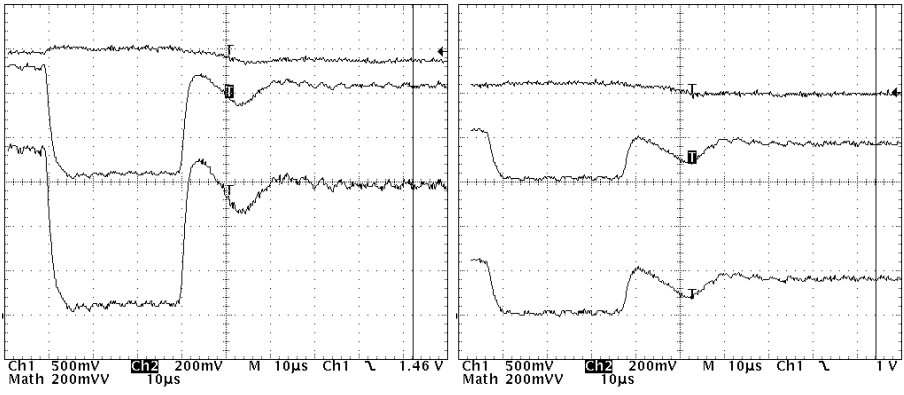


Figure 3.4: Oscilloscope Trace of Speed Switching for an Intel 80200. The three signals are, from top to bottom, V_{cc} the core voltage (Ch1, 500mV/div), I_{cc} core current (Ch2, 200mA/div) and P_{cc} core power (Math, 200mW/div). The horizontal scale is set to $10\mu s$/div. **Left:** switching from 733MHz@1.5V to 666MHz@1.4V **Right:** switching from 400MHz@1.1V to 333MHz@1.0V

PLL to re-lock on a new frequency. In any case, these solutions can bring a real DVS processor very close to an ideal one.

Furthermore, there are solutions which use voltage controlled clock generators and do not stall the processor while changing the clock frequency (see Example 3.3). In these solutions, the clock frequency follows the supply voltage, which requires certain time to switch between levels. Thus, even for these architectures, switching between processor speeds has certain time overhead.

Example 3.3 (lpARM Speed Switching Overhead):
The lpARM processor is a 0.6μ technology DVS-capable Low Power ARM processor developed at UC Berkeley [BPSB00, PBB00]. It includes an ARM8 core running at a clock frequency produced by an on-chip voltage-controlled oscillator (VCO). On lpARM, a speed switch from 5MHz@1.2V to 80MHz@3.8V takes a $70\mu s$, as reported in [BPSB00]. The same authors report in [PBB00] that a simulated lpARM takes $25\mu s$ to switch from 10MHz@1.1V to 100MHz@3.3V. Due to the particular design of the lpARM, the processor can continue executing instructions while switching speeds. Although the transition between speeds is not instantaneous, the property that the processor can continue operating while switching, makes the actual latency much smaller than the speed switch. Using a rough approximation, the actual latency (of gradually getting to the right speed instead of instant switching) would be around half the time to switch between two speeds. Finally, the actual switching

time overhead with the data given in [BPSB00] would be around $35\mu s$ from 5MHz@1.2V to 80MHz@3.8V.

Real processors do exhibit latency in a speed switch. Yet, depending on the number of speed switches relative to the performed tasks, there are cases when the time overhead may be small enough to be considered negligible. Most of the algorithms and simulations presented in this thesis assume that the speed switching overhead is negligible. Yet, they do not necessarily require zero overhead and, furthermore, they can be extended to consider non-zero switching latency.

To summarize, unless explicitly stated, the processor models used in the experiments and algorithms presented in this thesis exhibit a discrete range of supply voltage and zero-latency speed switching.

Examples of Variable Speed Processor Solutions

Variable speed processors are making their first steps in real applications. Here we briefly describe five solutions, implemented by various embedded processors. Four of them are industry developments by Transmeta, AMD, and Intel. The fifth solution is the result of a academic research project at UC Berkeley. The Transmeta and AMD approaches include both hardware features and software managers for power efficiency. This makes them rather transparent to the software developer. The Intel and Berkeley solutions are focused on the hardware support, offering full control to the software developer.

Example 3.4 (Transmeta Crusoe's LongRun):
Crusoe is a Transmeta processor family (TM5x00), with a VLIW[2] core and x86 Code Morphing^{TM} software that provides x86-compatibility. Besides four power management states, these processors support run-time voltage and clock frequency hopping. Frequency can change in steps of 33MHz and the supply voltage in steps of 25mV, within the hardware's operating range. The number of available speeds depends thus on the model. The TM5600 model for example, operates in normal mode between 300-667MHz and 1.2-1.6V [Fle01], meaning eleven different speed settings. The corresponding power consumption varies between 1.5W and 5.5W. The speed is decided using feedback from the Code Morphing algorithm, which reports the utilization. The LongRun manager employs this feedback to compute and control the optimal clock speed and voltage. Note that this is a fine grain control, transparent to the programmer. The algorithms we present in this thesis require direct control over the processor speed, and would substitute or augment LongRun. Nevertheless, the Crusoe architecture is a successful example of a variable

[2]VLIW: Very Long Instruction Word

speed processor, widely used in low power systems. A comparison with a conventional mobile x86 processor using Intel SpeedStep (see Example 3.6), running a software DVD player, reported in [Fle01], shows the TM5600 to consume almost three times less power than the mobile x86 (6W for TM5600 vs. 17W for the mobile x86).

Example 3.5 (AMD's PowerNow!):
AMD introduced PowerNow!, a technology for on-the-fly independent control of voltage and frequency. Their embedded processors from the AMD-K6-2E+ and AMD-K6-IIIE+ families are all implementing PowerNow!. According to [AMD00], AMD PowerNow! is able to support 32 different core voltage settings ranging from 0.925V to 2.00V with voltage steps of 25mV or 50mV. Clock frequency can change in steps of 33MHz or 50MHz, from an absolute low of 133MHz or 200MHz, respectively. The voltage and frequency changes are controlled through a special block, the Enhanced Power Management (EPM) block. At a speed change, an EPM timer ensures stable voltage and PLL frequency, operation which can take at most $200\mu s$. During this time, instruction processing stops. A comparison with a Pentium III 600+ using Intel SpeedStep (see Example 3.6) shows that the AMD's processor with PowerNow! consumes around 50% less power than the Pentium with SpeedStep (3W for AMD-K6-2E+ vs. 7W for Pentium III 600+).

Example 3.6 (Intel's SpeedStep):
Intel's SpeedStep is probably the earliest solution from the ones presented here, and consequently the weakest one. Besides normal operation, SpeedStep defines the following low power states: Sleep, Deep Sleep, and Deeper Sleep. It only specifies two speeds, orthogonal with the power states, a *Maximum Performance Mode* (fast clock, high voltage, high power) and a *Battery Optimized Mode* (slower clock, lower voltage, power efficient). For instance, Mobile Intel Pentium 4 - M Processor [Int02] uses 1.3V and 1.2V for the two speeds, while the clock frequencies are 1.8GHz (or as low as 1.4GHz depending on the model) and 1.2GHz respectively. The power consumption of the Mobile Pentium 4 is anywhere between 30W (*Maximum Performance* 1.8GHz) and 2.9W (in Deeper Sleep, 1V). Switching between speeds requires going to Deep Sleep, change the voltage and frequency, and wake up again, procedure which requires at least $40\mu s$.

Example 3.7 (Intel's XScale):
Intel has recently come out with XScale, an ARM core based architecture that supports on-the-fly clock frequency and supply voltage changes [Int00]. The frequency can be changed directly, by writing values in a register, while the voltage has to be provided from and controlled via an off-chip source. The XScale core specification allows 16 different clock settings, and four different power modes (one ACTIVE and three other). The actual meaning of these

settings are dependent on the Application Specific Standard Product (ASSP). For instance, the 80200 processor supports clock frequencies up to 733MHz, adjustable in steps of 33-66MHz. The core voltage can vary between 0.95V and 1.55V. For our specific test system with the 80200, described in Appendix B, there are only 6 speed settings (from 733MHz@1.5V down, in 66MHz/0.1V steps). Switching between speeds takes around $30\mu s$, as detailed in Example 3.2. The power consumption for the 80200 (core plus pin power) is anywhere between 1W (at maximum speed) and a few μW (in sleep mode).

Example 3.8 (UC Berkeley's lpARM):
The lpARM processor, developed at UC Berkeley, is a low power, ARM core based architecture, capable of run-time voltage and clock frequency changes. The prototype described in [BPSB00] (0.6μ technology) is, reportedly, able to run at clock frequencies in the 5-80MHz range, with 5MHz increments. The supply voltage is adjustable in the 1.2-3.8V range. See Example 3.3 for details on speed switching in lpARM.

These examples show that variable speed processors become more and more common. They usually have a discrete range of voltages and/or clock frequencies, and exhibit latency when switching between speeds. Voltage scheduling algorithms targeting energy efficiency have to take into account these characteristics of real processors. Although some of these real processors already provide voltage scheduling methods they are not suitable for hard real-time tasks. The scheduling algorithms presented in this thesis make good use of the hardware capabilities of such processors, especially in hard real-time environments.

3.4 Communication Models

In the task graph model, computational tasks may pass information to their successors. We assume that communication between two tasks situated on the same processor takes no time and energy at all. Whenever tasks situated on different processors have to communicate, the time and energy required depend on several factors:

- the amount of information to be transmitted: in principle the more information one has to communicate, the more time and energy it is required;
- the way the data is encoded: redundant data encoding may help to reduce for example the bus switching activity, but it may take longer time to communicate the same amount of information [SB95, MOI96, BDM$^+$97b, BDM$^+$97a];
- the type of channel used to communicate: this refers to both the physical characteristics (width of the bus) and the transmission protocol.

In our algorithms we consider that time and energy for a certain communication are fixed for a given communication channel. Other research explicitly addresses encoding schemes and bus protocols for reduced energy and power [SB95, MOI96, BDM+97a]. Furthermore, we assume that the time and energy consumption for each communication on every channel are provided to our algorithms as input data, along with the task attributes (such as period, deadline, and WCE).

With respect to scheduling and resource assignment, communications are treated similarly with tasks (as mentioned in Section 3.1.1), in the sense that only one communication can use a certain channel at a certain moment. Finally, in our system model, a certain communication cannot overlap with its source or destination computational task. Although communications can occur while a processor executes other computational tasks, they must be information transfers between tasks that are not currently executing.

CHAPTER 4

TASK LEVEL SCHEDULING

SCHEDULING FOR LOW ENERGY can be addressed either at individual task level or at the level of tasks group. At group level, efficient scheduling strategies are usually application specific. At task level, the scheduling methods are more general, since they are oblivious of the actual system. In this chapter, we address task level scheduling strategies, while task group level scheduling makes the subject of the next chapter.

At task level, the scheduling decisions concern the processor speeds or power modes sequence used while executing that task. Considering that a task τ needs to execute alone during an allowed time interval A, the scheduling problem consists, in principle, of mapping task regions (in clock cycles) to processor speeds. Using the processor speed as introduced by equation 3.5, we define the speed schedule of a task:

Definition 4.1. Given a task τ requiring X clock cycles to execute on a processor with S speeds, a **(speed) schedule** for τ is an assignment of processor speeds to task clock cycles (a mapping $\xi : \{1, \ldots, X\} \rightarrow S$ where $S \subseteq \Re^+$) such that the task execution time (computed as $t_\tau = \sum_{i=1}^{X} 1/[f_{ref}\xi(i)]$) respects the real-time constraints imposed through the allowed execution time A.

This definition is valid both for real processors, with discrete and limited range of speeds, and ideal processors, without speed constraints. Note also that it does not explicitly state the relation between the task execution time and the allowed execution time. For hard deadlines $t_\tau \leq A$ while for soft deadlines it may occasionally happen that $t_\tau > A$.

Actually, the scheduling problem is not fully solved unless one decides what the processor does during the entire allowed execution time. If the task finishes before the deadline, the processor may for for instance either

continue to execute NOPs, or stop executing instructions and switch to a low power mode.

4.1 Slower Execution vs. Shutdown

First, it is important to realize that, whenever possible, it is more energy efficient to run the processor slower than to execute as fast as possible followed by processor shutdown. Consider a task τ that executes X clock cycles during an allowed time A. Note that in order for the task to be finished during interval A, $X/f_{ref} \leq A$. In the first case, the task executes as fast as possible (at a clock frequency f_{ref}) and then the processor is shut down. By *shut down* we understand a low power mode, during which the processor cannot execute instructions. If we consider the power consumption in the low power mode small enough and the overhead for mode switching as negligible, the energy consumption for this scenario is (using equation 3.8):

$$E_{shutdown} = Xe_{ref} \tag{4.1}$$

In the second case, the processor speed is lower such that the task execution extends over the whole interval A. This would mean that the minimum required frequency is $f = X/A$. Using equations 3.5, 3.7, 3.8, and 4.1, we can compute the energy consumed in this scenario:

$$\begin{aligned} E_{slower} &= Xe_f = Xe_{ref}s_f^{\frac{2}{\gamma-1}} = \\ &= E_{shutdown}\left(\frac{X}{Af_{ref}}\right)^{\frac{2}{\gamma-1}} \end{aligned} \tag{4.2}$$

As discussed in Section 3.3.2, the power of the right term is always greater than 1. The term raised to power is on the other hand at most 1, as noted in the beginning of this section. Finally this means that $E_{slower} \leq E_{shutdown}$, or it is always better to run the task slower than to finish it early and then go to a sleep state.

Example 4.1 (Slow execution vs. shutdown on Intel 80200):
Consider a task that takes around 37ms to execute at the highest speed (733MHz at 1.5V) on an Intel 80200. The deadline is 100ms. One can either execute it as fast as possible and then go to a low power state (IDLE) or execute it slower (333MHz at 1.0V) and go to a low power state. Figure 4.1 depicts an oscilloscope trace for the first case, while figure 4.2 a trace for the second. The energy consumption turns out to be more than twice higher in the first case. This measurement confirms the fact that running slower when possible is a better strategy than running as fast as possible and switch to a low power mode.

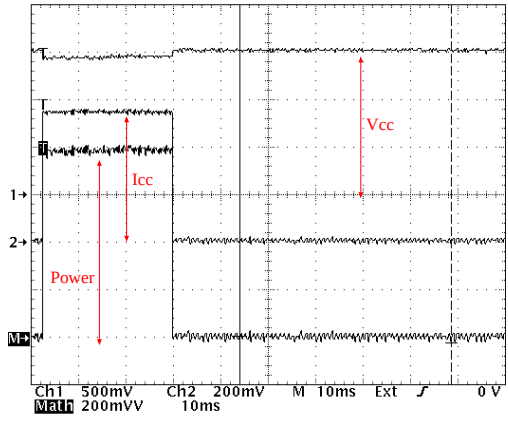


Figure 4.1: Oscilloscope trace of V_{cc}, I_{cc} and their product, the power (200mW/div), when the task executes as fast as possible. Energy, the area under power curve, is approximately 21.6mJ

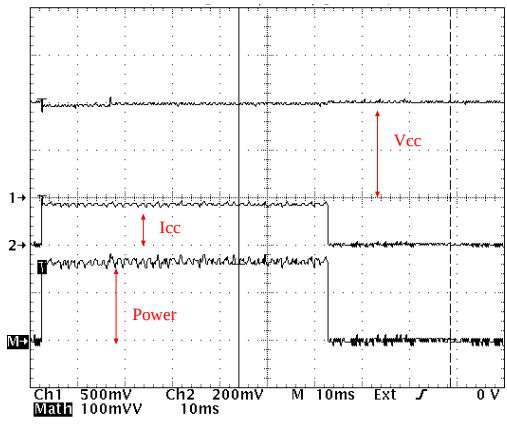


Figure 4.2: Oscilloscope trace of V_{cc}, I_{cc} and their product, the power (100mW/div), when the task executes as slow as possible. The energy consumed is approximately 9.6mJ

In practice, this difference is even more significant, from the following reasons. The power consumption in the low power mode is actually not zero, which makes the processor consume energy even in the low power mode. Switching between power modes is likely taking more time and energy than switching between processor speeds. This makes the variable voltage scheme

a more effective energy reduction technique than the simple dynamic power management (DPM).

4.2 Unique Execution Pattern

Tasks with all instances requiring the same number of clock cycles are what we call tasks with unique execution pattern. In other words, for a fixed processor speed, the execution time is constant and the same for all instances, regardless of the input data. Using the notations from Section 3.1.1:

Definition 4.2. A task τ with execution pattern $\eta(x)$ has a **unique execution pattern** if and only if $\exists C$ such that $\eta(C) = 1$ (and $\forall x \neq C$, $\eta(x) = 0$).

A speed schedule for a task with fixed execution pattern would, in principle, consist of an assignment of different speeds to the different clock cycles up to C, as given in Definition 4.1. We denote the execution interval of each clock cycle i by A_i ($= 1/[f_{ref}\xi(i)]$). The energy consumption of a single clock cycle can be rewritten using equations 3.5 and 3.7 ($\gamma = 2$) as:

$$e_i = e_{ref}\left(\frac{1}{f_{ref}A_i}\right)^2 = \mathcal{K}\frac{1}{A_i^2} \quad i = 1, \ldots, C \tag{4.3}$$

Finally, the total energy of the task, under the constraint that the total execution time τ meets deadline A, can be written as:

$$E = \sum_{i=1}^{C} e_i = \mathcal{K}\sum_{i=1}^{C}\frac{1}{A_i^2} \quad \text{where} \sum_{i=1}^{C} A_i = A \tag{4.4}$$

It can be shown (see Appendix A.1) that the lower bound for E is $\mathcal{K}C^3/A^2$ which can be obtained if and only if all the clock cycles are of the same certain length: $A_i = A/C$. This means that there is a unique ideal processor speed for all clock cycles. This is the speed for which the task execution exactly covers A (Figure 4.3.b). Formally, the minimal energy frequency and speed are:

$$f_{ideal} = \frac{C}{A} \quad s_{ideal} = \frac{C}{Af_{ref}} \tag{4.5}$$

Unfortunately, real processors can only work at a discrete range of clock frequencies. Yet, a close to optimal virtual speed can be obtained even on real processors, as discussed in Section 3.3.2.

4.3 Probabilistic Execution Pattern

If the required number of clock cycles differs from instance to instance, the task is said to have a probabilistic execution pattern. Since we assume to

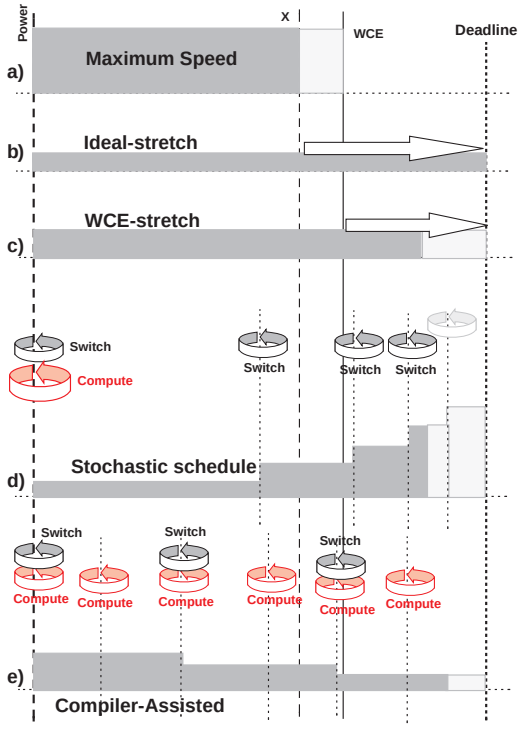


Figure 4.3: Various types of task level schedules: a) Using maximum speed, b) Ideal-stretch, c) WCE-stretch, d) stochastic, and e) compiler-assisted. The dark grey regions refer to a single instance execution, while the lighter grey ones refer to the worst case.

have no knowledge about the data input to each instance, or how this data is processed, we cannot be sure of the exact execution path beforehand. The number of clock cycles required by an instance is modeled as a probability distribution function (see Section 3.1.1). Tasks with unique execution pattern are actually a particular case of tasks with probabilistic execution pattern.

The important characteristic of tasks with probabilistic execution pattern is that they will not always exhibit their worst case behavior. If one can adapt the processor speed such that the task does not execute faster than it is needed, one can reduce the energy consumption. At the same time the real-time deadlines have to be met, forcing a lower limit on the task execution speed.

Ideally, one would know the exact number of required clock cycles before

the instance starts executing and thus schedule for the optimal speed (Figure 4.3.b). Of course, this choice, referred to as *ideal-stretch*, is seldom possible in practice. The obvious approach, referred to as WCE-*stretch*, is to plan for the worst case: assume that the task will exhibit its worst case behavior, and use the corresponding speed (see Figure 4.3.c). The drawback of this approach is that the worst case behavior is rather improbable, and using the high speed is seldom actually required to meet the deadline. This means that WCE-stretch is not energy efficient. There are other techniques which are more efficient from the energy point of view. We describe in detail our own approach to this problem, *stochastic voltage scheduling* (Figure 4.3.d). We also briefly describe another class of energy-efficient scheduling methods, called compiler-assisted scheduling, based on the work described in [SKL01] and [MACM00] (Figure 4.3.e). Finally, the section concludes with a more detailed comparison between the approaches described here.

4.3.1 Stochastic Scheduling

The basic idea behind stochastic scheduling consists in finding a speed schedule that minimizes the expected value of instance energy while still meeting the deadline. An instance starts executing at a low speed and then gradually accelerates, to meet the deadlines. Since the instance might not be a worst case, it can happen that high speed (and power eager) regions are avoided.

The stochastic schedule (Figure 4.3.d) for a task τ is obtained using its execution pattern $\eta(y)$. The execution pattern can be obtained off-line, via simulation, or built and improved at run-time. Let us denote by X the random variable associated with the number of clock cycles used by a task instance. We also use the cumulative density of probability function, $cdf(x) = \sum_{y=1}^{x} \eta(y)$, associated with the random variable X. This function reflects the probability that a task instance finishes before a certain number of clock cycles. If WCE is the worst case number of clock cycles, clearly $cdf(z) = 1$ for $\forall z \geq \text{WCE}$.

Recall that building a schedule for a task means assigning a specific processor speed for every clock cycle up to WCE. Each cycle x, depending on the adopted speed, will consume a specific energy, e_x. But each of these cycles are executed with a certain probability, so the average energy consumed by cycle x can be computed as $(1 - cdf(x))e_x$. To obtain the expected energy for the whole task, we have to consider all the cycles up to WCE:

$$\overline{E} = \sum_{x=1}^{\text{WCE}} (1 - cdf(x))e_x \tag{4.6}$$

This is the value we want to minimize by choosing appropriate voltage levels and clock frequencies for each cycle. Since WCE may be a large number in practice, one can group several consecutive clock cycles into equal size groups.

In that case, x would refer to such regions of consecutive clock cycles. For the sake of brevity and clarity we describe here only the simpler case, when the speeds are decided clock cycle by clock cycle.

A clock length of k_x corresponds to a clock frequency $f_x = 1/k_x$. Using equation 3.7 for $\beta = 2/(\gamma - 1)$ we can rewrite the clock cycle energy as:

$$\begin{aligned} e_x &= e_{f_x} = e_{ref}\left(\frac{f_x}{f_{ref}}\right)^{\beta} = \\ &= \frac{e_{ref}}{f_{ref}^{\beta}}\frac{1}{k_x^{\beta}} = \\ &= \mathcal{K}\frac{1}{k_x^{\beta}} \end{aligned} \tag{4.7}$$

where e_{ref}/f_{ref}^{β} is constant ($\mathcal{K}$) for a given processor. As mentioned in Section 3.3.2, in the common case $\gamma = 2$ making $\beta = 2$. For clarity we bind now $\beta = 2$, but the rest of the calculus can be carried out for any reasonable value of β.

Task τ has to complete its execution during an allowed execution time, A. If we denote the clock length associated to clock cycle x by k_x, this constraint can be written as:

$$\sum_{x=1}^{\text{WCE}} k_x \leq A \tag{4.8}$$

If we substitute 4.7 in 4.6 we obtain:

$$\overline{E} = \mathcal{K}\sum_{x=1}^{\text{WCE}} \frac{(1 - cdf(x))}{k_x^2} \tag{4.9}$$

which is the value to be minimized. For the sake of simplicity and without loss of generality we will assume $\mathcal{K} = 1$ from now on. By mathematical induction one can prove (see Appendix A.1) that the right hand side of 4.9 has a lower bound (using also 4.8):

$$E_{LB} = \frac{\left(\sum_{x=1}^{\text{WCE}} \sqrt[3]{1 - cdf(x)}\right)^3}{\left(\sum_{y=1}^{\text{WCE}} k_y\right)^2} \geq \frac{1}{A^2}\left(\sum_{x=1}^{\text{WCE}} \sqrt[3]{1 - cdf(x)}\right)^3 \tag{4.10}$$

This energy lower bound can be reached if and only if:

$$k_y = A\frac{\sqrt[3]{1 - cdf(y)}}{\sum_{x=1}^{\text{WCE}} \sqrt[3]{1 - cdf(x)}} \tag{4.11}$$

These are the optimal values for the clock cycle length in each clock cycle up to WCE.

For processors with a discrete range of speeds, these values will most likely not overlap with the available clock lengths. These ideal clock lengths have to be converted to available clock cycle lengths. This conversion is done in a similar way to deriving a dual level voltage schedule from an ideal one, as described in Section 3.3.2 and detailed in the following.

In principle, to transform the ideal clock frequencies into real ones, we have to distribute the work done in each ideal clock cycle to real clock cycles. To obtain the work performed during real clock cycles, we start from the two consecutive available clock cycles bounding the ideal clock cycle k_y, $CK_i < k_y \leq CK_{i+1}$. The work of the ideal cycle can be performed during the same time interval using real clock cycles, if:

$$\begin{aligned} k_y \quad &= \quad w_{iy}CK_i + w_{(i+1)y}CK_{i+1} \\ &\text{with} \quad w_{iy} + w_{(i+1)y} = 1 \\ &\text{and} \quad k_y \in (CK_i, CK_{i+1}] \end{aligned} \tag{4.12}$$

where w_{iy} is the part of the work of k_y given to CK_i and the rest is the work given to CK_{i+1}. Thus, each ideal cycle in the task will distribute its work between two of the several available clock lengths. It is possible that several ideal clock cycles k_y will distribute some of their work to the same real clock cycle j, since they end up to be right above or right below the real clock cycle value: $k_y \in (CK_{j-1}, CK_j]$ or $k_y \in (CK_j, CK_{j+1}]$. For simplicity, we denote the set of ideal clock cycles k_y distributing their work load to the same real clock cycle j by $\mathcal{Y}_j$. The accumulated workloads for each available clock cycle is obtained by summing up the workloads resulting from individual ideal clock cycles:

$$w_j = \sum_{k_y \in \mathcal{Y}_j} w_{jy}, \quad j \in 1 \ldots VL \tag{4.13}$$

where VL is the number of available processor speeds (corresponding to pairs of supply voltage levels and clock frequencies). Note that the cumulated workloads w_j are real numbers. Since we can only execute an integer number of clock cycles, these workloads have to be transformed to integers. This conversion is not as trivial as it seems at a first glance.

First, if the x in equation 4.6 was referring to groups of cycles rather than single clock cycles, the decimal part of w_i can be partially converted into integers. Thus, if x refers to groups of q actual clock cycles, the corresponding workload for each available clock cycle will actually be $w_j^{actual} = qw_j$, which is still a real number. This adjustment, if applicable, is useful in reducing the error introduced by the next step, which does the actual transformation from real numbers to integers.

Let us consider then a straightforward transformation, using smallest integer larger than the workload, $\lceil w_j \rceil$. Since a task instance starts executing

using the slowest speeds and each region uses one full additional clock cycle, it might happen that the deadline is missed. This comes from the fact that parts of the high speed clocks are forced into slow clocks. A better choice is to use the largest integer smaller than the workload $\lfloor w_j \rfloor$. Additionally, the cycles lost by this truncation, computed as $\text{WCE} - \sum_j \lfloor w_j \rfloor$ need to be added to the fastest speed workload. This may result in sub-optimal energy consumption, but the error is negligible for tasks requiring a large number of clock cycles. Of course, better transformations are possible, but we will not investigate this subject further.

Example 4.2 (Stochastic Schedules for Normal Distributions):
This example uses a task with an execution pattern distributed according to a normal probability with the mean $\mu = 70$ cycles and the standard deviation $\sigma = 10$. The number of clock cycles for the worst case execution $\text{WCE} = 100$. Note that these figures can be percentages instead of actual amounts of clock cycles. Let us consider that the available resource is a processor with four clock frequencies f, $f/2$, $f/3$, and $f/4$. Note that for the fastest clock f, the worst case execution will complete after a time interval given by $\text{WCE}_f = \text{WCE}/f$. We are able now to compute two voltage schedules obtained for two different values of the allowed execution time, $A = 3\text{WCE}_f$ and $A = 2\text{WCE}_f$. These are the cases when the allowed execution time is three times and respectively twice the time required by the worst case at the fastest clock. The schedules, depicted in Figure 4.4, are given in number of clock cycles executed at each available frequency. Note that the tasks may finish before executing the whole schedule, before reaching higher speed regions.

Computational complexity

Computing the stochastic schedule for a task has to be done before the task starts executing. If the allowed execution time A varies, the actual distribution of the workload to speeds also varies. In that case, the exact stochastic schedule has to be determined at run-time. Yet, there are computations that can be performed off-line. If the probability distribution of the execution pattern is available off-line, the coefficients of A in equation 4.11 can also be computed off-line. If the probability distribution is built at run-time, these coefficients need to be recomputed every time the distribution changes. This step has an algorithmic complexity of order $O(\text{WCE})$. The exact values for the ideal clock cycles have to be computed at run-time, when the allowed execution time becomes known (also exhibiting $O(\text{WCE})$ complexity). Computing the cumulative workloads requires finding the bounding clock frequencies, available on a real processor, for each ideal clock cycle $O(\text{WCE} \log VL)$. Transforming the real workloads into integer numbers will also exhibit $O(VL)$. Finally, assum-

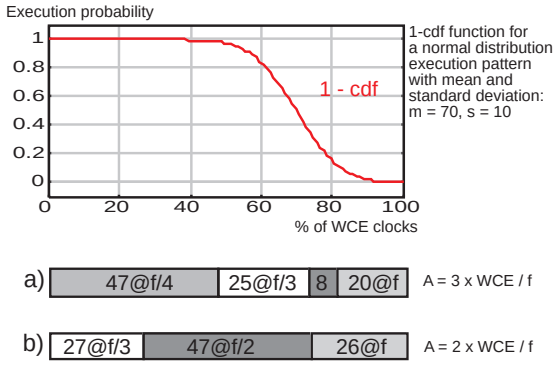


Figure 4.4: Two stochastic schedules for a task with normal distribution execution pattern, obtained for two different values of the allowed execution time.

ing $VL \leq \text{WCE}$, which is reasonable since VL, the number of processor speeds, is rather small, the overall worst case complexity for computing the stochastic schedule is $O(\text{WCE} \log VL)$. Since WCE is the determining factor in the overall complexity, it is clear now why working on groups of clock cycles instead of individual clock cycles is preferable.

4.3.2 Compiler-Assisted Speed Scheduling

For comparison, we briefly describe another class of energy-efficient scheduling methods for tasks with variable execution pattern. Instead of minimizing the energy over a large number of instances, these methods are based on the idea of scheduling each task instance as efficiently as possible. Each task is divided in sections for which the worst case execution pattern is estimated (see Figure 4.3.d). The sections are usually procedures and loop bodies as in [MACM00] or basic blocks detected at assembly level as in [SKL01], but may be arbitrary slices of code as in [LS00b]. The method is based on the observation that sections do not always follow their worst case scenario and subsequent sections may benefit from the resulting unused time, referred to as *slack*. At the beginning of each section the processor speed is adjusted to accommodate the slack and at the same time to be able to complete the remaining work even for the worst case scenario. In principle it is most effective to apply the new speed as soon as a decision decreased the number of cycles required by the remaining worst case execution. That is the reason why procedures, loops or basic blocks are chosen as sections. The compiler introduces the code needed for updating the worst case execution pattern and

for switching the processor speed. Therefore, this class of methods is referred to as *compiler-assisted scheduling*.

There are several important aspects in compiler-assisted scheduling:

- *section granularity* — If there were no overhead for the scheduling code, the ideal section would be the basic block (see [ASU86] for a definition of a basic block). Yet, one must take into consideration the overhead of computing and switching the processor speed. For example, if the basic blocks are around 10 clock cycles, and a speed switch takes 1000 cycles (Intel 80200, [Int01]), re-scheduling in every basic block will result in a ridiculously large increase in overall execution time. Moreover, there are basic blocks that do not change the worst case execution. Finally, since the available speed range is discrete, some of the re-scheduling points, resulting in minor changes in speeds, can be discarded. Therefore, in practice, the optimal section size may be much larger than a basic block. Choosing the best set of re-scheduling points is a complex task, usually handled by heuristics.

- *worst case pattern estimation* — Detecting the number of cycles executed by a section in the worst case is another delicate issue since it has to consider pipeline stalls, cache misses, interrupts, etc. Moreover, it often requires the designer to explicitly state loop bounds or input data characteristics.

- *slack distribution* — Once a section produces some slack, this has to be distributed to the following sections. There are several slack distribution strategies, but we only mention here two. A greedy approach would give all slack to the immediately next section, which might result in the processor speed oscillating up and down across sections. A better strategy is to distribute the slack proportionally over all sections, according to the ratio between each individual section and total remaining worst case execution time.

- *speed selection* — Both soft and hard real-time tasks may employ compiler-assisted speed scheduling. For hard real-time tasks, the speed will always be chosen to meet the deadline even in worst case. For soft real-time tasks, one may occasionally choose to select speeds according to a predictive algorithm [MACM00]. Note also that in practice only available speeds may be selected, making this aspect strongly related to the first one, section granularity.

It is important to realize which are the static/compile-time decisions and which are run-time decisions for a compiler-assisted schedule. First, the re-scheduling points are fixed by introducing the required code only at specific fixed locations. Moreover, if the allowed execution time for the task is fixed,

speed selection may also be static, without keeping track about the slack at run-time. In this case the code performing re-scheduling may be a simple speed change, without any computations. Yet, it is more likely that the allowed execution time changes from instance to instance at run-time. This means that the optimal speeds vary and have to be re-computed. Therefore, the overhead of re-scheduling increases. Furthermore, the subset of the chosen re-scheduling points might not be the best for each instance with its specific allowed execution time. The advantages and drawbacks of this method compared to the previously described ones are detailed in the next section.

4.4 Discussion

In this section we compare three of the scheduling methods described previously in this chapter: WCE-stretch, stochastic and compiler-assisted scheduling. First, we address some real-time issues in the context of the aforementioned scheduling methods. Then we examine the advantages of using our own stochastic schedule versus using a simple WCE-stretch schedule. Next we look at stochastic scheduling and compiler-assisted scheduling side by side, and compare them not only from the energy efficiency point of view, but also from the design complexity and implementation points of view.

4.4.1 A Few Real-Time Considerations

From the real-time perspective, the task level speed schedule decides if the task finishes in time or not. For hard deadlines, the task must finish during the allowed execution time A. All three scheduling methods, WCE-stretch, stochastic and compiler assisted, can do this without problems if the time estimates used to compute the schedule are accurate.

For soft deadlines, the situation is slightly different. Using a stochastic scheduling approach one can accurately choose the probability for an instance to finish after the deadline. If, for example, we are willing to allow 10% deadline misses, it is enough to examine the $1 - cdf(x)$ function and choose a new WCE for which the function value is 10%. The other two scheduling strategies, are not so successful. The WCE-stretch may only control the worst case deadline overshoot. The compiler-assisted approach may also control the overshoot, but for individual paths. Although the compiler-assisted method may be changed to allow an expected number of deadline violations, for the stochastic approach this is straightforward.

4.4.2 WCE Stretch vs. Stochastic

The WCE-stretch scheduling approach treats in principle all instances as being the worst case. This implies running at a constant high speed throughout all the phases of task instance execution. By contrast, the stochastic approach lowers the energy consumption of the early phases at the expense of the later phases. Over a large number of runs this leads to a considerable decrease for the early phases, since they will always be executed, and a small increase for the later phases, that are seldom executed.

Let us examine first an ideal situation, when the processor may run at any speed from almost 0 to almost ∞. The expected energy consumed by the stochastic schedule in this case can reach the lower bound given by equation 4.10:

$$E_{stoch} = \mathcal{K}\frac{1}{A^2}\left(\sum_{x=1}^{\text{WCE}} \sqrt[3]{1 - cdf(x)}\right)^3 \tag{4.14}$$

The expected energy of the WCE-stretch schedule can be computed as:

$$E_{\text{WCE}-stretch} = e_{\text{WCE}-stretch}\overline{X} \tag{4.15}$$

where $\overline{X}$ is the mean of the execution pattern distribution and $e_{\text{WCE}-stretch}$ is the clock cycle energy for the speed required by the WCE-stretch method. Using equation 4.7 (with $\beta = 2$) and the fact that the clock frequency for WCE-stretch is $f_{\text{WCE}-stretch} = \text{WCE}/A$, we can rewrite the expected energy of the WCE-stretch schedule as:

$$E_{\text{WCE}-stretch} = \mathcal{K}\frac{1}{A^2}\text{WCE}^2\overline{X} \tag{4.16}$$

Finally, to get an idea about the gain of using a stochastic schedule, we need to look at the ratio between the energy of the stochastic schedule and that of a WCE-stretch schedule:

$$\varepsilon = \frac{E_{stoch}}{E_{\text{WCE}-stretch}} = \frac{\left(\sum_{x=1}^{\text{WCE}} \sqrt[3]{1 - cdf(x)}\right)^3}{\overline{X}\text{WCE}^2} \tag{4.17}$$

Notice that the ratio depends on the distribution and it is independent on the allowed execution time A. This holds only for the ideal case when the range of speeds is unbounded. Table 4.1 contains the efficiency ratio ε for several distributions. Granted the shape of the distribution is important, the stochastic schedule is always better or at least as efficient in the long run as the WCE-stretch schedule.

For more realistic assumptions, such as limited range of speeds, the advantage of a stochastic schedule becomes smaller. Consider a task with an execution pattern varying according to a normal distribution with the mean

Table 4.1: Stochastic vs. WCE-stretch schedule energy for various task execution pattern distributions.

distribution	ε
Uniform(0, WCE)	0.8419
Normal($\mu = \text{WCE}/2$, $\sigma = \mu/3$)	0.6461
Normal($\mu = 2\text{WCE}/3$, $\sigma = \text{WCE}/9$)	0.7417
Normal($\mu = \text{WCE}/3$, $\sigma = \text{WCE}/9$)	0.3048
Exponential($\lambda = 10/\text{WCE}$)	0.2434

$\mu = (\text{BCE} + \text{WCE})/2$ and the standard deviation $\sigma = (\text{WCE} - \text{BCE})/6$. For several cases ranging from highly flexible execution time ($\text{BCE}/\text{WCE} = 0.1$) to almost fixed ($\text{BCE}/\text{WCE} = 0.9$) we built stochastic schedules for a range of allowed execution times (from WCE/f to $3 \times \text{WCE}/f$). For this experiment we considered a processor with 9 different voltage levels, equally distributed between f and $f/3$. For a large number of task instances generated according to the given distribution we computed both the energy of the stochastic schedule and the WCE-stretch schedule. The average energy consumption of the stochastic schedule as a part of the WCE-stretch schedule is depicted in Figure 4.5. Note that when the allowed time approaches either WCE/f or $3 \times \text{WCE}/f$, the energy consumptions become equal. The lowest possible clock frequency is $f/3$, which anyway means 3-times WCE/f, so there is no better schedule for these cases. In turn, when the allowed time closes WCE, there is no other way but to use the fastest clock. Somewhere between the slowest and the fastest frequencies (e.g., when allowed execution time is twice WCE) we obtain the largest energy gain since the stochastic schedule can use the whole spectrum of available frequencies. Note that the energy gains become more important when the task execution time varies more ($\text{BCE}/\text{WCE} \to 0.1$).

Example 4.3 (Task Level Voltage Schedules on Intel 80200):
To practically examine the energy saved by a stochastic approach versus a WCE-stretch method we used the Intel 80200 system described in Appendix B. The task in this experiment is composed of a simple loop that terminates after a variable number of iterations in each instance (see Appendix C). The number of iterations is computed at the beginning of each job according to a normal distribution with $\mu = 950$ and $\sigma = 300$. At the highest speed (733MHz), the best case and the worst case execution times are 30ms and 78ms, respectively. We considered that the task has to finish in at most 117ms (which is 150% of the worst case at the highest speed). When we use the WCE-stretch approach, we have to chose the ideal speed for which the worst case will finish exactly at the deadline. This will be realized in practice by using two real speed settings: 533MHz@1.3V and 466MHz@1.2V. Using only the 533MHz@1.3V speed

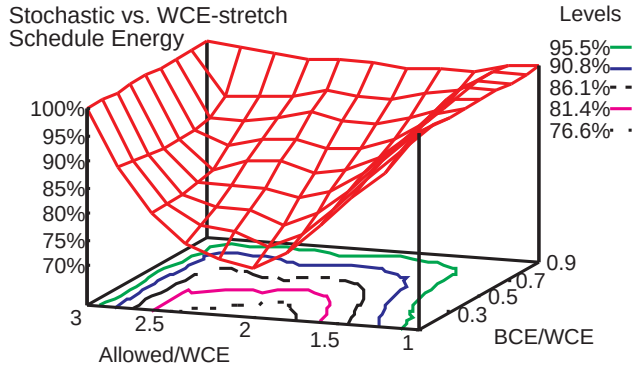


Figure 4.5: Average task energy for stochastic schedule compared to WCE-stretch schedule energy (100%) as a function of task execution pattern distribution and allowed time.

setting would give 107ms execution time in the worst case while using only the 466MHz@1.2V speed setting would be too slow since the worst case would take about 123ms. Finally, the WCE-stretch schedule will be composed by executing first 1150 iterations of the main loop at 466MHz@1.2V and the rest at 533MHz@1.3V. The power distribution for this schedule is depicted in the oscilloscope trace from Figure 4.6 **above**.

To obtain a stochastic schedule in this situation we have to know the best and worst case execution patterns. These two cases happen for the extremes of the value controlling the iterations in the task main loop. More precisely, these are 50 and 1850 respectively (the $[-3\sigma, 3\sigma]$ interval). Using these numbers as the input to our method for computing a stochastic schedule we get the number of iterations to be executed at each speed between the lowest and the highest: [186, 640, 181, 109, 79, 62, 593]. The power distribution for this schedule is depicted in the oscilloscope trace from Figure 4.6 **below**. When using the stochastic schedule, the worst case takes almost 117ms including the speed switches.

Running the two schedules for a large number of instances and averaging the power consumption, we obtain the average energy consumptions for both. The WCE-stretch average energy, as resulting from the trace in Figure 4.7 **above**, is around 24mJ. The stochastic schedule average energy, as resulting from the trace in Figure 4.7 **below**, is around 20mJ. Finally, this means that in the long run, using a stochastic scheduling approach, the energy consumption decreases by approximately 18% in this case.

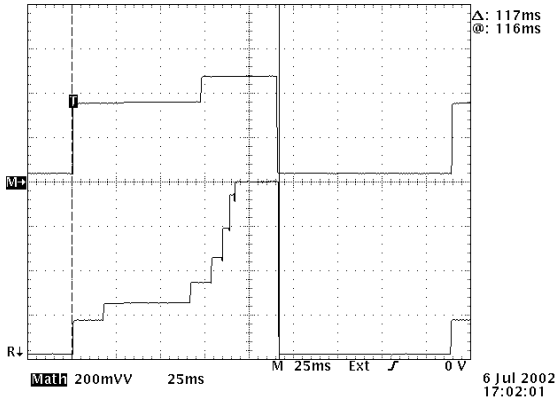


Figure 4.6: The power distribution for the WCE-stretch (**above**) and stochastic (**below**) schedules used in example 4.3. Different power levels reflect the different speed settings used. The worst case covers the whole schedule, while the other cases only the initial sections of the schedule.

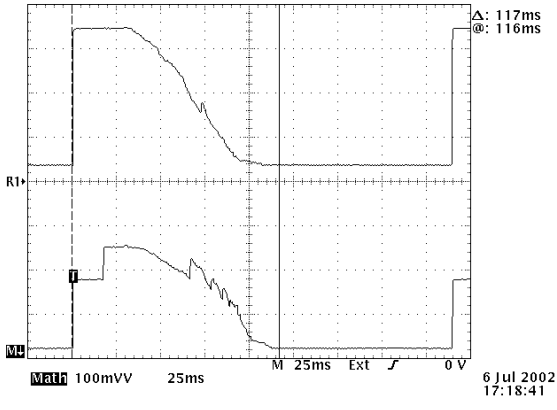


Figure 4.7: The averaged power curve of the WCE stretch (**above**) and stochastic (**below**) schedules used in example 4.3. The area under the graph represents the energy consumption, which is around 24mJ for the **above** curve and 20mJ for the **below** one.

4.4.3 Stochastic vs. Compiler-Assisted Scheduling

There are several important differences between our stochastic scheduling approach and a compiler-assisted method. We discuss them next.

Scheduling Overhead

As detailed in Section 4.3.1, for stochastic scheduling the stages requiring some computations are the off-line phase and at the execution start of a task instance. This implies, for example, computing the exact number of clock cycles after which to increase the processor speed. Subsequently, at these time moments, the only operation required consists of speed change requests to the processor. To give the control to the scheduler at these specific time moments, timers/event counters can be used. If the allowed execution time is the same for all instances, the speed switching code sequences can be inserted directly in the right points (as in Example 4.3).

The compiler-assisted approach involves an important off-line phase for identifying and processing the re-scheduling points. At run-time, the code inserted in these points actively checks the progress of the task instance and adjusts the processor speed accordingly. For a discrete and limited range of available speeds, the best set of points for switching speed may vary if the allowed execution time changes from instance to instance. To obtain good results, a compiler-assisted schedule needs usually more re-scheduling points than a stochastic schedule, depending, for example, on the number of basic blocks or functions. However, if the allowed execution time is the same for all instances, the appropriate speeds may be computed off-line and the scheduling points will only contain speed switching code.

For the most general case, the stochastic approach requires some overhead at each instance start-up, to compute the schedule. The compiler-assisted method has no overhead at start-up, yet it uses more re-scheduling points at run-time, even if not all of them result in speed switches. Considering the significant overhead of speed switching in real processors today ($\sim$ 1000 cycles on Intel 80200 [Int01]), a few additional cycles used on re-computing the optimal speed make for a negligible difference. To conclude, there is little difference in the overall run-time overhead between the two approaches.

Implementation Complexity

Considering the implementation level of both methods, one can distinguish between operating system (OS) level and task level. In principle we consider an implementation to be *OS-level* when the task is completely unaware that it runs on a variable speed or fixed speed processor. Alternatively we consider an implementation to be *task level* when the task handles all the speed scheduling related issues. Note that a task level implementation may or may not use interfaces provided by the OS.

The compiler-assisted approach requires access to the internal structure of the task. This is used to detect the re-scheduling points, each of which requires specific handling. The OS may be oblivious to the speed selection

related operations performed by the task. This may be an advantage if one is forced to use older, energy-unaware operating systems.

Stochastic scheduling does not need to have any information about the internal structure of the task. It can be implemented exclusively inside the OS, while the task remains oblivious to the fact that it runs on a variable speed processor. Alternatively, if one wishes to use an existent OS, a task level scheduler may be implemented together with the task. In this sense, stochastic scheduling is the less restrictive of the two methods.

The consequences of implementing voltage management at OS or at task level are important. In principle, it is safer to manage the hardware resources via common interfaces, that are aware of the system architecture. Furthermore, better management is possible if the decisions are taken based on the overall state of the system, instead of just partial or local states. In particular, slack can be distributed more efficiently if a central, common scheduler is used instead of letting the tasks handle slack internally. All these suggest that OS level speed management is more likely to be more energy efficient than task level scheduling. Of course, knowledge about the internal structure of the task would additionally improve the efficiency. The only instance for employing a pure task level speed management is when the requirements force the use an old (energy unaware) OS.

Energy Efficiency

Until now we have not looked at the actual energy efficiency of the two methods. In fact it is not easy to compare the two approaches in general since their results depend on rather orthogonal properties.

For example, compiler-assisted voltage scheduling depends very much on the internal task predictability, while it is not very sensitive to deadline variations. An important property of the compiler-assisted scheduling is that during task execution, more and more information is acquired about the actual execution time of the task. Knowing this right in the beginning of the task would lead to an ideal schedule, using the optimal speed. If the actual execution time is not revealed until the very end (the task terminates suddenly, based for example on an external interrupt), one has to be prepared for the worst case. In this situation, the task will actually use the WCE-stretch schedule. Between the two extremes just described, lies the real behavior of any compiler-assisted schedule.

The stochastic approach is not dependent on the internal task predictability since the schedule is fixed before the task starts executing. However, this method is very dependent on the deadline variations. If, for example, the allowed execution time is very close to the WCE at the maximum speed, there is little a stochastic schedule can do. All task instances would end up running at a close to maximum speed. On the other hand, the compiler-assisted method

would detect early finishing instances at run-time and lower the processor speed accordingly.

Finally, a comparison between the two methods from the energy efficiency point of view would only make sense for particular tasks and requirements. Example 4.4 contains such an analysis for a task with a normal distribution execution pattern.

Example 4.4 (Schedule Energy Efficiency Analysis):
Consider a task with an execution pattern exhibiting a Gaussian distribution with $\mu = \text{WCE}/2$ and $\sigma = \text{WCE}/6$. This is enough for building a stochastic schedule, yet for the compiler-assisted method we need more information about the internal structure of the task. For this reason we consider that there is a point in the task instance before which nothing is known about the actual execution time, and after which the exact execution time becomes known. If this point is in the beginning of the task, the compiler-assisted schedule will be closer to an ideal schedule. Conversely, if this decision point approaches the end of the task, the compiler-assisted schedule will be closer to the WCE schedule. In the following we denote the ratio between the first region clock cycles and the total execution time X by ϑ, the *unawareness factor* of the task. It is interesting to see how the energy efficiency of the compiler-assisted schedule depends on this factor. The total energy consumed in the compiler-assisted schedule is composed of the energy consumed by the two separate regions. The first one is executing $X_1 = \vartheta X$ cycles at a clock frequency required by finishing the WCE during the allowed execution time A, $f_1 = \text{WCE}/A$. The second region is executing the remaining $X_2 = (1-\vartheta)X$ cycles at the clock frequency required to exactly finish at the end of the allowed time A, $f_2 = X_2/(A - X_1/f_1)$. From these, using also the equations and notations in Sections 4.3.1 and 4.4.2, can be shown that the compiler-assisted schedule energy is:

$$E_{\text{CA}}(X) = E_1 + E_2 = \mathcal{K}\left(\frac{\text{WCE}}{A}\right)^2 X\left[\vartheta + \frac{(1-\vartheta)^3}{(\text{WCE}/X - \vartheta)^2}\right] \tag{4.18}$$

The mean or expected value for the compiler-assisted approach may be computed as:

$$E_{\text{CA}} = \overline{E_{\text{CA}}(X)} = \int_0^{\text{WCE}} E_{\text{CA}}(x)\eta(x)\mathrm{d}x \tag{4.19}$$

where $\eta(x)$ is the execution pattern distribution, here a Gaussian distribution with $\mu = \text{WCE}/2$ and $\sigma = \text{WCE}/6$. Note that E_{CA} cannot be computed simply as $E_{\text{CA}}(\overline{X})$. Furthermore, we are interested in comparing this with the $E_{\text{WCE}-stretch}$ energy, as computed in equation 4.16:

$$\varepsilon' = \frac{E_{\text{CA}}}{E_{\text{WCE}-stretch}} = \vartheta + \frac{(1-\vartheta)^3}{\overline{X}}\int_0^{\text{WCE}} \frac{x}{(\text{WCE}/x - \vartheta)^2}\eta(x)\mathrm{d}x \tag{4.20}$$

Note that for our execution pattern distribution, $\overline{X} = \text{WCE}/2$. Using numerical integration we computed the value of ε' for ϑ varying between 0 and 1 and for various values of WCE. It turns out that for the given distribution, ε' is only a function of ϑ, independent of the WCE or the allowed execution time A. The results are depicted in Figure 4.8.

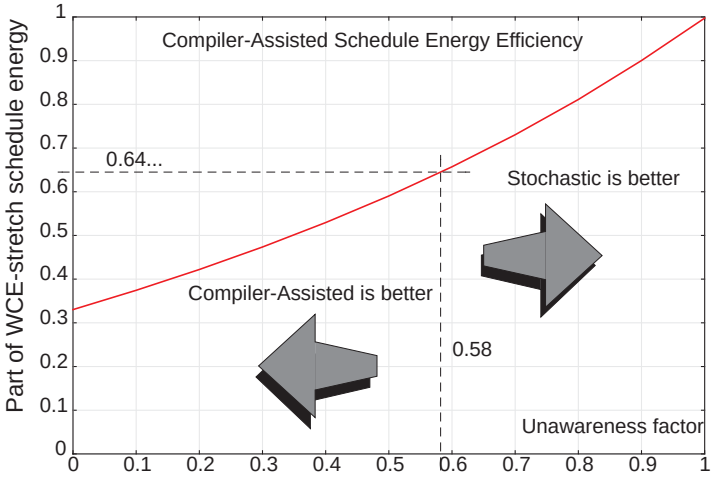


Figure 4.8: The compiler-assisted vs. the WCE-stretch schedule energy ratio, ε', as a function of task unawareness factor, ϑ.

It is important to realize that the expression of ε' is accurate only if the processor has some ideal characteristics. Namely, the processor has to be able to run at any speed between almost 0 and the worst case speed. For real processors, the efficiency of the compiler-assisted schedule degrades, as was the case for the stochastic schedule. Yet, for this analysis we will stick to the ideal model of the processor.

Line 2 in Table 4.1 already contains the efficiency of a stochastic schedule for the given distribution. The $0.64\ldots$ value of ε says that the expected energy of a stochastic schedule is always around 64% of the energy of the WCE-stretch schedule, independent of the internal characteristics of the task. However, Figure 4.8 shows that the compiler-assisted schedule consumes 64% of the WCE-stretch schedule energy for the unawareness factor $\vartheta \approx 0.58$. If the task exhibits an unawareness factor larger than 0.58 a compiler-assisted schedule would consume more energy than the stochastic schedule. On the contrary, if ϑ is smaller than 0.58, the compiler-assisted schedule would consume less energy than the stochastic schedule.

Conclusion

After comparing the stochastic and compiler-assisted scheduling methods, we conclude that there is no *best* scheduling strategy for all situations. Energy efficiency, scheduling overhead, implementation level are just a few factors that can pull the choice either way. One must carefully analyze the problem and pick the most suited scheduling method for the specific situation.

CHAPTER 5

TASK GROUP SCHEDULING

THIS CHAPTER ADDRESSES energy-efficient scheduling of task groups under real-time constrains. We discuss both static scheduling of task graphs and run-time scheduling of task sets. Task graph scheduling strategies were designed for multi-processor systems, but may be used also for the uni-processor case. For run-time scheduling we focus on the uni-processor case and schemes with both static priorities (Rate-Monotonic like) and dynamic priorities (Earliest Deadline First). We show that, with special scheduling strategies, groups composed of tasks with probabilistic execution pattern can run in an energy efficient manner. A classification of the scheduling methods presented here is given in Table 5.1.

Table 5.1: A classification of the task group scheduling strategies presented in this chapter. The section dedicated to each method is given in parenthesis.

	execution	**uni-proc.**	**multi-proc.**
Task Graph	—	Proportional Stretch (5.2)	LS-PS (5.3) LEneS (5.4)
Task Set	fixed	Maximum Required Speed (5.5)	—
	probabilistic	Slack Distribution RM (5.6) Uncertainty Based (5.7)	

From the scheduling techniques presented here, those addressing task graphs are static, performed off-line, oblivious to the dynamic characteristics of the task. In these methods, all tasks are treated as having unique (fixed) execution pattern, usually the same as WCE. The approaches used for the multi-processor case (LS-PS, LEneS) may in fact be used even for the uni-processor case, yet their performance is no better than the uni-processor

specific method (Proportional Stretch).

The scheduling approaches addressing task sets were designed, in principle, for the uni-processor case. If the tasks exhibit a fixed execution pattern, all the energy-efficient scheduling decisions may be taken off-line (Maximum Required Speed). For tasks with variable execution pattern, one may additionally employ run-time techniques for energy reduction (Uncertainty-Based Scheduling, Slack Distribution RM).

Note that all the scheduling strategies presented in this chapter assume that the binding of tasks to processors (task mapping) is already decided. The choices that have to be made at this level are related to timing, such as deciding the task order or/and exact duration for each task execution or, alternatively, the recommended processor speed for each task. Many of the strategies presented here are additions or improvements to already existing energy-unaware scheduling techniques.

5.1 The Energy of a Task Group Schedule

Before describing any scheduling techniques, it is important to take a look at the measure we want to minimize: the energy consumption. Each schedule yields a certain energy consumption, depending on the processor speeds assigned to the workload required by the tasks and the communication between them. Since we consider the communication energy to be decided by the task mapping, the energy we are interested to minimize during scheduling is the task group energy.

For periodic tasks with fixed execution time, the energy is constant over all hyper-periods and can be exactly computed off-line. Given a group of tasks $\{\tau_i\}$, each with a unique execution pattern and assigned to a processor P_i, a schedule ξ imposes a unique speed s_{ij} for each of the C_i clock cycles of task τ_i. Since the tasks may arrive with different periods T_i, it makes sense to compute the energy consumption over a hyper-period. The hyper-period, denoted by H, is computed as the least common multiplier of all the task periods, $H = lcm\{T_i\}_{i=1...N}$. Finally, to compute the energy needed by a certain task over a hyper-period one must add all the energy consumption of all its H/T_i number of instances. If we denote the energy of instance j of task i by E_{ij} then the energy of a schedule ξ is:

$$E_\xi = \sum_{i=1}^{N} \sum_{j=1}^{H/T_i} E_{ij} \tag{5.1}$$

For tasks with fixed execution pattern, the instance energy is fixed over all

instances in a hyper-period $E_{ij} = E_i$, so the schedule energy becomes:

$$E_\xi = \sum_{i=1}^{N} \frac{H}{T_i} E_i \tag{5.2}$$

Note that, for task graphs, all tasks have the same period, meaning that all tasks run only once in a hyper-period. In this case, in equation 5.2, $H/T_i = 1$. Also, the processors may have different power consumption characteristics that have to be taken into consideration by using corresponding energy-speed functions, e^{P_i}.

Using the notations from Section 3.3.2, the energy of a task with fixed execution pattern is computed as:

$$E_i = \sum_{j=1}^{C_i} e^{P_i}(s_{ij}) \tag{5.3}$$

Using the equation 3.7, for processors that can run at any speed, the expression above becomes:

$$E_i = e_{ref}^{P_i} \sum_{j=1}^{C_i} {s_{ij}}^{\beta} \tag{5.4}$$

For tasks with variable execution time, one can compute the expected energy off-line:

$$\overline{E_i} = \sum_{j=1}^{C_i} {s_{ij}}^{\beta} (1 - cdf_i(j)) \tag{5.5}$$

The actual energy consumption can be computed usually only after execution, when all the execution times are available.

Finally, we are interested in sets of tasks with fixed execution time, each running at a single ideal speed, on a single processor. We will also use the most common value for $\beta = 2$. From equations 5.2 and 5.4 we obtain the schedule energy:

$$E_\xi = \sum_{i=1}^{N} \frac{H}{T_i} e_{ref} C_i {s_i}^2 = H e_{ref} \sum_{i=1}^{N} \frac{C_i}{T_i} {s_i}^2 \tag{5.6}$$

When all tasks run at the same speed s, and using Definition 3.4 of processor utilization U, the above equation becomes:

$$E_\xi = H e_{ref} f_{ref} s^2 U \tag{5.7}$$

Note that, if no speed scaling is performed, as in all classic real-time scheduling algorithms, the energy of a schedule is $E_1 = H e_{ref} f_{ref} U$. This means

that the energy consumption of a uniformly scaled schedule differs from a non-scaled schedule by a factor of s^2. The processor utilization can be at most 100%, which can be achieved by a uniformly scaled schedule if and only if the speed is reduced to $s_{min} = U$. Consequently, the lower bound for energy consumption of a uniformly scaled schedule E_{LB} has then the following property:

$$E_{LB} = U^2 E_1 \tag{5.8}$$

Although we arrived at this formula for task sets, the same conclusion can be drawn for task graphs running on homogeneous architectures. Most of the following energy reduction methods are attempting to maximize utilization, varying the speed as little as possible and still keeping a feasible schedule. For task graphs on heterogeneous architectures, the more specialized LEneS algorithm (Section 5.4) is a possible answer.

5.2 The Proportional Stretch Approach

Consider for the moment a task group with a single, global timing constraint (deadline A), running on a single processor. As shown in Section 4.1, from the energy point of view it is best if any idle time is eliminated by running the tasks slower. In fact, following the same reasoning given in Section 4.2, we can conclude that there is a unique speed for all tasks that gives an optimal, minimal energy schedule.

At the reference speed, for the reference frequency f_{ref}, the total execution time for the group will be $C = 1/f_{ref} \sum_i C_i$. For the deadline C, the optimal unique speed used by any ordering is exactly the reference speed. Any order is therefore an optimal schedule from the energy consumption point of view. For all other deadlines A, the optimal schedule is obtained when all tasks execute at the same speed, covering the entire time interval A. The optimal speed is in this case $s_A = C/A$. The conclusion is that an optimal schedule for a deadline A may be obtained from any optimal schedule for C by a simple uniform stretch, proportional to C/A. Note also that s_A is actually equivalent to the processor utilization U, matching the conclusion on energy lower bound from Section 5.1. Finally, proportional stretch is a method that can be applied even to multi-processor systems and has a few important properties, as we detail in the following.

proportional stretch is a simple, and usually off-line technique, similar to the task level WCE-stretch. In fact, it is not a proper scheduling algorithm, but just a method of transforming an already existent schedule into another schedule, and do this optimally from the energy point of view. In our classification, this approach is declared as working on task graphs, but it can work as well for sets of tasks with the same deadline. The method can be applied

thus to task groups with a single deadline, on which internal deadlines depend, if they exist. We have also classified this as a uni-processor technique, yet it works also for multi-processor systems using a single type of processors. For heterogeneous multi-processor systems this method is not optimal. Furthermore, this method is only optimal for the case when the number of speeds used by a processor is not restricted. Whenever the transformation imposes speeds that cannot be achieved by the processor, the transformed schedule might not be optimal.

In principle, proportional stretch adapts an existent (possibly energy efficient) schedule built for a deadline A_1 to a new deadline A_2. More precisely, all the speeds required to run the tasks in the schedule for A_1 are scaled with the same factor given by $\rho = A_1/A_2$. In fact, this is equivalent to saying that the actual execution time for each task, or task portion is stretched by the same factor ρ.

Definition 5.1. Given task group $\{\tau_i\}_{i=1...N}$ with a deadline A_1 and a schedule ξ_1 imposing the task execution times $\{t_i\}_{i=1...N}$, a **proportional stretch** for a new deadline A_2 is a new schedule ξ_2, obtained from ξ_1, that scales each task execution times with the same factor A_1/A_2, resulting in the task execution times given by $\{A_1/A_2 t_i\}_{i=1...N}$.

This definition is valid for both non-preemptive and preemptive schedules. Moreover, there is no restriction on the task level schedule at this point. Each task may run at any combination of processor speeds inside its allowed execution time interval. Informally, the entire schedule stretches like an rubber band to fit the new deadline.

This method has an important characteristic on ideal processors (able to run at any speed). The technique guarantees that if the schedule for A_1 is optimal from the energy point of view, then the schedule for A_2 will also be optimal.

Theorem 5.1 (Proportional Stretch Optimality):
If a schedule for a deadline A_1 is energy optimal, then the proportional stretch for any other deadline A_2 is also energy optimal for that deadline.

The theorem can be proven immediately using the relation between the energy of a schedule and its stretched equivalent:

Lemma 5.2 (The Energy of a Proportionally Stretched Schedule):
Given a schedule ξ with an energy consumption E, then its proportional stretch by a factor ρ has an energy consumption equal to $\rho^\beta E$.

The lemma is deduced from equations 5.2 and 5.4, where all the processors have the same power characteristic (obvious for the uni-processor case),

by substituting each speed s_{ij} with scaled speeds ρs_{ij}. Note that the proportional stretch transformation preserves the order given by the energy consumption. Given two schedules for deadline A_1, denoted by $\xi_1(A_1)$ and $\xi_2(A_1)$, having the energy consumptions ordered as $E_1(A_1) < E_2(A_1)$, the energy consumptions of their proportional stretches for deadline A_2 are still in the same relation: $E_1(A_2) < E_2(A_2)$. Therefore, if a schedule yields minimum energy consumption for A_1, its ρ proportional stretch also yields the minimum energy consumption for A_2.

Returning to the theorem, for any schedule respecting A_1 there is exists a $\rho = A_1/A_2$ proportionally stretched schedule for A_2 (Remember that the processor can run at any speed). Conversely, for any schedule respecting A_2, there is a proportionally stretched schedule respecting A_1, with a factor $\rho^{-1} = A_2/A_1$. The proportional stretch transformation is thus a bijective relation. This means that there are no schedules for one deadline that are not transformations of schedules for the other deadline. Combining this observation with the consequences of Lemma 5.2, we arrive at the property expressed by Theorem 5.1.

Note that, if the system has resources that can operate at only one speed, Theorem 5.1 no longer applies. Since we consider inter-processor communications to be of fixed delay, the above optimality claim does not hold for schedules containing inter-processor communications. The problem comes from the fact that the communication delay does not change, making the actual stretching factor for tasks lower in practice. In such cases, one could consider the path with the critical computational time as the one to be stretched. Consequently, the available time should exclude the communication time, in particular the maximum communication time on the path with longer computations than the critical time. Let us denote paths by p_i and p_j, critical path by p_c, the sum of delays for stretch-able tasks in a path p by $Var(p)$, and the sum of delays for fixed tasks of a path p by $Fix(p)$. With this notations, a good proportional stretch factor is computed as:

$$\rho' = \frac{\max_{p_i}\{Var(p_i)\}}{A - \max_{Var(p_j) \geq Var(p_c)}\{Fix(p_j)\}} \tag{5.9}$$

We will not pursue analyzing this stretching method, since our LEneS algorithm, presented later on, is anyway able to handle fixed delay tasks better.

The following scheduling methods will often make use of the proportional stretch transformation to derive energy-efficient schedules for different deadlines from a certain initial schedule. As mentioned before, the derived schedules are energy optimal only if the initial schedules are energy optimal. Moreover, for multi-processor systems, all the processors must be of the same type to guarantee optimality. Furthermore, the schedule has to be free of fixed delay operations (such as inter-processor communications). In all other cases,

although not yielding optimal schedules, proportional stretch remains a simple and fast method to adapt schedules to new deadlines in an energy efficient manner.

5.3 List-Scheduling with Proportional Stretch

One of the classic scheduling heuristics for resource constrained task graphs is *list-scheduling* [DeM94]. Widely used for static scheduling, list-scheduling is in fact a family of priority based scheduling algorithms. Selecting the right priority function plays an important role in finding schedules. For example, a typical problem is execution time minimization under resource constraints. In this case, a popular priority function is the one based on the worst case remaining execution time (or critical-path). The longer the critical-path starting in a certain node, the higher the priority of that node. The constraints on resources usually refer to the number of available resources of a certain type, as is often the case in high-level synthesis. Alternatively, the resource constraints may refer to the exact assignment of operations to resources.

A pseudo-code description of a generic list-scheduling heuristic is given in Figure 5.1. The algorithm accepts as input a graph Γ, having its nodes τ already assigned to specific resources ($Res(\tau)$), and a priority function f. Note that the heuristic iteratively schedules nodes (τ), that have all their predecessors ($Pred(\tau, \Gamma)$) scheduled. It does so based on a priority function, that usually depends on the specific task but may include other parameters. If the priority of a node changes through out the scheduling procedure, depending for example on the previously scheduled nodes (Σ) or scheduling step (t), the priority is said to be *dynamic*. However, if the node priorities remain constant throughout the scheduling procedure, the priorities are said to be *static*. In this case they may be computed only once, in the beginning of the algorithm. Section 5.4 describes an algorithm from the list-scheduling family, having a dynamic priority function based on a combination between energy and execution time. For the moment we will focus on the classic list-scheduling with critical-path priority function extended with a proportional stretch transformation.

As mentioned before, a priority function commonly used with list-scheduling for execution time optimization is the *critical-path* priority. The graph nodes which have the longest path of successors, in terms of execution time, are assigned the highest priority. In fact critical-path yields static priorities, since it depends on the graph structure only.

The classic list-scheduling heuristic assumes fixed speed processors. With this assumption, it is enough to decide the start moment of a task to obtain a schedule. However, a speed scheduling algorithm must decide not only the start moment but also the processor speed or the end moment of

```
List_Scheduling(Γ(N,V), f) {
  t ← 0;
  Σ ← ∅;
  do {
    for each free resource r ∈ R {
      N_ready ← {τ ∈ N \ Σ | Pred(τ, Γ) ∈ Σ ∧  Res(τ) = r};
      for each task τ ∈ N_ready
        calculate p_τ ← f(τ, ...);
      τ_next ← τ ∧  p_τ = min{p_i}_{i∈N_ready};
      Schedule τ_next at moment t;
      Σ ← Σ ∪ {τ_next}
    }
    t ← the earliest moment r resource becomes available;
    Update the free resources in R;
  } while(Σ ≠ N);
}
```

Figure 5.1: Pseudo-code for the list-scheduling heuristic

a task (see Section 5.4). Still we may use a classic list-scheduling on a DVS-processor if we perform subsequent transformations for adjusting the execution speed. Taking into account that list-scheduling with critical-path priority tries to minimize the execution time, the proportional stretch transformation (from Section 5.2) appears to be a good subsequent step. This observation led us to the list-scheduling with proportional stretch (LS-PS) strategy for DVS-processors.

The LS-PS approach is comprised of two successive off-line steps:

1. Perform list-scheduling with critical-path priority using the reference speed(s) of the processor(s). This step will result in a schedule requiring time C to complete.

2. Perform a proportional stretch on the schedule resulted from the previous step with the factor C/A, where A is the allowed execution time for the whole task graph.

A consequence of this algorithm is that all tasks on a certain processor will execute at the same speed. Although an easy method to implement, LP-PS performs optimally only under specific circumstances. From the analysis of the proportional stretch transformation given in Section 5.2, we deduce that LP-PS is optimal for homogeneous systems and directed task graphs with all paths equal in length[1]. Note also that it must be free of non-zero delay

[1]The graphs do not need to yield 100% utilization, but only to have all paths critical.

communications, for the optimality claim to hold. For homogeneous systems, where the processors have different power characteristics, or containing non-scalable operations (e.g., non-zero delay communications), or the graph contains non-critical paths, LP-PS fails to employ certain possibilities for energy reduction (see Figure 5.2 a and b).

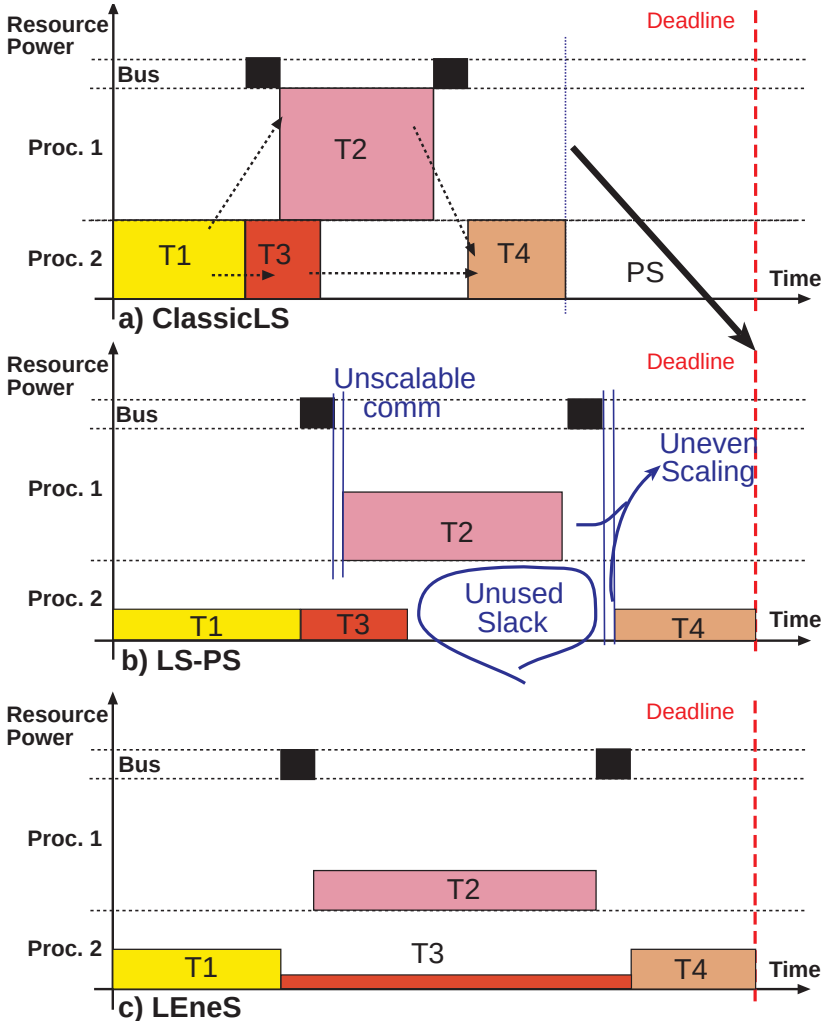


Figure 5.2: Scheduling a simple task graph with three different strategies.

We designed a better algorithm, called LEneS, that takes into account both issues mentioned before, as detailed next.

5.4 The LEneS Algorithm

In this section we present an off-line, non-preemptive **L**ow-**Ene**rgy **S**cheduling (LEneS) algorithm designed for Enhanced Task Graphs (ETG). The ETGs are already assigned to resources, which are variable speed processors. Being an off-line (static) scheduling algorithm, LEneS regards tasks as having a unique execution pattern. The algorithm also works for tasks with probabilistic execution pattern, if we employ only the tasks' WCE.

LEneS is based on a list-scheduling algorithm, and uses a priority function sensitive on both critical path and energy. In every scheduling step, the node priorities change and have to be recalculated. Moreover, this priority function is tuned during several scheduling attempts. Whenever a scheduling attempt fails (the deadline is violated), the priority function is adjusted and a re-scheduling is attempted.

Next, the scheduling problem is defined more precisely. Then we provide a background for understanding the priority function, followed by the description of the priority function used by LEneS. We describe the method for tuning this priority function, and finally we show some experimental results.

5.4.1 Scheduling an Enhanced Task Graph

The ETG, as described in Section 3.2.2, is especially useful in representing tasks on variable speed processors, since the duration of a task may vary with the processor speed. In an ETG, the nodes represent start-task and end-task events that have to take place at certain time moments. Scheduling an ETG means finding these time moments while respecting the dependencies and timing constraints:

Definition 5.2. Given a deadline d, a **schedule** for an enhanced task graph $\Gamma_E = (N_{start} \cup N_{end}, V')$ is an assignment of time moments to nodes (a mapping $\xi : N \rightarrow \Re$, where $N = N_{start} \cup N_{end}$) such that $\forall v_{ij} \in V', \xi(i) + w_{ij} \leq \xi(j)$ and $\forall k \in N, 0 \leq \xi(k) \leq d$.

Here, w_{ij} is the weight of the edge between nodes i and j, if it exists, and 0 otherwise. These weights denote the timing constraints as minimal delays between two events. For example, the minimal delay between a start and an end event of the same task is given by the task WCE at the highest processor speed.

Beside dependencies and timing constraints, a schedule must also respect resource constraints. In this case, since the ETG is already assigned, the only constraint is that tasks using the same processor should have non-overlapping execution times. Formally, if tasks $\tau_1 = (start_1, end_1)$ and $\tau_2 = (start_2, end_2)$ are assigned to the same resource, $[\xi(start_1), \xi(end_1)) \cap [\xi(start_2), \xi(end_2)) = \varnothing$. Task graph assignment is discussed in more detail in Chapter 6.

Starting from the definition of a schedule, we also define the concept of *partial schedule* of an ETG. A partial schedule is a set of all possible schedules, given that a certain node can be scheduled anywhere inside a certain time interval:

Definition 5.3. Given an ETG $\Gamma_E = (N, V')$ and a deadline d, a **partial schedule** is an assignment of an interval $[a_i, b_i]$ to each node $i \in N$, where $0 \leq a_i \leq b_i \leq d$, and ,$\forall k \in N \wedge \forall t_k \in [a_k, b_k]$, $\exists \sigma_k = \{t_j | j \in N, j \neq k, t_j \in [a_j, b_j]\}$ such that $\xi = \sigma_k \cup \{t_k\}$ is a schedule.

From this definition it is clear that two partial schedules may contain common schedules. Furthermore, we can define the following relation between two partial schedules:

Definition 5.4. A partial schedule ξ^1 **covers** another partial schedule ξ^2 if $\forall i \in N$, $a_i^1 \leq a_i^2 \wedge b_i^2 \leq b_i^1$, where a_i and b_i have the same meaning as in definition 5.3, and the superscripts identify the partial schedule. ξ^2 is, in this case, a subset of the set of schedules represented by ξ^1.

The idea of the LEneS algorithm consists in choosing, in each scheduling step, a partial schedule covered by the old one. The chosen partial schedule should have the smallest possible energy compared to all the other partial schedules. We define the energy consumption associated with schedules and partial schedules in Section 5.4.2. The starting partial schedule must be one covering all the possible schedules. For this, we start by performing the As Soon As Possible (ASAP) and As Late As Possible (ALAP) scheduling for the ETG, without resource constraints.

Definition 5.5. The ASAP **schedule** of an ETG, $\Gamma_E = (N, V')$ is the schedule in which the exact time moments for scheduling each $i \in N$ node are given by $\xi(i) = max\left(\{0\} \cup \{\xi(j) + w_{ji}\}_{v_{ji} \in V'}\right)$, also denoted by ASAP_i.

Definition 5.6. The ALAP **schedule** of an ETG, $\Gamma_E = (N, V')$ is the schedule in which the exact time moments for scheduling each $i \in N$ node are given by $\xi(i) = min\left(\{d\} \cup \{\xi(j) - w_{ij}\}_{v_{ij} \in V'}\right)$, also denoted by ALAP_i.

Finally, for an ETG and a certain deadline d, no node can be scheduled outside its $[\text{ASAP}_i, \text{ALAP}_i]$ interval, also referred to as *node mobility*. Therefore, it is sufficient to use these values as the initial values for our LEneS approach, to make sure all possible schedules are considered during scheduling.

5.4.2 The Average Energy of a Schedule

Since we are interested in finding the schedule with a minimal energy consumption, we need to define what is the energy of a schedule. For this reason

we start from the energy expression of individual tasks. The energy consumed by the task can be approximated by a piece-wise linear dependency on its execution time as discussed in Chapter 3. Given the energy function we define *the average energy of an* ETG *node*. We consider that the start-nodes have zero energy, while the average energy of the end-nodes models the full task energy. Given that an end-node can be scheduled anywhere inside a certain time interval $[a, b] \in [\text{ASAP}, \text{ALAP}]$, its average energy is:

$$\overline{E_{[a,b]}} = \frac{1}{b-a}\int_a^b E(t)\mathrm{d}t \quad \text{ASAP} \leq a < b \leq \text{ALAP} \tag{5.10}$$

We consider the average energy over an interval $[a, b]$ for a certain end-node, as a measure of the quality of the set of solutions obtained by scheduling that end-node anywhere in $[a, b]$. At limit, when a gets closer to b, $\overline{E_{[b,b]}} = E(b)$ is the energy yielded by scheduling that node exactly at moment b. For a given node, we are able to compare different possible time intervals, or sets of solutions using the average energy as a measure.

The notion of average energy can be extended to sets of schedules of an ETG by using partial schedules. We define the *average energy of a partial schedule* using the average energy of a node as:

$$\overline{E} = \sum_{i \in N} \overline{E_{[a_i,b_i]}} \tag{5.11}$$

Since all the start-nodes are considered to have zero energy, the sum given above ultimately involves only the end-nodes.

We can now compare any two partial schedules, using their energy consumption. Given two partial schedules, we consider that one is better than the other if it has lower average energy.

Finally, the scheduling starts from the partial schedule covering all others, which is obtained by using the ASAP-ALAP intervals for the given deadline d. We will denote the energy of this initial partial schedule by $\overline{E_{[0,d]}}$. Then, for each time moment t from 0 up to d, decisions are taken that change the partial schedule. We will denote the partial schedule for a certain time moment t (also referred to as scheduling step), by $\overline{E_t}$. The exact algorithm used to restrict the partial schedule to a schedule is detailed in Section 5.4.4. Next we focus on the priority function used by this algorithm.

5.4.3 The Priority Function

The priority function for a node reflects the energy gain (or loss) induced by a specific scheduling decision. At a certain time step t, there are nodes (with index i) which are eligible for scheduling. If they are delayed by a certain time ϵ, their mobility will change from $[t, e_i]$ to $[t + \epsilon, e_i]$. This change can propagate

to the nodes ordered after them. For each of the eligible nodes at a certain scheduling step, delaying the node by ϵ yields a new partial schedule, with a corresponding average energy, $\overline{E^i_{t+\epsilon}}$. We are interested in the partial schedule yielding the most significant energy reduction. Therefore, our priority function for a node i, which is about to be scheduled at a certain time step t, is computed as the difference between the average energy of the current partial schedule, $\overline{E^i_t}$, and the one obtained by scheduling node i later, $\overline{E^i_{t+\epsilon}}$. In the special case when the moment t is the latest possible moment, the node must be scheduled, so its priority becomes infinite:

$$f(i,t) = \begin{cases} \overline{E^i_t} - \overline{E^i_{t+\epsilon}} & \text{if } \text{ALAP}_i > t \\ \infty & \text{otherwise} \end{cases} \tag{5.12}$$

A negative priority means that it is better to schedule the node later, while a positive value means that it is better to schedule the node at that very moment. The priority function presented above considers only the energy aspect, and may fail to lead to feasible schedules, especially when the deadline is tight. To be able to find schedules even for tight deadlines, we used the following, improved priority function:

$$g(i,t) = f(i,t) + \alpha_i \frac{|f(i,t)|}{d - t - cp_i} \tag{5.13}$$

where d is the deadline and cp_i is the delay of the longest path starting in node i (for the fastest processor speed). Each node has an associated coefficient α_i, which controls the emphasis on lowering energy vs. generating a tight schedule. Having a different α for each node allows us to treat the nodes on the critical path in a different manner, focusing, for those nodes, more on fulfilling the deadline than on lowering the energy. With the priority given above, if all α_i are large enough, the priority function behaves closer to a classic, critical-path priority. Moreover, the set of smallest α_i for a given graph and certain deadline yields the lowest energy consumption for that graph and deadline. Next, we describe the method of tuning the values of the α set.

Tuning the Priority Function

Depending on the values for the α coefficients, it can happen that no schedule is found. In that case, the αs for the nodes on the critical path are increased, thus emphasizing the timing aspect of the priority function. A new scheduling is attempted with the new α values. In the worst case, all α reach their maximal value, MAX_α, set by the designer, and the $g(t,i)$ priority function becomes similar to a classic critical-path priority function.

5.4.4 LEneS Algorithm Pseudo-code

A pseudo-code description of the LEneS algorithm is given in Figure 5.3. The algorithm consist of list-scheduling on ETG, ETG_LS, using our priority function, wrapped in a tuning algorithm for the α_i coefficients.

The algorithm starts by setting all the α_i coefficients to 0. A null α coefficient in equation 5.13 means that the node priorities will be based only on their possible impact on energy consumption. Thus, LEneS attempts to schedule the ETG for the current set of coefficients, using the ETG_LS procedure described in detail in the next section. If this fails, the deadline appears to be too tight to accommodate a loose (low speed) critical path. The algorithm then increases the influence of timing on the priority for the nodes on the critical path. This is done by increasing the α coefficients for the nodes on the critical path. If all nodes on the critical path have already reached the maximum value for α and no schedule was found, non-critical paths must be made more sensitive on timing, since they might indirectly prevent the task graph from finishing before the deadline. This is done by increasing all α coefficients in the graph. Finally, if no schedule is found even when all the coefficients reached the maximum value, the algorithm concludes that there is no possible schedule. This situation is in fact equivalent to scheduling using a classic list-scheduling with critical-path as priority function. If the classic list-scheduling cannot find a schedule, then neither can LEneS. Note that the algorithm focuses in the beginning only on energy consumption and, if no schedule is found, focuses more and more on timing, by increasing the α coefficients. The result of this procedure is a set of α which yields low energy schedule under the given timing constraint.

```
boolean LEneS(ETG , deadline) {
  set all α_i ← 0;
  while(not ETG_LS(ETG , g, deadline)) {
    let {α_k} be the coefficients of the nodes on the critical path;
    if(all α_i ≥ MAX_α) return false;
    else if(all α_k == MAX_α) increase all α_i with ε;
    else increase only all α_k with ε;
  }
  return true;
}
```

Figure 5.3: The Full LEneS Algorithm

List-Scheduling on ETGs

As mentioned before, our LENES algorithm is based on list-scheduling adapted for tasks running on variable speed processors. Scheduling a task on these kind of processors means deciding at least two parameters: the start of task execution and the speed of the processor (alternatively the end of task execution). Therefore, ETGs are well suited for our problem. Since ETG nodes are events describing the start and end of execution for a task, one has to be careful to schedule new tasks (start events) only when the processor is free (after end events). This is one of the major adaptations of list-scheduling in our scheduling algorithm, LENES. Another adaptation is the method of advancing the time in each scheduling step (t in the listing from Figure 5.1). In principle, list-scheduling on fixed speed processors may *jump* at the end of a task, since the execution time is unique and known. However, for variable speed processors, a task may finish after different time intervals, depending on the running speed. Even for processors with a limited number of speeds, the task execution time may vary smoothly, as shown in Section 3.3.2. Therefore, the time should vary almost continuously wherever a scheduling decision can be made. In a real implementation, one may choose to vary the time in sufficiently small time steps (e.g. the length of the clock at the highest speed). The drawback of choosing too big time steps is that the algorithm might find less efficient schedules. On the other hand, choosing too small time steps leads to longer execution times. The algorithmic complexity of the whole LENES algorithm is investigated in Section 5.4.5. A third important characteristic of our LENES algorithm is the adaptive priority function, extensively described in Section 5.4.3.

The specific list-scheduling algorithm used by LENES is presented in Figure 5.4. The ETG nodes are scheduled iteratively on free resources. Whenever a start node is scheduled on a certain resource, that resource is removed from the free resources list. Once the corresponding end node has been scheduled, the resource becomes free again. Note that the algorithm first attempts to schedule end nodes, in the hope that more resources become free or more start nodes become ready. Only nodes with positive priorities are considered for scheduling. A positive priority means that there is an decrease in energy when the node is scheduled earlier rather than later. Moreover, for each resource only the node with the highest positive priority is scheduled at one time. Note also that time advances in constant and relatively small steps ϵ. Treating time in this pseudo-continuous manner is necessary in our case, since tasks may finish at any moment, depending on the processor speed. Ideally the nodes should be scheduled at the precise time moment when their priority becomes zero, if initially was negative. In practice, finding these exact time moments is difficult, resting on an accuracy versus speed trade-off. Using a constant step as time increment turned out to be a sufficiently exact

method for our case, if the step is chosen wisely.

Finally, if the time scale passes the deadline and there still are unscheduled nodes, the algorithm ends. This means that the ETG may not be scheduled before the given deadline for the current priority function parameters. If all nodes are scheduled by the time we reach the deadline, the algorithm succeeds.

```
boolean ETG_LS(Γ(N_start ∪ N_end,V), f, deadline) {
  R_free ← all; t ← 0; Σ ← ∅;
  do {
    /* extract all end task nodes and compute their priority */
    N_end,ready ← {i | i ∈ N_end \ Σ ∧ ∀ j ∈ Pred(i, Γ), j ∈ Σ ∧ t_j + w_ij ≤ t};
    for each node i ∈ N_end,ready calculate p_i ← f(i, ...);
    /* keep the nodes that have a positive priority */
    N+_end,ready ← {i | i ∈ N_end,ready ∧ p_i > 0};
    for each i ∈ N+_end,ready {
      /* schedule the node now, on its assigned resource */
      t_i ← i; Σ ← Σ ∪ {i}; R_free ← R_free ∪ {Res(i)};
    }
    /* extract all start task nodes and compute their priority */
    N_start,ready ← {i | i ∈ N_start \ Σ ∧ ∀ j ∈ Pred(i, Γ), j ∈ Σ ∧ t_j + w_ij ≤ t};
    for each task i ∈ N_start,ready calculate p_i ← f(i, ...);
    /* keep those that have positive priority */
    N+_start,ready ← {i | i ∈ N_start,ready ∧ p_i > 0};
    L_ready ← Order N+_start,ready by decreasing p_i;
    for each node i ∈ L_ready from head to tail
      if(Res(i) ∈ R_free)      {
        /* schedule the node now, on its assigned resource */
        t_i ← i; Σ ← Σ ∪ {i}; R_free ← R_free \ {Res(i)};
      }
    t ← t + ε;
  } while(t < deadline);
  Try to schedule all remaining end task nodes;
  if(Σ ≠ N_start ∪ N_end) return false;
  else return true;
}
```

Figure 5.4: The adapted List-Scheduling used by LEneS

5.4.5 LEneS Evaluation

There are two important characteristics of our LEneS approach that have to be investigated. First, we are interested in the algorithmic complexity or the computational time required to carry out our algorithm. Although this is less important for an off-line algorithm such as LEneS, it is still decisive for achieving a fast design space exploration. Secondly, the essential characteristic of an energy-aware algorithm is its efficiency in finding low energy solutions. We address both of these characteristics next.

Algorithmic Complexity and Execution Time

The complexity analysis of the LEneS algorithm shows that it has a computational complexity of $O(VMN^3log(MAX\_\alpha))$, where N is the number of nodes in the ETG, M is the number of time steps in the tightest deadline, V is the highest number of speeds supported by a processor, and MAX_α is the maximal value allowed for the α's. This expression is confirmed if we look at the time required for scheduling for different settings.

The following experiment evaluates the LEneS algorithm from the scheduling speed point of view. The results are depicted in Figure 5.5. The points in the base plane of the 3D graph reflect the system configuration: number of processors and the distribution of tasks on the processors. For each of the configurations, we generated hundred random task graphs and then used LEneS to schedule them, obtaining an average scheduling time. The average time needed to perform the scheduling is represented on the vertical axis. Using interpolation, we obtained the dotted curves on the surface. The curves mark different time levels, ranging from 1 second to 10 minutes. For example, for a task graph of 56 tasks, evenly distributed on eight processors (seven tasks per processor), LEneS will require around 5 minutes to find the schedule. For scheduling the largest type of graphs (hundred nodes on ten processors), LEneS requires around 24 minutes. For this experiment we assumed that the processors can run at three different speed settings. We considered the supply voltages for these settings to be 3.3V, 2.1V, and 0.9V. The clock frequencies at each voltage were obtained using a realistic CMOS delay-voltage dependency (Section 2.1.1) with threshold voltage set at 0.4V. The values depicted in Figure 5.5 were obtained by running LEneS on a Sun Ultra 10 workstation (440MHz UltraSparcIIi processor, 256MB RAM).

In some cases, the long execution time for large designs makes our method suitable only for final scheduling, and not for fast evaluation inside a design-space exploration loop. Yet, this drawback can be overcome, if LEneS is combined with a simpler scheduling strategy or a fast estimator, as we point out in Chapter 6.

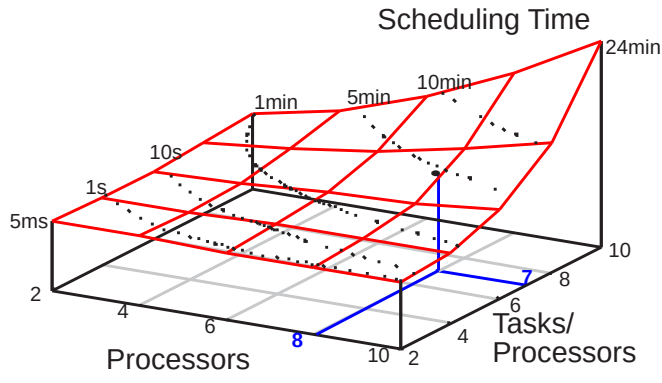


Figure 5.5: Execution time required by performing LEneS on various architectures and ETGs (carried out on a 440MHz UltraSparcIIi processor, 256MB RAM).

Energy Savings

The next set of experiments inspects the energy saving capability of the LEneS algorithm compared to the classic list-scheduling with critical-path based priority function (ClassicLS). For several system configurations, as in the previous experiment, we scheduled the ETGs using both LEneS and ClassicLS. For the tightest schedule length, we compared the energies consumed by the two solutions, obtaining a surface similar to the one in Figure 5.5. For clarity, we depicted in Figure 5.6 only the projection of the levels on the horizontal plane instead of the whole 3D graph. For this experiment, we assumed that we use processors with four speed settings (supply voltages of 3.3V, 2.5V, 1.7V, and 0.9V with corresponding realistic frequency values). Note that the saved energy can be as high as 28% when using LEneS as opposed to ClassicLS. For architectures with two voltage processors (3.3V and 0.9V), using a similar experiment we obtained slightly smaller energy savings. In the majority of the cases, the saved energy was four times smaller compared to the four supply voltage processors. This difference comes from the fact that for a two speed processor, the energy-delay curve (Figure 3.3) is a worse approximation of the ideal one, compared to the case of a four speed processor.

Note that the saved energy increases with the degree of parallelism (more processors or less tasks per processor). This comes from the fact that the percent of tasks on the critical-path decreases. In this case, there are potentially more tasks which can run slower, and thus save energy. The critical-path length is also influenced by the assignment of tasks to processors, not only by the dependencies in the task graph. An unbalanced assignment can overload

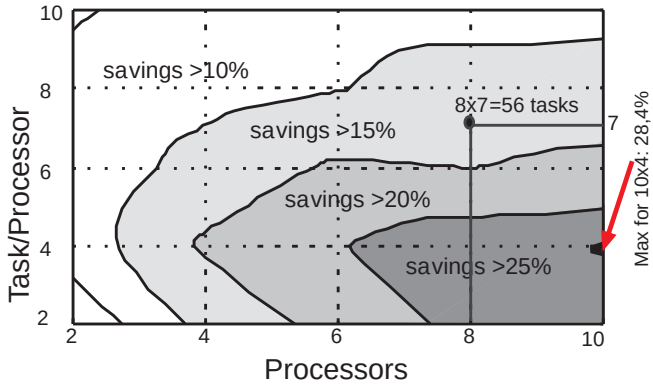


Figure 5.6: Energy savings obtained via LENES compared to ClassicLS on different size architectures and ETGs.

a processor unnecessarily, increasing also the critical-path. In these cases, our LENES method performs extremely well since it can take advantage of the idle processors. However, if the processors are perfectly balanced, LENES behaves as ClassicLS. A more detailed analysis of the influence of assignment on scheduling can be found in Chapter 6.

In the experiments presented until now, we assumed that we always have to execute the task graph as fast as possible. In reality, in most of the cases the deadlines are given as design requirements. Often the designers do not require the tasks to execute as fast as possible. In these cases, there is a time slack which can be used to further reduce the energy. The next experiment explores the behavior of LENES in these more relaxed cases. We considered three scheduling methods: the classical list-scheduling with critical path as priority (ClassicLS), list-scheduling with proportional stretch (LS-PS) from Section 5.3, and LENES with proportional stretch (LENES-PS) as described in Section 5.2. For various extensions of the tightest deadline, we compared the energy saved by using LENES-PS over the other two approaches for a number of random task graphs. In Figure 5.7 we depicted the curves obtained by averaging the results for two sets of thirty random graphs (TG_1 and TG_2). Both sets contain task graphs of thirty nodes, but the degree of parallelism differs, representing two extremes. TG_1 uses ten processors (high parallelism), while TG_2 uses only three processors (low parallelism). The continuous curves show the energy saved by LENES-PS over LS-PS. The dotted ones show the energy saved by LENES-PS over ClassicLS. Although, LS-PS performs well, being able to save around 60% energy at 50% deadline extension compared to ClassicLS, LENES-PS performs best by saving 7–28% energy, compared to LS-PS.

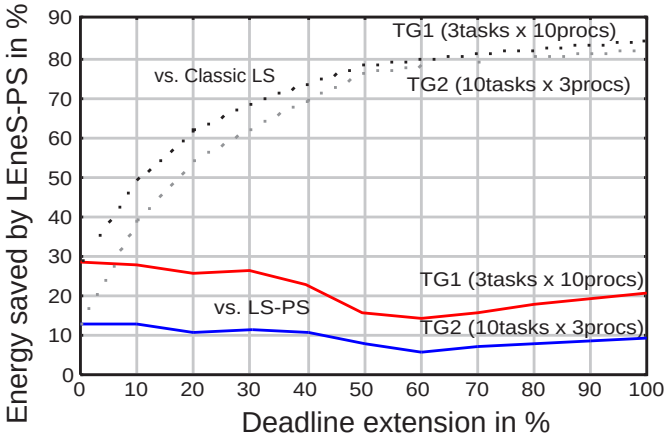


Figure 5.7: Relative energy savings over ClassicLS for LEneS-PS and LS-PS for different deadlines.

The final experiment explores the potential energy savings for a real-life application. The sub-system we are interested in is an optical flow detection (OFD) algorithm, which is part of a traffic monitoring system (see [DGK+00]). In this implementation, the optical flow algorithm consists of 32 tasks, running on two ADSP-21061L digital signal processors. Limited by other tasks, OFD can process an image of 78×120 pixels at a rate up to 12.5Hz. The estimated energy consumption for one iteration of the OFD implementation at this rate is 27.2mJ. Depending on traffic or monitoring altitude, such a high processing rate is a waste of resources. For instance, at a high altitude and sparse traffic, the bird's-eye view changes very slowly, therefore, requiring infrequent updates. In many cases, a rate of 2Hz is sufficient, which means approximately six times deadline extension for one iteration of the OFD. Important energy savings would be obtained if the design were to use processors supporting multiple voltages. Moreover, using the approach we presented here, the schedule can be dynamically adapted at run-time to fit the desired deadline, depending on the required processing rate. Assuming that the DSPs can operate at 3.3V, 2.5V, 1.7V, or 0.9V, we applied the LEneS method for the OFD. The results show as much as 83% processor energy saving possibility for a sampling rate of 2Hz. For that rate, all the tasks are using the two lowest possible voltages. Overall, the processors have to run 42.48% of the time at 1.7V, and the rest of the time at 0.9V. For this particular example, the difference between the energy obtained through LEneS-PS and LS-PS was less than 4%. This difference is rather small because of the reduced number of tasks

executing off the critical path, and which anyway have little freedom to scale. Note also that as the deadline extension grows, the tasks use lower voltages. For lower supply voltages, the energy-delay dependency (see Figure 3.3) has a smaller slope, so longer delays yield gradually smaller energy savings. At limit, all tasks execute at the lowest voltage, and further deadline extension will not bring new energy savings.

5.4.6 Conclusions

In this section we introduced a new static scheduling method, LEneS, that handles designs with dependent tasks mapped onto dynamic supply voltage processors. LEneS is based on list-scheduling and uses a special priority function to derive schedules with low energy consumption. Using a method for tuning its priority function, our algorithm is able to find schedules that are more energy efficient than other energy-aware scheduling strategies. The experiments we present show that even for the tightest possible deadline, up to 28% energy savings can be obtained without any performance loss, using our scheduling approach. For loose deadlines, LEneS can be combined with the scaling method from Section 5.2, leading to important energy savings. In particular, using this technique, we can obtain 77% energy savings over a critical-path priority based list-scheduling, for a 50% deadline extension.

5.5 Maximum Required Speed Approach

The Maximum Required Speed (MRS) method presented in this section is designed as an off-line step for task sets scheduling. The main idea of this approach rests on reaching a close to 100% processor utilization by identifying tasks that can run slower. More precisely, MRS is not a scheduling algorithm in itself, but a procedure to compute the smallest possible processing speeds for which a task set is still schedulable with a specific on-line algorithm. At run-time, the task set is still scheduled according to the on-line algorithm, but the processor speeds are set using the off-line computed values. Note that the speed choice must not affect the feasibility of the schedule. For tasks with fixed execution time, the only modification of the non-DVS run-time algorithms resides in adding a speed switching sequence. We focus here on two MRS instances, distinguished by classic run-time scheduling methods: Earliest Deadline First (EDF) and Rate Monotonic (RM).

5.5.1 MRS for EDF scheduling

Given a set of tasks with arbitrary periods, deadlines, and execution times, the EDF schedule feasibility necessary condition is that the processor utiliza-

tion (as in Definition 3.4) be not larger than 1 ([BMR90]). In this general case, the $U \leq 1$ condition is not sufficient, meaning that there might not exist any schedule for the task set in question. The direct implication on our speed scheduling problem is that any modification of the execution time might lead to a task set that cannot be scheduled. We examine next a few restrictions of the general task set model.

Deadlines equal to periods

The $U \leq 1$ turns out to be also a sufficient condition for synchronous [LL73] and asynchronous [Cof76] periodic task sets, with deadlines equal to periods ($\forall \tau_i$, $D_i = T_i$). This means that one can run the processor slower, such that the actual utilization approaches 1 and at the same time reduce the power consumption while keeping the feasibility of the schedule. Finally, in this particular case, the maximum required speed for EDF is common for all tasks and equal to the processor utilization for the reference (maximal) speed:

$$s_{\text{EDF-MRS}} = U \tag{5.14}$$

Since this technique is straightforward, it is commonly adopted in EDF with DVS scheduling ([PS01, SC01a]). In fact, the energy yield by EDF-MRS turns out to be the lower bound of the energy consumption for a given task set, since it uses up the processor time entirely and runs all tasks at a common, smallest speed (as discussed in Sections 4.2, 5.1, and 5.2).

Deadlines shorter than periods

Unfortunately, for task sets where $D_i \leq T_i$, which is a simple extension of the $D = T$ model given above, the feasibility analysis is much more difficult, becoming intractable ([LM80, BRH90]). Consequently, finding optimal speeds — which is equivalent to examining families of task sets — is an even more difficult problem. Yet, upper bounds for maximum required speeds can be found using sufficiency conditions. For example, an easy to compute upper bound for the maximum required speed can be deduced from the sufficient condition described by Theorem 3.11 from [SSRB98]:

$$\sum_{i=1}^{N} \frac{\mathcal{C}_i}{\min\{D_i, T_i\}} \leq 1 \tag{5.15}$$

where $\mathcal{C}_i$ refers to the time required by the worst case execution pattern WCE at the reference (and fastest) frequency f_{ref}: $\mathcal{C}_i = \text{WCE}_i / f_{ref}$. One may thus allow lower frequencies as long as this condition is not violated, and the task

set remains feasible under EDF scheduling. Finally, the speed upper bound in this case becomes:

$$\overline{s_{\text{EDF-MRS}}} = \sum_{i=1}^{N} \frac{\mathcal{C}_i}{\min\{D_i, T_i\}} \tag{5.16}$$

Note that when deadlines are equal to periods this upper bound overlaps with the value of $s_{\text{EDF-MRS}}$ given before. It is also possible that this upper bound computes to a value greater than 1, even if the task set can be scheduled under EDF. For these situations, $\overline{s_{\text{EDF-MRS}}}$ has little practical use, and other methods have to be employed for finding better speed bounds. For more restricted task set models, there are approaches that instead of finding a unique speed for all tasks, compute specific speeds for each and every task. We examine one of these next.

Same period for all tasks

For task sets with deadlines shorter than their periods ($D_i \leq T_i$), but all having the same period ($\forall \tau_i, \tau_j,\ T_i = T_j$), the EDF schedule feasibility problem is solvable in polynomial time (the Sequencing with Release Times and Deadlines, SS1, with preemption problem from [GJ79]). For tasks with different arrival times (*asynchronous*), Yao et al. [YDS95] present an off-line algorithm for computing the processor speeds yielding the lowest energy consumption. A similar but simpler strategy can be applied to sets of tasks with the same arrival times (*synchronous*). We present such a method in the following. First we give a few useful definitions, adapted from [SSRB98].

Definition 5.7. Given a set of real-time jobs running on a variable speed processor, the **processor demand** of the job set on the interval $[t_1, t_2)$ is

$$h_{[t_1,t_2)} = \sum_{t_1 \leq D_k < t_2} \frac{C_k}{f_k} \tag{5.17}$$

where each job executes its C_k clock cycles at a specific clock frequency f_k.

In other words, the processor demand is the amount of computation time, at various frequencies, required by the jobs that can execute after t_1 and have to finish before moment t_2.

Definition 5.8. Given a set of real-time jobs running on a variable speed processor, the **loading factor** on the interval $[t_1, t_2)$ is the fraction of the interval needed to execute its job, that is, $u_{[t_1,t_2)} = h_{[t_1,t_2)}/(t_2 - t_1)$

The important intervals in our case are those delimited by the deadlines D_k, $k = 1 \ldots N$, and the common arrival time 0. Intuitively the necessary condition for a feasible schedule is for all of these N loading factors to be smaller

than 1. In fact this is also a sufficient condition, as proven in [SSRB98] (Theorem 3.5). Finally, this means that we can tweak with the clock frequency while maintaining the feasibility of the EDF schedule if the following condition remains true:

$$\sup_{0 \le t_1 < t_2 \le D_N} \{u_{[t_1,t_2)}\} \le 1 \tag{5.18}$$

Using the above observation, we arrive at a specific algorithm for assigning clock frequencies to tasks, given in Figure 5.8. The algorithm iteratively increases the loading factors to the maximum possible, such that the feasibility condition remains true. It starts by computing the loading factors for the intervals of type $[0, D_k)$ using the maximum frequency f_{max} for all tasks. If i is the index of the largest loading factor u_i, this corresponds to interval $[0, D_i)$. This interval contains the most demanding sequence of jobs, namely $\tau_1 \dots \tau_i$. Using a frequency of $f_i = f_{max} u_i$ for these tasks will still produce a feasible schedule (all loading factors remain less or equal to 1). The new loading factor for the task sequence up to τ_i becomes in fact 1. This also means that no other tasks may execute during the $[0, D_i)$ interval. We can then consider that we have to solve a new problem: that of scheduling tasks $\tau_{i+1} \dots \tau_N$ during the interval $[D_i, D_N)$. Note that no matter how we affect these lower priority tasks, they will have no influence on the higher priority tasks and the decision on their clock frequency thereof. In the new problem, by modifying all deadlines to $D'_j = D_j - D_i$ we arrive at a similar problem as the one we started with, except with a reduced number of tasks. We can now re-iterate the procedure for this reduced problem. All this continues until we cannot build a reduced problem, since there are no tasks left. At this point we assign frequencies to all tasks. The supply voltage is then adjusted to the optimal one for each frequency. This procedure is in fact quite similar to the one we use for RM-MRS, described in Section 5.5.2. The algorithm presented in Figure 5.8 can be made more efficient, once we notice the dependency between loading factors computed for a given problem. Namely, the loading factors corresponding to lower priority tasks contain the loading factors for the higher priority tasks. Considering the loading factors for tasks τ_p and τ_{p+k}, this dependency can be written as:

$$u_{p+k} = \frac{u_p D_p + \sum_{p<j\le p+k} C_j / f_j}{D_{p+k}} \tag{5.19}$$

The loading factors can be, thus, computed incrementally. Finally, the worst case time complexity of an improved algorithm turns out to be of order $O(N^2)$.

Example 5.1 (EDF-MRS for a set of five tasks):
Consider the set of five tasks given in Table 5.2 that has to be scheduled on a variable speed processor with the maximum clock frequency of 1GHz. Following the algorithm from Figure 5.8 we compute the maximum required speeds for all tasks. Compared to the single, maximal speed case the energy

```
edf_mrs_sameT({τ1,...,τN }) {
  for i = 1 to N do fi ← fmax;
  q ← 1; δ ← 0;
  do {
    for i = q to N do
      ui ← (Σ_{0<j≤i} Cj/fj) / (Di − δ);
    p = index of max_{p≤i≤N}{ui};
    δ = Dp;
    for i = q to p do
      fi ← fmax*up;
    q ← p + 1;
  } while(p < N);
  return {fi}
}
```

Figure 5.8: An algorithm for detecting maximum required speeds for an EDF schedule of set of tasks with common period and different deadlines

Table 5.2: EDF-MRS applied on a five tasks set with common period (T = 20ms)

Task			Loading Factor		Final f
#	C($\times 10^6$)	D(ms)	1st iteration,$\delta = 0$	2nd iteration,$\delta = 9$	(MHz)
1	1	4	$1/4 = 0.25$	-	667
2	3	8	$(1+3)/8 = 0.5$	-	667
3	2	9	$(\mathbf{4+2})/\mathbf{9} = \mathbf{0.(6)}$	-	667
4	1	14	$(6+1)/14 = 0.5$	$1/5 = 0.2$	364
5	3	20	$(7+3)/20 = 0.5$	$(\mathbf{1+3})/\mathbf{11} = \mathbf{0.364}$	364

consumption is reduced, and can be computed as follows. First, we assume the common dependency between energy and speed, as given in equation 3.8. Namely, the energy depends quadratically on speed. In our case 60% of the work (6×10^6 cycles out of 10×10^6) will execute at a speed of 0.667 (667MHz / 1GHz). The rest will execute at a speed of 0.364 (364MHz/1GHz). Finally, the energy consumption of the MRS schedule, as a percentage of the maximum speed schedule energy is:

$$E_{\text{MRS}}/E_{max} = 0.6 * 0.667^2 + 0.4 * 0.364^2 \approx 0.32$$

In conclusion, our EDF-MRS schedule saves in this case about 68% energy compared to the single, maximum speed schedule.

5.5.2 MRS for RM scheduling

For RM scheduling, computing the right speeds is more complicated than for the EDF. First, one cannot directly employ the utilization at the maximum speed as the new common speed for all tasks, as in EDF since the task set might become un-schedulable. One may choose to use as a limit the utilization imposed by the condition proposed by Liu and Layland in [LL73]. With this approach the maximum required speed is unique and equal to:

$$s_{\text{RM-MRS}} = \frac{U}{N(2^{1/N} - 1)} \tag{5.20}$$

At a closer look, the schedule feasibility condition proposed in [LL73] is a sufficient one and covers the worst possible case for the task group characteristics. An exact analysis as proposed in [LSD89] may further reveal possibilities for stretching tasks while still meeting the deadlines. Based on this, [SC99] describes a method to compute the maximum required frequency (speed) for a task set. We go even further, and instead of computing a single common maximum required speed for the whole task set $\{\tau_i\}_{i=1...N}$, as in [SC99], we compute individual speeds for each task τ_i. Note that the speed required by a task is inverse proportional to the task stretching factor. Finding the maximum required speeds is in fact equivalent to finding the minimal stretching factors $\{\alpha_i\}_{i=1...N}$. We focus on computing the α factors.

As introduced by equation 3.1, we model a task as a triple including the task deadline D_i and period T_i. Since MRS is a static method, the third relevant task parameter in our case is the worst case execution time (WCET). The task WCET, denoted by $\mathcal{C}_i$ in the following, refers to the time required by the worst case execution pattern WCE at the reference (and fastest) frequency f_{ref}: $\mathcal{C}_i = \text{WCE}_i/f_{ref}$. Note that for a task with a unique execution pattern, where BCE = WCE = C, WCET can also be written $\mathcal{C}_i = C_i/f_{ref}$. Furthermore, we consider that the tasks in the group are indexed according to their priority, computed as in RM-scheduling ($1/T_i$).

We compute the stretching factors in an iterative manner, starting from the highest priority tasks and continuing with lower priority tasks. Consider that index q points to the latest task which has been assigned a stretching factor. Initially, $q = 0$. Each of the tasks $\{\tau_i\}_{q<i\leq N}$ has to be executed before one of its scheduling points S_i as defined in [LSD89]:

$$S_i = \left\{ kT_j | 1 \leq j \leq i \wedge 1 \leq k \leq \left\lfloor \frac{T_i}{T_j} \right\rfloor \right\} \tag{5.21}$$

The above equation defines all the scheduling points when the deadlines are equal to task periods, $T_i = D_i$. For task sets where $T_i \neq D_i$, we need to change the set of scheduling points according to:

$$S'_i = \{t | t \in S_i \wedge t < D_i\} \cup \{D_i\} \tag{5.22}$$

Task τ_i exactly meets its deadline if there exists a scheduling point $S_{ij} \in S_i$ for which the following relation holds:

$$\sum_{1 \le r \le q} \alpha_r \mathcal{C}_r \left\lceil \frac{S_{ij}}{T_r} \right\rceil + \alpha_{ij} \sum_{q < p \le i} \mathcal{C}_p \left\lceil \frac{S_{ij}}{T_p} \right\rceil = S_{ij} \tag{5.23}$$

Note that for the tasks which already have assigned a stretching factor we used that one, α_r, while for the rest of the tasks we assumed they will all use the same and yet to be computed stretching factor, α_{ij}, which is dependent on the scheduling point. For task τ_i, the best scheduling choice, from the energy point of view, is the largest of its α_{ij}. At the same time, from equation 5.23, this is equal for all tasks $\{\tau_i\}_{q<i\le N}$. In fact, there is a task with index m for which its best stretching factor is the smallest among all other tasks:

$$\exists m, q < m \le N, \text{ such that } \max_{q<j\le m}(\alpha_{mj}) = \min_{q<i\le n}\left(\max_{q<j\le i}(\alpha_{ij})\right) \tag{5.24}$$

Note that this does not necessarily correspond to the last task, τ_N. If $q = 0$, this task sets the minimal clock frequency as computed in [SC99]. Having found the index m, all tasks between q and m can be at most stretched (equally) by the stretching factor of m. Thus, we assign them stretching factors as:

$$\alpha_r = \max_{q<j\le m}(\alpha_{mj}), r = q+1 \ldots m \tag{5.25}$$

With this, an iteration of the algorithm for finding the stretching factors is complete. The next iteration then proceeds for $q = m$. The process ends when q reaches N, meaning all tasks have been given their own off-line stretching factors. Finally, the maximum required processor speed for each task is given by the inverse of its off-line stretching factor:

$$s_i = 1/\alpha_i, i = 1 \ldots N \tag{5.26}$$

Example 5.2 (RM-MRS for a Set of Five Tasks):
This example contains the results of MRS for the set of five tasks described in Table 5.3. For this set, the task deadlines are equal to the task periods. For the RM scheduling method, the stretching factors are computed individually. Note that tasks 3 and 4 can be stretched off-line more than 1 and 2, while 5 has the largest stretching factor. The processor utilization changes from 0.687 to 0.994. Observe also that the stretching factors for the lower priority tasks require more iterations to compute. For the EDF scheduling, there is a single stretching factor, common to all tasks, equal to $1/0.687$. The maximum required processor speeds relative to the reference speed are obtained by inverting the α factors.

If we consider that the tasks have fixed execution pattern, we can easily compute the energy consumptions for the RM-MRS and EDF-MRS. For this

Table 5.3: A Numerical Example of MRS

Task			MRS α factors (and speeds)		
			RM		EDF
#	WCET (C)	Period (T)	$\alpha_{\#}$ (speeds)	iterations	α (speed)
1	1	5	1.428 (0.700)	1	.
2	5	11	1.428 (0.700)	1	.
3	1	45	1.785 (0.560)	2	1.4556 (0.687)
4	1	130	1.785 (0.560)	2	.
5	1	370	2.357 (0.424)	3	.

we found out the number of executed instances of each task over the task set hyper-period, computed as the least common multiplier (lcm) of the task periods. For our example, $lcm(5, 11, 45, 130, 370)$ is 476190. Next, we know that the energy consumed during a clock cycle is dependent on the square of the processor speed (see equation 3.8). The energy of a task instance is therefore proportional to $C_i s_i^2$. Finally, we can sum up the energy consumption after the number of instances for each task. The numerical results are detailed in Table 5.4. Note that, for this example, we assumed that no power is consumed during idle and speed switching. Also, the processor is ideal in the sense that it can run at any speed under the reference speed. It is interesting to note that, for this case, the energy consumed using RM-MRS is very close to that using EFD-MRS. Both approaches manage to save about 52% of the energy consumed by using only the classic RM and EDF employing the same, reference speed for all tasks. Moreover, the EDF-MRS energy is exactly the lower bound energy as defined by equation 5.8. Namely, the energy lower bound is U^2 of the maximum speed energy, which numerically is $0.687^2 = 0.4720$, or 47.20%.

Table 5.4: Computing the RM-MRS and EFD-MRS energy consumption for the task set in Table 5.3

Task	Instance Energy		Instances per	All Instances Energy	
#	RM-MRS	EDF-MRS	Hyper-period	RM-MRS	EDF-MRS
1	0.4904	0.4720	95238	46704.0	44952.3
2	2.4520	2.3599	43290	106147.0	102160.1
3	0.3139	0.4720	10582	3321.6	4994.7
4	0.3139	0.4720	3663	1149.8	1728.9
5	0.1800	0.4720	1287	231.7	607.5
Total Energy Consumption:				157554.1	154443.5
% from Max Speed Energy (327220.0)				48.15%	47.20%

5.6 RM Scheduling with Slack Distribution

Off-line speed scheduling methods are sufficient for tasks with fixed execution pattern. The strategies presented until now are all striving to increase the utilization by running the task slower, consuming the idle processor times. Whenever tasks with variable execution pattern are present in the system, there are certain idle periods that are very hard or impossible to predict off-line. These idle times can be efficiently used only by run-time speed scheduling methods. These run-time methods have to detect and maybe predict the time which remains unused by every instance (or *slack*, as introduced in Section 4.3.2). Furthermore, the slack has to be distributed to future task instances, allowing each of them to run at a lower speed. Once an instance received its slack, it can dispose of it according to various task level speed scheduling methods. Such run-time speed scheduling methods, or *slack distribution* strategies, are orthogonal with both the off-line methods and task level speed scheduling approaches. All these methods may be used in conjunction, for achieving a more efficient system from the energy point of view.

On-line speed scheduling was already addressed to a certain extent by several publications. In [SC99] a task instance is run at a lower speed only if is the only one running and has enough time until a new task arrives. In all other situations tasks are executed at the speed dictated by the off-line analysis. In [LS00b] tasks are stretched to their WCET at run-time, independent of other tasks, using several internal checking/re-scheduling points. The algorithm presented in [LK99] uses only two voltage levels. The slack produced by finishing a task early is entirely used to run the processor at the low voltage. As soon as this slack is consumed, the task switches to the high voltage. Our scheduling method is perhaps most resemblant to the optimal scheduling method OPASTS presented in [HPS98]. Yet, OPASTS performs analysis over task hyper-periods, which may lead to working on a huge number of task instances for certain task sets.

In this section we describe a slack distribution strategy built on top of rate-monotonic (RM) scheduling. Our method is designed to work on processors with an arbitrary number of speeds. It has a low computational complexity, independent of the characteristics of the task sets. Briefly, in our strategy, an early finishing task may pass on its unused processor time to any of the tasks executing next. But this slack cannot be used by any task at any time since deadlines have to be met. We solve this by considering several levels of slacks, with different priorities, as in the slack stealing algorithm [LRT92].

Next, we present the on-line slack distribution strategy in detail (Section 5.6.1) and prove that our strategy keeps the worst case response time guaranteed by RM-scheduling (Section 5.6.2). We continue by describing a few experiments showing the effectivity of our method (Section 5.6.3) and, finally, present our conclusions in Section 5.6.4.

5.6.1 The Slack Distribution Strategy

Our on-line slack distribution strategy makes use of several levels of slack. For a task set $\{\tau_i\}_{1 \leq i \leq N}$ that exhibits M different priorities, we use M levels of real-time slack $\{S_j\}_{1 \leq j \leq M}$. Without great loss of generality consider that the tasks have different priorities, or $M = N$. We also consider that the task set and slack levels are already ordered by priority, where level 1 corresponds to the highest priority. The slack in each level is a cumulative value, the sum of the unused processor times remaining from the tasks with higher priority. Initially, all level slacks S_j are set to 0. At run-time, the slack levels are managed as follows:

- Whenever an instance k of a task τ_i with priority i **starts executing**, it can use an arbitrary part $\Delta\mathcal{C}_i^k$ of the slack available at level i, S_i. So the allowed execution time for instance k of task τ_i will be:

$$A_i^k = \mathcal{C}_i + \Delta\mathcal{C}_i^k \tag{5.27}$$

  where $\mathcal{C}_i$ is task τ_i worst case execution time for the maximum (reference) processor speed, equal to WCE_i/f_{ref}. The remaining slack from level i cannot be used again on the same level. Therefore the slack level i is reset to 0. We can also see this as a degradation of the slack from level i into level $i+1$ slack. To summarize, each level of slack will be updated according to:

$$S'_j = \begin{cases} 0, & j \leq i \\ S_j - \Delta\mathcal{C}_i^k, & j > i \end{cases} \tag{5.28}$$

- Whenever a task instance **finishes its execution**, it will generate some slack if it finishes before its allowed time. If X_i^k is the actual execution time of instance k of task τ_i, the generated slack is:

$$\Delta A_i^k = A_i^k - X_i^k \tag{5.29}$$

  This slack can be used by the lower priority tasks. In this case, the slack levels are updated according to:

$$S''_j = \begin{cases} S_j, & j \leq i \\ S_j + \Delta A_i^k, & j > i \end{cases} \tag{5.30}$$

- **idle processor times** are subtracted for all slacks. This ensures that the critical instance from the classic RM analysis remains the same.

The computational complexity required by the on-line method is, thus, linearly dependent to the number of slack levels: $O(M)$.

Note that task instances can only use slack generated from higher priority tasks and produce lower priority slack. We call this slack degradation. Whenever the lowest priority task starts executing, all slack levels are reset. Note also that not necessarily all slack at one level is used by a single task. Various strategies can be employed, but we mention here only the two we used in our experiments:

- **Greedy**: the task gets all the slack available for its level:

$$\Delta\mathcal{C}_i^k = S_i \tag{5.31}$$

- **Mean proportional**: we consider the mean execution time $\overline{X_i}$ for each task instances waiting to execute (in the ready queue). The slack is proportionally distributed according to these:

$$\Delta\mathcal{C}_i^k = S_i \frac{\overline{X_i}}{\sum_{\tau_j \in \text{ReadyQ}} \overline{X_j}} \tag{5.32}$$

5.6.2 Worst Case Response Time Analysis

The strategy of managing the slack just described, allows us to keep the critical instance response time for all tasks, as we prove next. The response time $R_i(t)$ for task τ_i is computed as:

$$R_i = A_i + I_i(t) \tag{5.33}$$

where A_i is its allowed execution time, as before, and $I_i(t)$ is the interference from the other tasks. From the managing strategy given before, the cumulated slack on each level i, at a certain time t is of the form:

$$S_i(t) = S_{i-1} - \sum_k \Delta\mathcal{C}_{i-1}^k + \sum_k \Delta A_{i-1}^k, \quad k = \left\lceil \frac{t}{T_{i-1}} \right\rceil \tag{5.34}$$

More informally, the slack of level i is composed of all slack from level $i-1$, less the slack used by the instances of tasks with priority $i-1$ but plus all the slack generated by these. The number of instances executed in the current hyper-period, k, is determined by the task period. Note that S_1 is always zero. Eliminating the iteration from the previous formula:

$$S_i(t) = \sum_{1 \le j < i} \left(\sum_k \Delta A_j^k - \sum_k \Delta\mathcal{C}_j^k \right), \quad k = \left\lceil \frac{t}{T_j} \right\rceil \tag{5.35}$$

The task with the highest priority will never receive slack, therefore $\Delta\mathcal{C}_1^k = 0$. The interference from the high priority tasks, $I_i(t)$ is the time used to execute

all arrived instances of these high priority tasks:

$$I_i(t) = \sum_{1 \le j < i} \sum_{k} X_j^k, \quad k = \left\lceil \frac{t}{T_j} \right\rceil \tag{5.36}$$

With the notations from the Section 5.6.1, we write the relation between the instance k execution time, X_j^k, its allowed time A_j^k, its used slack $\Delta\mathcal{C}_j^k$ and its produced slack, ΔA_j^k:

$$X_j^k = A_j^k - \Delta A_j^k = \mathcal{C}_j + \Delta\mathcal{C}_j^k - \Delta A_j^k, \quad k = \left\lceil \frac{t}{T_j} \right\rceil \tag{5.37}$$

Introducing this in 5.36:

$$I_i(t) = \sum_{1 \le j < i} \sum_{k} (\mathcal{C}_j + \Delta\mathcal{C}_j^k - \Delta A_j^k), \quad k = \left\lceil \frac{t}{T_j} \right\rceil \tag{5.38}$$

The last two terms in the sum are actually giving the slack of level i, as in 5.35, so we can re-write 5.38 as:

$$I_i(t) = \sum_{1 \le j < i} k\mathcal{C}_j - S_i(t), \quad k = \left\lceil \frac{t}{T_j} \right\rceil \tag{5.39}$$

Note that the maximal response time for a task is obtained when it uses all the slack available at its level:

$$R_i(t) = \mathcal{C}_i + I_i(t) + S_i(t) \tag{5.40}$$

From the last two equations:

$$R_i(t) = \mathcal{C}_i + \sum_{1 \le j < i} \left\lceil \frac{t}{T_j} \right\rceil \mathcal{C}_j \tag{5.41}$$

which is exactly the response time as defined by the RM analysis, obtained when all tasks execute their worst case [LL73, BW01]. Thus, if the RM analysis decides that a task set is schedulable, the conclusion remains valid when using our on-line slack distribution policy.

5.6.3 Experimental Results

To evaluate our slack distribution strategy from the energy consumption point of view, we compared it to several other methods or possible bounds:

- **100%**: No speed scheduling is performed, all tasks run as fast as possible.

- **Upper Bound**: Is the theoretical upper bound in energy reduction. This is obtained in a post-execution analysis, by considering that the tasks are uniformly stretched up to maximum processor utilization (see the *Proportional Stretch Approach*, Section 5.2). This limit is hardly achievable in practice, since the actual execution patterns for all task instances are never available beforehand. Moreover, this optimum obtained by uniformly stretching all instances usually violates some deadlines, being therefore impractical.

- **Off-line+1stretch**: This method is composed of the off-line RM-MRS step and a very simple run-time speed scheduling, originally described in [SC99]. Namely, whenever a job is running alone on the processor, it is allowed to use all the time until the next arrival of any job.

- **All**: Our run-time speed scheduling strategy using the *mean proportional* slack distribution method, and the *Stochastic Scheduling* task level strategy (see Section 4.3.1), also augmented with the previous *Off-line+1stretch* method. This is the most complete low energy scheduling approach, combining off-line, run-time, task level and task set level scheduling strategies.

- **Ideal**: Same as *All*, except assuming exact knowledge about each job execution pattern. This means that every time a task arrives, its exact execution pattern becomes known. Thus, we can directly set an ideal speed for that job.

The virtual processor used for these experiments has 14 voltage levels, with clock frequencies equally distributed between f=100MHz and 11MHz. A power-down mode is also available, in which the processor consumes 5% of the highest frequency average energy. We assumed that the NOP instruction consumes only 20% of the average power at the maximum speed, as in [SC99]. For the first scheme, *100%*, we assume that whenever the processor is idle, it executes NOPs, while for the rest of the schemes, we assume that the processor goes into the available low power mode.

First, we took two real-life hard-RT applications and simulated them on the virtual processor using the scheduling modes given above. The first application, slightly adapted from [KRH+96], is a computerized numerical control machine (CNC controller) and has the task set characteristics given in Table 5.5. The second application, adapted from [LVM91], is a generic avionics platform (GAP) and has the task set characteristics given in Table 5.6. For both task sets we assumed tasks with probabilistic execution patterns as follows. We considered that the number of clock cycles varies between a best case (BCE) and a worst case (WCE) according to a normal distribution. All distributions have the mean $\mu = (\text{BCE} + \text{WCE})/2$ and the standard deviation

$\sigma = (\text{WCE} - \text{BCE})/6$. Keeping the WCE given in the initial specification, we varied the BCE such that the BCE/WCE ratio changed from 0.1 to 0.9. A small ratio means that the task execution pattern can vary a lot, while a close to 1 ratio means that the task has almost fixed execution pattern. For each ratio we applied the scheduling methods given above and estimated the energy consumption via simulation. The results are depicted in Figure 5.9 for CNC and in Figure 5.10 for GAP.

Table 5.5: Task set characteristics for the CNC controller

# (priority)	in [KRH+96] appear as	Task Characteristics		
		WCE@$f_{max}(\mu s)$	T (μs)	D (μs)
1	τ_{smpl}	35	2400	2400
2	τ_{calv}	40	2400	2400
3	τ_{xref}	165	2400	2400
4	τ_{yref}	165	2400	2400
5	τ_{xctrl}	570	9600	4000
6	τ_{yctrl}	570	7800	4000
7	τ_{dist}	180	4800	4800
8	τ_{stts}	720	4800	4800

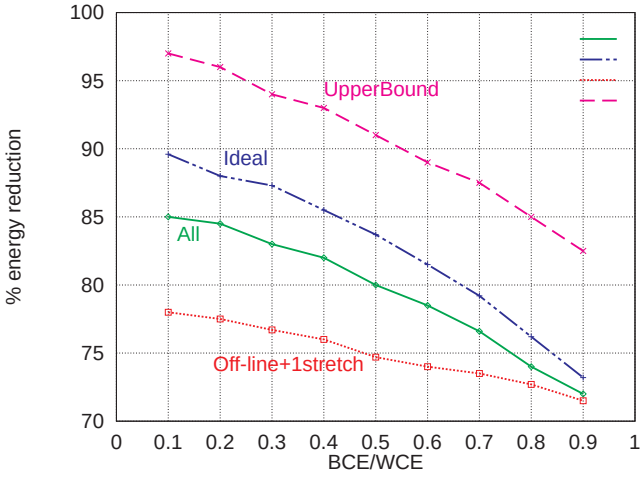


Figure 5.9: The energy reduction off the *100%* execution mode for the CNC controller.

We also tested our *All* scheduling policy on randomly generated task sets of 50 and 100 tasks. The task sets were generated as follows. For each set, the

Table 5.6: Task set characteristics for GAP

# (priority)	in [LVM91] appear as	Task Characteristics	
		WCE@$f_{max}(\mu s)$	T = D (μs)
1	Radar_Tracking_Filter	200	2500
2	RWR_Contact_Mgmt	500	2500
3	Data_Bus_Poll_Device	100	4000
4	Weapon_Aiming	300	5000
5	Radar_Target_Update	500	5000
6	Nav_Update	800	5900
7	Display_Graphic	900	8000
8	Display_Hook_Update	200	8000
9	Tracking_Target_Upd	500	10000
10	Weapon_Release	300	20000
11	Nav_Steering_Cmds	300	20000
12	Display_Stores_Update	100	20000
13	Display_Keyset	100	20000
14	Display_Stat_Update	300	20000
15	BET_E_Status_Update	100	100000
16	Nav_Status	100	100000
	Timer_Interrupt	not included (implicit)	
	Weapon_Protocol	not included (aperiodic)	
Total Utilization = 84.05			

task periods (and deadlines) were selected using a uniform distribution in 100 ... 5000 and 100 ... 10000 respectively. The worst case execution times were then randomly generated such that the task set would yield approximately 0.67 processor utilization, for the fastest clock. The average utilization after off-line RM-MRS turned out to be 0.92 for the sets of 50 tasks, and 0.85 for the sets of 100 tasks. Using the same processor type as in the previous experiment, we simulated the run-time behavior of the several scheduling methods. We again used post-simulation data to obtain the upper bound, as in the previous experiment. The values depicted in Figure 5.11 are averages over one hundred sets of tasks.

As results from these experiments, for tasks with probabilistic execution pattern, an on-line slack distribution strategy yields a dramatical decrease in energy consumption. Simpler on-line scheduling techniques, such as *Off-line+1stretch* [SC99], are improved significantly by adding our slack distribution policy. Task level scheduling strategies may additionally contribute to this result, yet with a marginal 3-4%. In fact, any further decrease in energy consumption requires great efforts, since we are approaching the practical

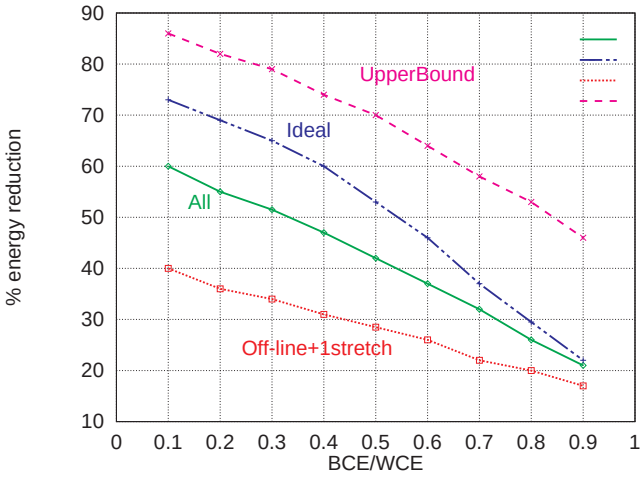


Figure 5.10: The energy reduction off the *100%* execution mode for the GAP application.

bound. An interesting property of the *All* policy seems to be the fidelity it follows the *UpperBound*. *All* is consistently 12%, 25%, 20% and 25% under the *UpperBound* for CNC, GAP, 50 task sets, and 100 task sets, respectively. This means that our scheduling strategy uses the variation of BCE/WCE ideally, performing with the same efficiency independent of the task execution pattern characteristics.

5.6.4 Conclusions

In this section we presented and analyzed a scheduling policy for hard real-time tasks running on a variable voltage supply processor, with the final purpose of reducing the energy consumption. The policy is designed for sets of tasks with fixed priorities assigned in a rate/deadline monotonic manner.

It consists of both off-line and on-line scheduling decisions, taken both at task and task set levels. The off-line decisions use exact timing analysis to derive off-line speed scaling factors for each task. The on-line policy distributes available processor time on priority basis, using slack levels and statistics. The section also contains a proof that our scheduling policy meets all deadlines. Our method can be fully implemented in a RTOS[2], without appealing to special compilers or changing the software. Yet, combined with the aforementioned methods, our approach may yield even greater energy reductions. The

[2]RTOS: Real-Time Operating System

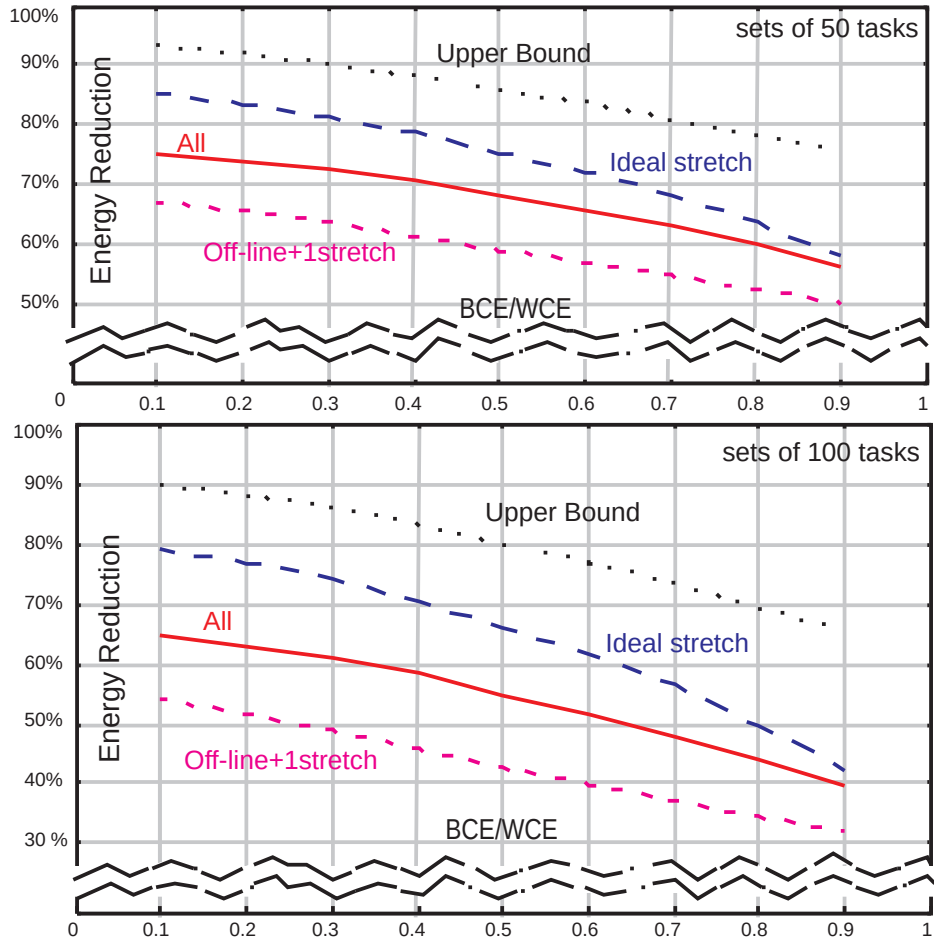


Figure 5.11: The average energy reduction off the *100%* execution mode for a hundred sets of (*above*) 50 tasks and (*below*) 100 tasks.

experimental results show that our policy can be successfully used to reduce the energy consumption in hard real-time systems.

5.7 Uncertainty-Based Scheduling

The scheduling technique described in this section is specifically designed for tasks with probabilistic execution time. Scheduling hard real-time tasks means guaranteeing the deadlines, even in the worst case. A big discrepancy between the average and the worst case execution patterns implies that the task usually ends up being scheduled at a much higher speed than it is actually required. Narrowing down this discrepancy as soon as possible during run-time re-scheduling means achieving a close to optimal speed faster. This difference between the worst case and average execution pattern reflects the uncertainty about the actual execution time of a task. Hence, the name of our method: *Uncertainty-Based Scheduling* (UBS). Our UBS technique can be classified as an off-line sequencing method, that improves the run-time decisions regarding the required processor speeds. Thus, our approach implies the use of a run-time speed scheduler, which adjusts the processor speed whenever a task finishes or in specific preemption points determined off-line by UBS.

In the following, we describe UBS for two specific cases. The first is the very simple case of scheduling a set of independent tasks before a given deadline. The second focuses on improving EDF with UBS for sets of tasks with different deadlines.

5.7.1 UBS for Tasks with Unique Deadline

The simplest possible scheduling situation is that of independent tasks with a unique common deadline. For tasks with unique execution pattern, it is optimal to run all tasks at the same speed, such that the set finishes exactly at the deadline (Section 4.1). For tasks with probabilistic execution time, a run-time scheduler would improve the energy efficiency. Once a task finishes execution, the run-time scheduler would recompute the optimal speed using the remaining tasks WCE and the remaining available time. Formally, the optimal speed required after executing a number of k tasks would be computed as:

$$s(\tau_1, \ldots, \tau_k) = \frac{1}{f_{ref}} \frac{\sum_{i=k+1}^{n} \mathrm{WCE}_i}{A - \sum_{j=1}^{k} X_j} \tag{5.42}$$

where A is the common deadline and X_j are the actual number of clock cycles executed by the already finished k tasks. Note that the speeds depend on the already scheduled tasks. Therefore, the order in which the tasks in the set are executed has direct influence on the speeds and consequently, on the energy consumption.

Definition 5.9. Given a task set $\{\tau_i\}_{1 \le i \le N}$, an **execution order** $O =< o_1, o_2, \ldots, o_N >$ is a permutation of $< 1, 2, \ldots, N >$ which specifies the

exact sequence of task dispatching: τ_{o_1} executes first, then τ_{o_2}, and so on, ending with τ_{o_N}.

From now on, we will denote the actual number of clock cycles executed by τ_{o_i} by x_i. Remember that the energy consumption during a single clock cycle depends quadratically on speed, as shown in Section 3.3.2: $e(s) = \mathcal{K}s^2$. Remember also that given a task executes at a constant speed s for X clock cycles, its energy consumption is: $E = Xe(s)$. We can now write the expression of the energy consumption of a single iteration, for a given order $O =< o_1, o_2, \ldots, o_n >$:

$$\begin{aligned} E_O &= E(< o_1, \ldots, o_N >) = \\ &= E(< o_1 >) + E(o_2| < o_1 >) + \ldots + E(o_N| < o_1, \ldots, o_{N-1} >) = \\ &= x_1 e(s) + \cdots + x_N e(s(\tau_{o_1}, \ldots, \tau o_{N-1})) \\ &= \mathcal{K}\left(x_1 s^2 + \cdots x_N s(\tau_{o_1}, \ldots, \tau o_{N-1})^2\right) \end{aligned} \tag{5.43}$$

We denoted by $E(o_k| < o_1, \ldots, o_{k-1} >)$ the energy consumed by an instance of task τ_{o_k} given it executes after tasks $\tau_{o_1} \ldots \tau_{o_{k-1}}$, in this order. Also, the initial speed, computed before any task starts executing, was denoted by s.

The above expression refers to a single instance of the task set. To examine the energy effects over a longer time we have to compute the expected value of the energy consumption. The expected energy consumption for a given order $O =< o_1, o_2, \ldots, o_n >$ can be computed as:

$$\overline{E_O} = \int \cdots \int_O E(< o_1, \ldots, o_N >)\eta(x_1) \ldots \eta(x_N)\mathrm{d}x_1 \ldots \mathrm{d}x_N \tag{5.44}$$

Example 5.3 (Order and Energy of a Three Tasks Set):
Consider a set of three tasks with a uniform distribution as execution pattern, having the following associated BCE and WCE: τ_1:(12,20), τ_2:(10,30), τ_3:(24,40). We examine here two execution orders: $< 1, 3, 2 >$ and $< 2, 3, 1 >$. Assuming we use the above described run-time speed scheduling strategy, we can compute the expected energy consumptions associated with these orders (see Equation 5.44). For example, for order $< 2, 3, 1 >$ the expected energy is computed as the following triple integral:

$$\int_{10}^{30}\int_{24}^{40}\int_{12}^{20} \frac{1}{128}\left[\frac{81x_2}{2000} + \frac{245x_3}{(100-x_2)^2} + \frac{45x_1}{(100-x_2-x_3)^2}\right]\mathrm{d}x_2\mathrm{d}x_3\mathrm{d}x_1$$

Using numerical integration, we obtained the expected energy values for the two different orders mentioned above. The results are listed in Table 5.7. Note that the energy is a function of three random variables. We wanted to see the difference between using the function of expected values ($E[\overline{X}]$) instead of the expected value of the function ($\overline{E[X]}$). The $E[\overline{X}]$ approximation

of the expected energy definitely computes faster since the integrals disappear, but the accuracy of the results might be a problem in some cases. For the two orders presented below, the differences are a mere 0.6% and 1.3%, respectively. We also computed the energy for two additional scenarios. Case 3 is the ideal case, when we know that all tasks will execute always at their mean execution time ($(\text{WCE} - \text{BCE})/2$). In this case, the optimal speed is 0.68 of the nominal speed. In case 4, no run-time speed scheduling is performed and therefore the speed must always accommodate WCE. Finally, note that the execution order has a significant influence on the energy consumption. The order $< 2, 3, 1 >$ is preferred over $< 1, 3, 2 >$, since the first yields only 88% of the second order energy consumption.

Table 5.7: Expected energy consumption for four different scenarios

Case	Execution Type	$\overline{E[X]}$	$E[\overline{X}]$	$\overline{E[X]}$ % Ideal
1	Order $< 1, 3, 2 >$	42.094	41.839	134%
2	Order $< 2, 3, 1 >$	37.482	36.978	119%
3	Ideal: always mean	31.443 (speed 0.68)		100%
4	Off-line WCE speed	55.080 (speed 0.9)		175%

Problem Definition

With the specified run-time speed scheduling strategy and the given task set model, the problem we are facing is finding the execution order for which the expected energy consumption is minimized. A simplified problem is the case when each task generates only instances with the same execution time. The run-time speed scheduler still considers the tasks WCE for computing the next optimal speed. For clarity, we denote next the worst case execution time WCE/f_{ref} by c and the instance execution time by x. Furthermore, we will assume that the energy depends quadratically on speed and the cycle energy for the reference frequency is 1. We can now present an easy to read formulation of the simplified ordering problem.

Definition 5.10 (The Simplified UBS Problem). Given:

- A set of n tasks as pairs of (real) numbers:

$$(x_1, c_1), (x_2, c_2), \ldots, (x_i, c_i), \ldots, (x_n, c_n) \quad c_i > x_i > 0$$

- A time interval A such that:

$$\sum_{i=1}^{n} c_i \leq A$$

The tasks are to be executed on a processor in a certain order defined as a permutation $\pi(1, 2, \ldots, n)$ such that the following measure (energy consumption) is minimized:

$$\begin{aligned} E &= x_{\pi_1} + x_{\pi_2}\left(\frac{\sum_{k=1}^{n} c_{\pi_k} - c_{\pi_1}}{A - x_{\pi_1}}\right)^2 + \cdots \\ &\cdots + x_{\pi_i}\left(\frac{\sum_{k=1}^{n} c_{\pi_k} - \sum_{j=1}^{i-1} c_{\pi_j}}{A - \sum_{j=1}^{i-1} x_{\pi_j}}\right)^2 + \cdots + x_{\pi_n}\left(\frac{c_{\pi_n}}{A - \sum_{j=1}^{n-1} x_{\pi_j}}\right)^2 \end{aligned}$$

The simplified ordering problem as presented above appears to be NP-hard (see Appendix A.2). We conclude then that the initial case, for which the instances have variable execution patterns, is at least as hard.

In Search of the Optimal Order

For the problem defined above, an exact algorithm would take a prohibitive amount of time for finding the optimal order. Thus, we need a heuristic that can provide us with a near optimal order. In this section we describe such a heuristic, together with the reason behind it.

First, it is important to realize that computing the actual energy for a given order is very expensive because of the multiple integral. Moreover, this integral is usually only solvable using numerical integration. However, the $E[\overline{X}]$ approximation for our expected energy, $\overline{E[X]}$, seems to be somewhere around 1-2%, as in Example 5.3. From now on we will use this approximation for our heuristic.

For a certain order $< 1, \ldots, k, k+1, \ldots, N >$, we can compute the energy consumption according to equation 5.43. Let the initial speed be s_0 and the last speed s_{N-1}. We also denote the speeds for the partial order $< 1, \ldots, k-1 >$ and $< 1, \ldots, k-1, k >$ by s_{k-1} and $s_{k-1,k}$ respectively. These speeds are computed according to equation 5.42. Finally the energy for this order is:

$$E = \overline{X_1} s_0^2 + \cdots + \overline{X_k} s_{k-1}^2 + \overline{X_{k+1}} s_{k-1,k}^2 + \cdots + \overline{X_N} s_{N-1}^2 \tag{5.45}$$

Swapping the places of of tasks k and $k+1$, the sum yielding the energy consumption will differ only at terms k and $k+1$. All others contain attributes of both tasks in commutative relations (sums), where order is not important. More precisely, the speed after a partial order $< 1, \ldots, k, k+1, \ldots, p >$ is exactly the same as the speed for the order $< 1, \ldots, k+1, k, \ldots, p >$. Denoting the new energy by E', we can write the difference in the two energy as follows:

$$\begin{aligned} E' - E &= \overline{X_{k+1}} s_{k-1}^2 + \overline{X_k} s_{k-1,k+1}^2 - \overline{X_k} s_{k-1}^2 - \overline{X_{k+1}} s_{k-1,k}^2 \\ &= \overline{X_{k+1}} \left(s_{k-1}^2 - s_{k-1,k}^2\right) - \overline{X_k} \left(s_{k-1}^2 - s_{k-1,k+1}^2\right) \end{aligned} \tag{5.46}$$

For the $k, k+1$ order to consume less energy than the $k+1, k$ order, we must have $E' > E$. Considering that $s_{k-1} > s_{k-1,k}$ and $s_{k-1} > s_{k-1,k+1}$, the previous condition can be written as:

$$\frac{\overline{X_k}}{s_{k-1}^2 - s_{k-1,k}^2} < \frac{\overline{X_{k+1}}}{s_{k-1}^2 - s_{k-1,k+1}^2} \tag{5.47}$$

Note that both sides of this inequality depend only on the set of previous tasks $1 \ldots k-1$, and the current task k (or $k+1$). Also the order of the previous tasks is not important. Using these observations we can build a constructive algorithm, which incrementally adds tasks to a partial order o according to the following priority function:

$$p_{ubs}(o, \tau_k) = \frac{\overline{X_k}}{s_o^2 - s_{o,k}^2} \tag{5.48}$$

where s_o is the speed after the given partial order and $s_{o,k}$ is the speed after adding task τ_k at the end of the partial order o. Note that a low value of p_{ubs} means high priority.

There are two important properties of the p_{ubs} priority function:

- **favors short tasks** (smaller $\overline{X}$ yield lower values). This makes sense, considering we want to achieve a lower speed faster.
- **favors lower speeds** (smaller $s_{o,k}$ yield lower values). This is also desirable since we want to achieve as low speeds as possible early. Note that lower speed means also a large difference between the mean execution time $\overline{X}$ and the worst case WCE. In other words, tasks exhibiting big uncertainties are favored, hence the name for our method: *uncertainty based* scheduling.

The heuristic for deciding the execution order starts with an empty set of scheduled tasks, and the initial speed computed to accommodate the worst case for all tasks (as in case 4 in Example 5.3). The priorities for the unscheduled tasks are the computed using p_{ubs} and the one with the highest priority (lowest value) scheduled. The order is updated and the procedure re-iterates until all tasks are scheduled. As described here, this algorithm exhibits a worst case time complexity of the order $O(N^2)$, with N as the number of tasks. Next, we examine the efficiency of this approach in finding good solutions.

Example 5.4 (Simple UBS on Intel 80200 XScale):
We examined the performance of our scheduling heuristic on the real platform described in Appendix B. For this we built a set of five tasks (presented in Table 5.8), each of them performing simple arithmetic operations inside a

loop. Each task has a uniform distribution as execution pattern, from a BCE to a WCE given in number of loop iterations. Measured on the system, each loop corresponds to approximately 3550 MCLKs, or memory clock cycles. Since in our system the MCLK is 100MHz, each loop iteration is around 35.5μs. In the worst case, the task set at the highest processor speed (733MHz) would take 1490731 MCLKs or almost 15ms. In the absence of any dedicated real-time clock available on the board, we use the MCLK counters to get information about actual task execution times. Since the MCLK counter cannot be trusted during a frequency switch, the drawback with this solution is the inaccuracy introduced by speed switching in timing. However, the total time spent in switching is around $5 \times 30\mu$s, which is 0.15ms, or hundred times smaller than the total execution time of the task set in the worst case at the highest speed. In this light, the influence of speed switching on timing can be considered negligible. We set the experiment such that the period and deadline for the task set is 16ms, to accommodate also the timing delay introduced by speed and task switching. This means that for the worst case less than 7% slack is available in the system.

Table 5.8: The five tasks set scheduled on XScale

Task	BCE		WCE	
#	loops	MCLKs	loops	MCLKs
1	10	35504	100	355010
2	20	70992	50	177460
3	15	53247	80	283928
4	30	106482	100	354921
5	80	283928	90	319417
+	155	550153	420	1490736
WCE time @ 733MHz is approx. 15ms				

We considered the following scheduling strategies:

- **No Scaling**: All tasks execute at the maximum speed, 733MHz.
- **Best Order**: The speed is computed at run-time before each task starts executing. Each task executes at a single speed, out of the five available in our test system. This speed is the minimum speed that guarantees the deadline even in the worst case. The order computed by our UBS heuristic, in this case yielding the order: 1,3,4,2,5.
- **Worst Order**: Same run-time scheduling strategy as before, except the order is reversed: 5,2,4,3,1.

- **Variable Random Order**: Same run-time scheduling strategy, except the order is randomly chosen for each instance of the task set. In the long run, this strategy should yield neither the worst nor the best result from the energy point of view. In fact, it should be very close to the mean energy between a certain order and its reverse order. Considering that the random re-ordering of tasks requires more energy than a fixed order, the energy consumption for this policy will be slightly over the actual mean.

Note that the latest three strategies share the run-time speed computation part, and differ only in task ordering. Furthermore, for all the above strategies the processor goes to a low power state (IDLE) once all jobs complete.

We run the task set for all these scheduling strategies, and average the power consumption over a large number (thousands) of task set instances. The average shapes of the power curve are depicted in Figure 5.12. Table 5.9 presents the numerical results. Note that the different orders yield different average energy consumptions. Moreover, as expected, the *variable random order* energy is very close to the median between the best and reverse best (worst) order. Finally, the *best order* yields an energy consumption that is only 80% of the *no scaling* case, and 92% of the median, *variable random order* energy. This experiment shows that our UBS ordering performs better than chance (median) even on a real system.

Table 5.9: Average energy consumptions for the various scheduling methods

#	Method	Energy/Period (mJ)	As % of 1	As % of 3
1	**No Scaling**	88.95	100.00	115.10
2	**Worst Order**	82.35	92.58	106.70
3	**Variable Order**	77.28	86.88	100.00
4	**Best Order**	71.15	80.00	92.07

UBS Heuristic vs. Optimal Order

We compare our fast ordering heuristic based on the p_{ubs} priority function with optimal orders, obtained using full search.

For this, we randomly generated sets of tasks with the following characteristics. The WCE for each tasks are chosen from a uniform distribution such that the total WCE for each set is constant, 10000. Next, the BCE for each task is chosen from a uniform distribution to fall somewhere between 0 and $0.8 \times \text{WCE}$. The execution pattern for each task is set to a normal distribution, with mean $\mu = (\text{WCE} + \text{BCE})/2$ and standard deviation $\sigma = (\text{WCE} - \text{BCE})/6$.

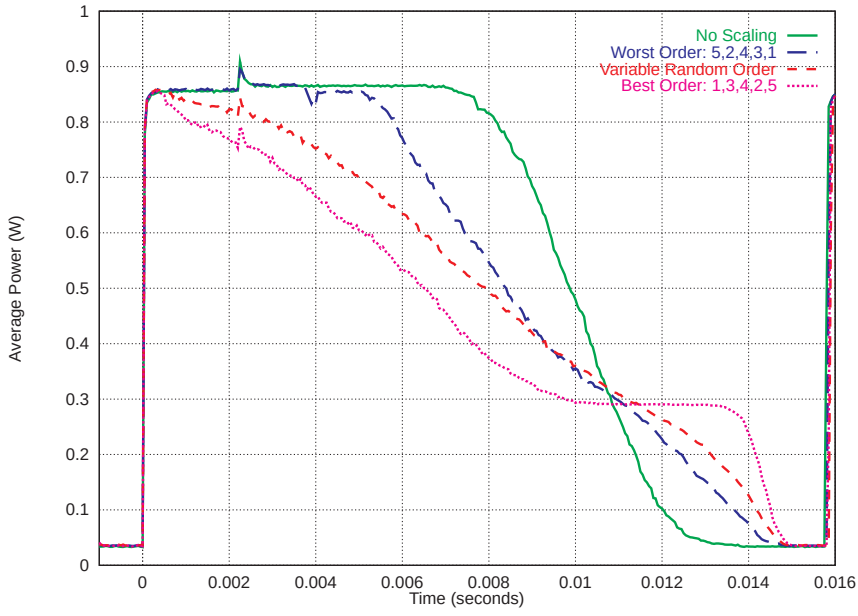


Figure 5.12: Oscilloscope trace for the power consumption during several scheduling methods for the given set of tasks with probabilistic execution pattern. The curves are averages over thousands of instances.

The deadline for each task set is chosen to be the sum of all task WCE (10000). Therefore, in the worst case, the processor would have to run at the maximum speed to meet this deadline. For each task set we obtained the order according to our p_{ubs} function and computed its energy using numerical integration from Equation 5.44. We carried this out for three hundred task sets of a given size, and selected the minimum, maximum and computed the average value over all. We obtained this triplet for sets of three, four, five, and six tasks, and plot them in Figure 5.13 as *UBS*.

For each task set we also performed full search to find the optimal order. For every possible order of tasks, we computed the energy consumption using numerical integration from Equation 5.44. From all orders we kept the one yielding the lowest energy consumption. As before, we noted the minimum, maximum and averages of all these energy values, and plot them in Figure 5.13 as *FullSearch*. All values are relative to non-scaling average case energy for each specific graph.

Note that the maximum set size we report results for is six, since the run times for the experiments became prohibitively long. For example, after almost four days of run time, using full search we were unable to examine

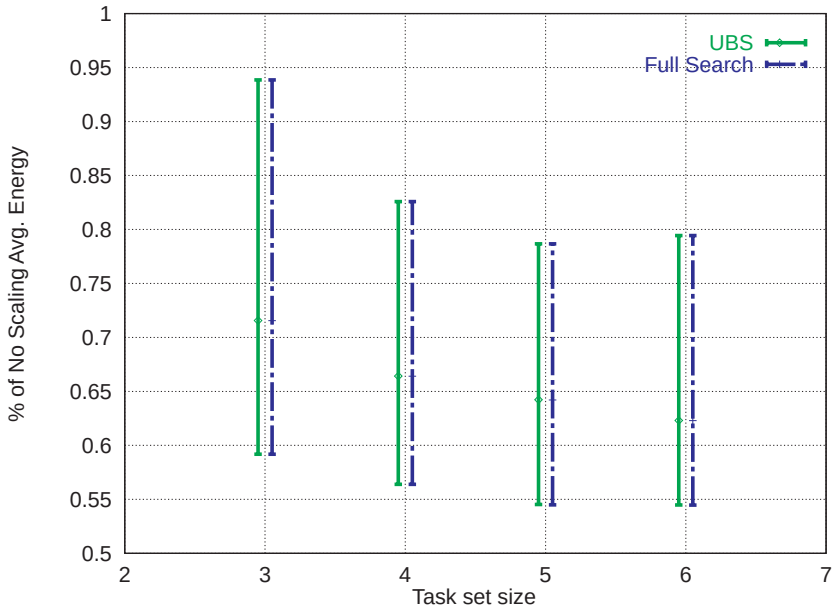


Figure 5.13: Average, minimum, and maximum "lowest energy" values found by UBS and Full Search for different task set sizes.

even two hundred sets of seven tasks (on an AMD Athlon XP1600+, 256MB RAM, Java implementation under SuSE Linux 7.3). However, the same task finished in six hours for sets of six tasks. There are several reasons for this increase in execution time. The time for computing the numerical integral grows exponentially: we used five integration points for each dimension, thus, for each integral, 5^N values need to be computed, where N is the number of tasks. Then, the number of possible orders for a set of N tasks grows in a factorial manner, $N!$. A rough estimate says that full search for a set of seven tasks requires approximately $5 \times 7 = 35$ times more execution time than a full search for a set of six tasks. On the other hand, UBS keeps a low complexity $O(N^2)$, being able to find solutions fast, even for large task sets.

From Figure 5.13 it results that UBS is very efficient in finding optimal order, at least for small task sets. In fact, the difference between the energy found by full search and UBS is less than 1%. Also compared to the non-scaling case, when the processor always runs at the maximum speed and then goes to idle at 0 power, UBS can in average reduce the power consumption around 28-37% without any deadline extension.

5.7.2 Extending UBS for EDF

The UBS strategy can be extended to more complex cases, as an improvement of classic real-time scheduling techniques. Note that, from the timing point of view, tasks with the same priority may execute in any order, without affecting the real-time behavior of the task set. This means that one can choose the most energy-efficient order for these tasks and still meet all deadlines. Whenever there are no tasks with the same priority, one may slightly alter the task set without affecting the timing. We present next such a method for sets of tasks with unique period and different deadlines.

In the previous section, we presented the UBS ordering heuristic for sets of tasks with unique period and deadline. Now, we will examine how UBS may be applied to an extension of the task set model, with tasks to having different deadlines. From [SSRB98], we know that the preemptive earliest deadline first policy (EDF) finds feasible schedules even up to 100% processor utilization. Thus, we build our UBS heuristics on the EDF strategy. First, computing the run-time speeds for each task can be done using the EDF-MRS algorithm, presented in Figure 5.8. Although intended as an off-line speed selection strategy, EDF-MRS can be transformed into a run-time policy, if employed before every new job starts executing. More precisely, if an instance of task k finishes at time t, the speed for the next job is computed as (see Section 5.5.1):

$$s(\tau_1, \ldots, \tau_k) = \max_{k<j\leq N} \left\{ \frac{1}{f_{ref}} \frac{\sum_{i=k+1}^{j} \text{WCE}_i}{D_j - t} \right\} \tag{5.49}$$

Our initial UBS strategy works on tasks with the same priority from the real-time point of view. Therefore, the basic idea of the extension for EDF is to look at the intervals between two consecutive deadlines. The sub-set of tasks using any of these intervals can be ordered according to our p_{ubs} priority function. The pseudo-code for our EDF-UBS algorithm is given in Figure 5.14. At the beginning of the ordering procedure, all tasks are assumed to use the speeds computed using EDF-MRS. Starting from the latest deadline D_N, we examine the tasks required to finish between two consecutive deadlines $(D_{i-1}, D_i]$. Initially this *WorkList* contains only the tasks required to finish by D_N. The tasks in the *WorkList* are ordered according to p_{ubs} and use the maximal required speed computed by EDF-MRS. These tasks fill the $(D_{i-1}, D_i]$ interval backwards from D_i, starting from the lowest priority task according to p_{ubs}. At some point, it is possible that the whole interval is occupied and there still are tasks in the *WorkList*. All these tasks will need to be scheduled earlier than D_{i-1}, so they are moved to the next iteration *WorkList*. Moreover, it is possible that a specific task has to be split in two, in order to entirely fill the $(D_{i-1}, D_i]$ interval. In this case, the prefix is regarded as a new task, with a new WCE and a new expected execution time $\overline{X}$. The new WCE is ob-

```
edf_ubs({τ_k,...,τ_N}) {
  /* Compute task speeds with EDF_MRS */
  WorkList ← ∅;
  for i = N downto 2 do {
    t ← D_i;
    NextWorkList ← ∅;
    WorkList ← WorkList ∪ {all τ with deadline D_i};
    Order WorkList descending according to p_ubs;
    for each τ_j ∈ WorkList do {
      if(WCE_j/f_j ≤ t − D_{i−1})
        Schedule τ_j during [t − WCE_j/f_j, t);
      else {
        /* This task does not fit. Split it into prefix and postfix */
        WCE_prefix ← WCE_j − (t − D_{i−1})*f_j;
        D_prefix ← D_{i−1};
        X̄_prefix ← prefix_expected_execution;
        τ_prefix ← new task(WCE_prefix, D_prefix, X̄_prefix);
        /* Will schedule this next iteration */
        NextWorkList ← NextWorkList ∪ {τ_prefix};
        WCE_postfix ← WCE_j − WCE_prefix;
        D_postfix ← D_i;
        X̄_postfix ← postfix_expected_execution;
        τ_postfix ← new task(WCE_postfix, D_postfix, X̄_postfix);
        Schedule τ_postfix during [D_{i−1}, t);
        /* Extract tauj from the working list */
        WorkList ← WorkList \ {τ_j};
        /* Add remaining unscheduled tasks to the next iteration list */
        NextWorkList ← NextWorkList ∪ {unscheduled τ from WorkList};
        break;
      }
      /* Move to the previous inter-deadline interval */
      t ← t − WCE_j/f_j;
    }
    WorkList ← NextWorkList;
  }
}
```

Figure 5.14: The EDF-UBS algorithm for deciding an energy-efficient order of sets of tasks with different deadlines and unique period.

tained from the old WCE, less the part that can be scheduled in $(D_{i-1}, D_i]$. The expected execution time for the prefix task can be obtained from the task execution pattern distribution. For example, for a uniform distribution this is

the mean between the (old) BCE and the new WCE. Finally, the prefix task is added to the next iteration *WorkList* and can be regarded now as a task which has to finish before D_{i-1}. The postfix (the rest of the split task), is still scheduled in the current $(D_{i-1}, D_i]$ interval. The procedure moves on to scheduling tasks between D_{i-2} and D_{i-1}. Finally, the procedure ends when we reached time 0 and all tasks have been scheduled.

Note that the EDF-UBS procedure given above may introduce preemption points, otherwise unnecessary in the classic EDF approach. However, it is likely that not all task splits will lead to preemptions, since the prefix and the postfix may happen to follow each other immediately in the complete schedule. In these cases, the parts are merged to form the old task and no context switching is necessary. It is also important to notice that if the prefix tasks do not finish before the preemption point, there is no need to recompute the speeds or order the remaining tasks or task parts. Both EDF-MRS and EDF-UBS are called only when an original task finishes before its WCE.

Example 5.5 (EDF-UBS for a set of five tasks):
Consider the set of five tasks introduced in Example 5.1. Additionally, assume that tasks 3 and 5 have their execution pattern according to a uniform distribution between 0 and their WCE, while the rest of the tasks, 1,2, and 4, always finish at their WCE. We are interested in the EDF-UBS order at the moment when all the tasks are present in the system and are about to start executing.

Our EDF-UBS method starts by deciding the maximum required speeds for each task using EDF-MRS. For our set, we already computed these values in Example 5.1. Tasks 4 and 5 run at 364MHz while tasks 1,2, and 3 run at 667MHz. We start now from the longest deadline and look back to the previous deadline, obtaining the interval $[14, 20)$. During this interval, the only task executing is task 5, at 364MHz. Note that task 5 cannot finish during this interval since it would require 8.24ms instead of $20 - 14 = 6$ms. Here, we have the situation of splitting a task into a prefix and a postfix. Task 5 postfix (5_+) is composed of 2.18×10^6 cycles in worst case, half of that in the expected case (because of the uniform distribution), and has to finish before 20ms. Task 5 prefix (5_-) is composed of the rest of 0.82×10^6 cycles in the worst case, half of that in the expected case, and has to finish before 14ms. In other words, task 5 prefix cannot be executed in $[14, 20)$, and has been moved for scheduling somewhere before that. Next, we have to examine the interval $[9, 14)$. The two tasks to be scheduled here are task 4 and task 5 prefix. Using our p_{ubs} priority function, it turns out that task 5 prefix has to execute first during $[9, 11.25)$ and task 4 last, during $[11.25, 14)$. Fortunately in this case, all tasks fit in the working interval, so no task splitting occurs. The procedure continues then in a similar manner for intervals $[8, 9)$, $[4, 8)$, and $[0, 4)$, as detailed by Table 5.10. Note that, after completing the whole

procedure, task 2 prefix and task 2 postfix are exactly one after the other, so they can be merged together into task 2 again.

Table 5.10: Applying EDF-UBS for a five tasks set given in Example 5.1.

Working Interval	Working Tasks		UBS Order	Mentions
	Original	Leftovers		
$[14, 20)$	5	-	5_+	5 splits into: $5_-(\text{WCE} = 0.82, \overline{X} = 0.41)$ $5_+(\text{WCE} = 2.18, \overline{X} = 1.09)$
$[9, 14)$	4	5_-	5_-:$[9, 11.25)$,**4**:$[11.25, 14)$	
$[8, 9)$	3	-	3_+	3 splits into: $3_-(\text{WCE} = 1.33, \overline{X} = 0.66)$ $3_+(\text{WCE} = 0.67, \overline{X} = 0.33)$
$[4, 8)$	2	3_-	2_+	2 splits into: $2_-(\text{WCE} = 0.33, \overline{X} = 0.33)$ $2_+(\text{WCE} = 2.67, \overline{X} = 2.67)$
$[0, 4)$	1	3_-; 2_-	3_-:$[0, 2)$,**1**:$[2, 3.5)$,2_-:$[3.5, 4)$	
Final order: (after merging 2_- and 2_+) 3_-:$[0, 2)$,**1**:$[2, 3.5)$,**2**:$[3.5, 8)$,3_+:$[8, 9)$,5_-:$[9, 11.25)$,**4**:$[11.25, 14)$,5_+:$[14, 20)$				

For this particular example, two tasks have been split yielding two preemption points that were not present in the classic EDF schedule. The advantage from the energy point of view is that processor speed can be lowered early, and therefore reduce the energy consumption. For example, executing 3_- before 1 and 2 may lead to finishing task 3 during its 3_- part. This means that task 3 is out of the system, and task 1 and 2 may be rescheduled to use a lower speed using EDF-MRS. At this point, EDF-MRS will consider only tasks 1, 2, 4, and 5. On the other hand, keeping the classic order 1,2,3,4,5, results in inevitably executing 1 and 2 at a high speed.

To realize the important difference between using only run-time EDF-MRS and additionally using EDF-UBS, let us consider the best case behavior, when both tasks 3 and 5 finish as soon as they start. For EDF-UBS, at moment 0, the above scheduling takes place and task 3_- starts executing, finishing at once. At this moment we recompute the speeds for tasks 1,2,4, and 5 using EDF-MRS. This results in the required speed of 500MHz for tasks 1 and 2, and 333MHz for tasks 4 and 5. Next we may apply EDF-UBS again, on the task set 1,2,4 and 5, as depicted in Table 5.11.

Tasks 1 and 2 will now continue to execute all their cycles at 500MHz. No changes in the schedule occur until 5_- starts and immediately finishes its execution. At this point, time 8ms, the new best speed for task 4 is 167MHz. Finally the energy consumed during this time is given by the cycles executed

Table 5.11: Applying EDF-UBS for the tasks remaining after task 3 finishes as soon as it starts.

Working Interval	Working Tasks		UBS Order	Mentions
	Original	Leftovers		
$[14, 20)$	**5**	-	5_+	**5** splits into: $5_-(\text{WCE} = 1.0, \overline{X} = 0.5)$ $5_+(\text{WCE} = 2.0, \overline{X} = 1.0)$
$[8, 14)$	**4**	5_-	5_-:$[8, 11)$,**4**:$[11, 14)$	
⋮	⋮			
Final order: **1**:$(0, 2)$,**2**:$[2, 4)$,5_-:$[8, 11)$,**4**:$[11, 14)$,5_+:$[14, 20)$				

by tasks 1 and 2 at 500MHz and 4 at 167MHz (using Equation 3.8):

$$E^{best}_{edf\_ubs} = e_{ref}\left[(1+3)\times 10^6 \times 0.5^2 + 1\times 10^6 \times 0.167^2\right] = 1.028\times 10^6 e_{ref}$$

On the other hand, the run-time EDF-MRS schedule would require tasks 1 and 2 to execute at 667MHz before moving on to task 3. At time moment 6ms, task 3 starts and immediately finishes its execution. A new EDF-MRS speed selection for tasks 4 and 5 yields a speed of 285MHz for both ($(1+3)\times 10^6/(20\text{ms} - 6\text{ms})$). Task 4 executes all its cycles at 285MHz and finally task 5 starts and finishes immediately. The energy consumption for this case is (using Equation 3.8):

$$E^{best}_{edf} = e_{ref}\left[(1+3)\times 10^6 \times 0.667^2 + 1\times 10^6 \times 0.285^2\right] = 1.861\times 10^6 e_{ref}$$

The ratio between the two energy consumptions is clearly in the favor of EDF-UBS, showing that $E^{best}_{edf\_ubs}$ is only 55.24% of the run-time EDF-MRS energy. If all tasks exhibit their worst case behavior, the energy consumption of the two approaches are roughly the same.

In the ideal situation, for zero overhead in preemption or scheduling, using the EDF-UBS over the simpler run-time EDF-MRS always pays off. In practice, the more complex computations required by EDF-UBS and the preemption costs introduce an overhead in energy consumption. Depending on the task set characteristics it will not always pay off to use EDF-UBS instead of the simpler run-time EDF-MRS strategy. Although UBS can, in theory, lead only to improvements from the energy consumption point of view, the practical case requires careful evaluation, because of the scheduling overhead.

5.7.3 Conclusions

Scheduling strategies that take into account the probabilistic parameters of task execution have been proven successful for task level speed selection. In this section we have shown that knowledge about the execution pattern of tasks can be used at task group level in a similar manner. In principle the earlier one eliminates the uncertainties, the faster the optimal speed is reached. Based on this observation, we introduced the Uncertainty-Based Scheduling (UBS) method for reduced energy consumption. Our method orders tasks or tasks portions with the same real-time priority such that the average energy is minimized. Using a simple, but efficient priority function, UBS schedules first tasks that are short or yield a large decrease in speed. We have shown that for sets of tasks with unique period and deadline, UBS is energy efficient on a real system, based on the Intel80200 XScale processor. Furthermore, UBS is an energy-aware scheduling policy that can be added on top of other strategies originally designed for guaranteeing real-time behavior. As an example, we described an extension of the Earliest Deadline First EDF policy that takes advantage of UBS.

CHAPTER 6

ARCHITECTURE SELECTION AND SCHEDULING

THE SCHEDULING STRATEGIES presented in the previous two chapters assume that the system architecture is already decided. Yet, selecting the architecture and binding the tasks to processors are important design choices, that greatly influence scheduling and, ultimately, energy consumption. For this reason, the current chapter examines architecture selection and scheduling in conjunction, as integral parts of a design flow for energy-efficient systems. In particular, the chapter starts by introducing a generic design flow for low energy, involving assignment and scheduling at some point. Next, two particular cases are examined: First, fixed speed processor systems are addressed using a combined, single step approach for scheduling and task mapping. Then we describe two possible design flows intended for architectures with variable speed processors. The experimental results presented for each approach show the importance of addressing the energy issue as soon as possible in the design process, during every design decision.

6.1 An Overview of Our Design Flow

Although our work focuses on energy consumption minimization, there are several system parameters that actually have to be considered during the design process. The final design is a trade-off between fabrication cost, energy consumption, flexibility and many other metrics. In our view, only the designer can decide on the importance of each of these metrics, or choose the preferred solution, in other words. For this reason, we adopt a design flow

that involves design space exploration. Before taking a decision the designer should be able to examine several design solutions, relax constraints, enforce new ones, and re-design at will.

Figure 6.1 contains an overview of our energy-aware design flow. In our specific case, the input to the design exploration process is the task-graph describing the system functionality. The number and type of available resources, or the *resource pool*, is often given/controlled by the designer. By altering the resource pool, the designer can explore the design space in search of more suitable solutions. In real-time systems, important constraints are related to timing. In our case, the execution deadline is given as a requirement. Thus, the design process, as described here, is time and resource constrained. For

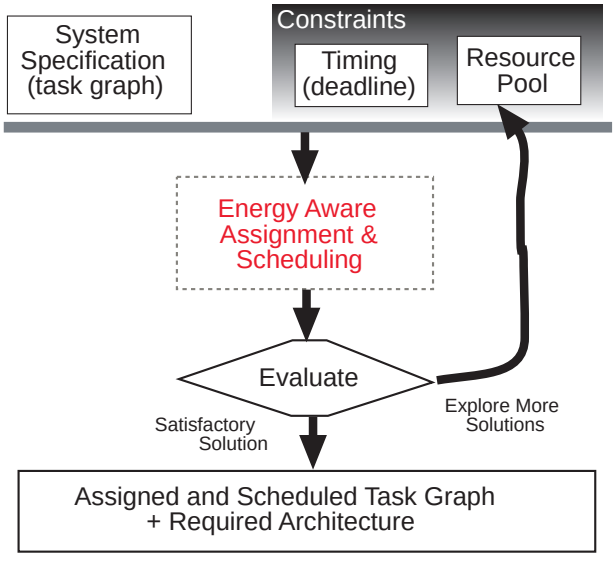


Figure 6.1: Generic design flow using design space exploration

a certain input configuration, the process should be able to assign tasks to some of the resources in the pool (deciding thus the architecture) and schedule them, such that the energy consumption is minimized. The designer must be able to examine the solution presented at this point, and decide if it is acceptable. If the solution is not good enough for some reason, the designer should be able to alter the initial constraints and reiterate the design process in the hope of finding a better solution. Note that the iterations must be fast enough to yield an interactive process, in which the designer can (un)do changes and receive feedback in due time.

The design approaches presented next address only one such iteration,

covering the dashed box titled *Energy Aware Assignment & Scheduling* from Figure 6.1.

6.2 Fixed Speed Processor Architectures

Classic system-level synthesis problems, for fixed speed processor architectures, are often solved using integer linear programming (ILP) [PP92], constraint programming (CP) [Kuc01], or various heuristics [Wol97]. The goal, in these cases, is usually the minimization of the execution time of a given task graph using a pre-defined resource pool. Assigning tasks to resources and to time moments are carried out in an integrated manner, appearing as a single step seamlessly combining scheduling and assignment. The advantage of such an integrated approach is the potentially larger set of choices during design, as opposed to a step by step process (assignment followed by scheduling). In turn, the search space may become so vast, that only a small number of possible solutions may actually be inspected. However, good heuristics may help in choosing the interesting subset of the possible solutions.

Instead of the classic scheduling problems, which usually optimize the schedule length, we are interested in minimizing the energy consumption. Moreover, the energy consumption is a non-linear function of processor power and task execution time, even for fixed speed processors. This makes our optimization problem more difficult than the classic schedule length minimization problem. In the following, we briefly present a constraint programming approach to the joint assignment-scheduling problem, under time and resource constraints. Our method, introduced in [GK99, SGK00] and detailed in [Gru00a], is based on a constraint programming approach, targeting schedule length minimization under resource constraints, described in [Kuc99].

6.2.1 Modeling the Problem with Constraints

Informally, the problem we are willing to solve is the following. Given the tasks, the dependencies between them, and a set of available resources, we want to determine the exact time moments and resources used by each task. The variables of interest for each task are thus the resource used and the exact moment of time the task executes. We assume non-preemptive execution and that each task uses only one resource. Also, the execution length for a task may vary with the resource used. Furthermore, the schedule has to meet the deadline. Finally, the goal is to minimize the energy consumption of the whole system. More formally, the above problem is described as follows.

Task Modeling

Each task i in the task graph is modeled by a triple of *variables*: $< T_i, D_i, R_i >$, where T_i is the start time for task i, D_i is the execution time required by the task on its assigned resource R_i. All these variables can take values from a *domain*. For our purpose, since the number of resources is finite and the execution for each task is fixed, these domains are finite. The domain of R_i for example may initially be the whole set of resources, but, as constraints are imposed, it may become smaller. For simplicity, we consider that resources are identified by a unique number, therefore initially:

$$dom(R_i) = \{1, \ldots, \rho\} \tag{6.1}$$

Without loss of generality we can even consider that T_i and D_i are given in clock cycles, if the processors have the same clock speed. Otherwise we can choose the greatest common divisor (gcd) of the clock cycle lengths as time unit. The initial domain of T_i stretches from a certain zero moment to the deadline, since we may schedule task i anywhere during this time:

$$dom(T_i) = \{t_0, \ldots, t_{deadline}\} \tag{6.2}$$

Further constraints resulting from dependencies will likely restrict this initial domain. Finally, the domain of D_i is given by the execution time d_{ir} (estimates or measurements) of task i on each available resource r. Note also that D_i and R_i strongly determine each other, since a certain assignment fixes the execution time, while a certain execution time might be achieved only on a few resources, if at all. Formally, we can write this as:

$$dom(D_i) = \bigcup_{r \in dom(R_i)} \{d_{ir}\}, \quad dom(R_i) = \bigcup_{d \in dom(D_i)} \{p | d_{ip} \equiv d\} \tag{6.3}$$

Modeling Dependencies

The directed edges in the task graph reflect the dependencies between various tasks. If the task graph is $\Gamma(N, V)$ we can formally include this partial order in our constraint model using the following relations:

$$\forall v_{ij} \in V \quad T_i + D_i \leq T_j \tag{6.4}$$

The above relation imposes the requirement that if there exist an edge from i to j, then task j may only start after task i completed its execution. Additionally, all tasks must finish before the deadline:

$$\forall i \in N \quad T_i + D_i \leq T_{deadline} \tag{6.5}$$

Modeling Resource Usage

One of the assumptions for our problem is that resources may handle only one task at the time. Furthermore, we assume that tasks use at most one resource at the time. Computational tasks always use one processor. Communications can also be viewed as tasks using a single resource, which is a communication channel (bus, point-to-point link). Yet, communications are slightly more special than computations, since two communicating tasks executing on the same processor may use shared memory to exchange information. In this case, the communication seems to have disappeared since it does neither use an external channel, nor takes time. This situations can be captured without problems by additional constraints that relate the execution time of a communication to the resources assigned to the tasks involved.

In any case, a task can be viewed as a rectangle in an imaginary time-resource space. On the time axis, the task stretches from T_i over D_i time units. On the resource axis, the task covers resource R_i, by for example stretching between R_i and next resource $R_i + 1$. With this interpretation, the constraint for the exclusive use of resources is that for no rectangles in the time-resource space to overlap:

$$\forall i, j \in N \quad T_i \geq T_j + D_j \vee T_j \geq T_i + D_i \vee R_i \neq R_j \tag{6.6}$$

Since this kind of constraints involve all tasks at once, they are usually referred to as *global constraints*. Although these can be modeled using a disjunction of primitive constraints, most constraint programming environments offer optimized implementations of various global constraints. In particular, *CHIP5* [COS96] offers the **diffn** constraint for modeling non-overlapping n-dimensional rectangles, while *SICStus* [Lab02] offers, for example, the **disjoint2** constraint for 2-dimensional rectangles.

Modeling Energy

The goal of our optimization process is the minimization of the total energy. The total energy is in fact the sum of individual task energies. Once a task is assigned to a resource, we can directly estimate its energy. Assuming the average power consumption for each resource is given, the task energy is simply computed as its execution delay times the average power. For complex tasks, involving a proportionate blend of instructions, the energy estimate computed in this way is not far from the actual energy consumption. Alternatively, if the design process requires accurate metrics, the energy for each task on every resource may be obtained via simulations or actual measurements. The method of obtaining these energy estimates is in fact beyond the scope of our discussion. In any case, for each task i, our method requires an energy estimate e_{ir} on every resource r. The dependency between the resource R_i and task energy

E_i is in fact similar to that between task execution time D_i and its resource R_i:

$$dom(E_i) = \bigcup_{r \in dom(R_i)} \{e_{ir}\}, \quad dom(R_i) = \bigcup_{e \in dom(E_i)} \{p | e_{ip} \equiv e\} \tag{6.7}$$

Finally, the value to minimize is the total energy:

$$E = \sum_{i \in N} E_i \tag{6.8}$$

A simple theoretical lower bound of the energy consumption can be computed for each task graph assuming that:

a. there is no communication over buses between tasks and

b. every task is executed on the processor which requires the lowest possible energy for executing that task.

With these observations the lower bound for the energy can be computed as:

$$E_{LB} = \sum_{i \in N} \min_{j \in dom(R_i)} e_{ij} \tag{6.9}$$

In practice this lower bound is very optimistic, and can be achieved only in very special cases. When all the tasks are executed as in point b given above, in general, they might need to execute on different processors. Consequently, if there is any communication between this kind of tasks, it has to be performed on the bus. On the other hand, if all the communicating tasks are executed on the same processor, such that all use shared memory without loading the bus (as in point a), this might not be the best choice from the computational tasks energy point of view. Finally, even if a and b are satisfied, the time constraints might be violated, leading to invalid solutions. In the experiments presented later on, this is the energy lower bound we use for evaluating the different synthesis algorithms.

Example 6.1 (An Example of Modeling with Constraints):
Consider the task graph composed of three computations and two communications depicted in Figure 6.2. Each node in the graph will be modeled by a 4-tuple of finite domain variables $< T_i, D_i, R_i, E_i >$, denoting the start time, delay, resource, and energy used by task i, respectively. Let us consider that there are three processors available, all able to execute the computations and one bus for communications. We will identify these resources through integers 1,2,3 for processors and 4 for the bus. Note that the domains for the resource variables R_i (equation 6.1) should reflect the fact that computations

can only be executed by processors and communications only executed on the bus:

$$dom(R_1) = dom(R_2) = dom(R_3) = \{1, 2, 3\} \quad dom(R_4) = dom(R_5) = \{4\}$$

The delay D_i and energy E_i for each task, on each of these resources, are also given in Figure 6.2. These relations between resources and delay, are modeled using constraints of type 6.3. For instance, for task τ_3 this appears as:

$$(dom(R_3) = \{1, 3\} \iff D_3 = 1) \vee (R_3 = 2 \iff D_3 = 2)$$

Similar relations have to be written for energy, following the form of constraint 6.7. For task τ_3 this appears as:

$$(dom(R_3) = \{1, 3\} \iff E_3 = 4) \vee (R_3 = 2 \iff E_3 = 1)$$

Note that for communications, there are situations when the delay and energy can become zero, according to our assumption. This is the case for τ_5 when both τ_2 and τ_3 execute on the same processor. This situation is expressed formally as:

$$R_2 = R_3 \iff (D_5 = 0 \wedge E_5 = 0)$$

Regarding timing, let us consider that the task graph has to execute between 0 and 6. This would mean that $dom(T_i) = \{0 \dots 6\}$. Furthermore, we have to enforce the precedence constraints modeled by the arcs in the graph:

$$T_1 + D_1 \leq T_4 \wedge T_2 + D_2 \leq T_5 \wedge T_4 + D_4 \leq T_3 \wedge T_5 + D_5 \leq T_3 \wedge T_3 + D_3 \leq 6$$

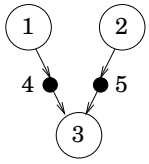
where the last relation expresses the fact that τ_3 must finish before the deadline. Finally, we are interested in the total energy of the system:

$$E = E_1 + E_2 + E_3 + E_4 + E_5$$

This is the measure we want to minimize under the constraints presented above. Note that, according to equation 6.9, the lower bound for this value would be $4 + 1 + 1 + 0 + 0 = 6$, which would mean that all tasks execute on different processors, yet no communication takes place. Obviously, this lower bound is not achievable in this case.

6.2.2 Searching for Solutions

Once the problem has been specified using constraints, the next step is to look for solutions. With the variable set $\mathcal{V} = \{V_1, V_2, \dots, V_n\}$, searching for a solution means assigning gradually to each variable V_i, values from its domain $dom(V_i) = \{w_{i1}, w_{i2}, \dots\}$, and backtracking when no further assignments are



	τ_1		τ_2		τ_3		τ_4		τ_5	
R_i	D_1	E_1	D_2	E_2	D_3	E_3	D_4	E_4	D_5	E_5
1	2	8	5	3	1	4	–	–	–	–
2	4	4	3	4	2	1	–	–	–	–
3	3	6	6	1	1	4	–	–	–	–
4	–	–	–	–	–	–	1	2	1	2

Figure 6.2: Task graph and resource pool for Example 6.1

possible. The search space can be viewed as a search tree, in which every level is associated to one variable (Figure 6.3). After each assignment, the constraints are revised using the specific value for the assigned variable. As a result, some of the domains of the still unassigned variables may shrink, thus pruning the search tree. Whenever an assignment leads to at least one void domain, that branch cannot produce any solution with that partial assignment, so a new assignment must be tried. If there are no further possible assignments, the search must backtrack to the previous level. When all the variables have been assigned, a solution is found. If the purpose of the search is finding one solution, the whole process stops. If the purpose of the search is choosing a solution from a set of solution, other assignments have to be tried, backtracking to the previous level. In particular, we want to find those values for the task start times T_i and resources R_i that yield the minimum value for energy E.

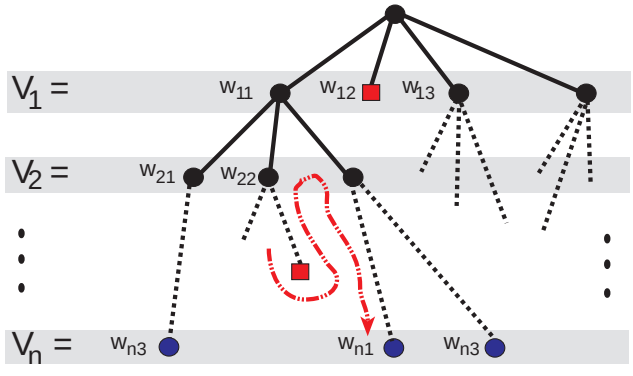


Figure 6.3: A search tree. On every level, a variable is bound to a value from its domain. The square nodes represent assignments which do not lead to any solutions.

This search process has actually a lot of similarities with a typical *branch-and-bound* process. The cost deciding the bounding is also a constrained vari-

able with its own domain. During the search, while propagating (recalculating) the constraints, the domain of the cost variable might become empty. This will yield cutting that branch, and trying others. As in a typical branch-and-bound process, each time a better solution is produced, a new bound is imposed leading to a restriction of the domain of the cost variable. The search strategy presented above explores the entire search space, and it will be referred to as *full-search* or *complete-search*.

Depending on the complexity of the problem, the search space can become huge, and finding all solutions becomes infeasible. In these situations, partial exploration of the search space is used. The solutions found using partial search cannot be proven to be optimal, but they can be near optimal, if the partial search strategy is well chosen.

In our experiments we often use a heuristic named *credit-search*, presented in [BBSC97]. Credit-search starts as a pseudo-exhaustive search at the upper levels of the search tree, and continues as a limited local search on the lower levels. Each decision taken during the search will consume a number of *credit points* from an initial credit. Each branch will get a number of credit points, depending on a preset *distribution ratio*. As soon as the credit points are consumed, the search tries to go on the first available branch. Whenever the search blocks, it backtracks and tries new branches. The number of explored branches is controlled mainly by the initial credit, while the disposal of these branches depends on the distribution ratio. The local search is controlled by the number of *allowed backtracks*. For a sufficiently large initial credit, and/or number of backtracks, credit-search degenerates into a full-search.

To implement our design flow using the constraint programming modeling methods just described, we made use of the COSYTEC logic programming environment CHIP 5.1 [COS96]. All the experiments were performed using this implementation.

6.2.3 Experiments

The experiments described in this section were designed to capture the main features of our single step design approach, using constraint programming. First, they show the numerous design choices when low energy is targeted. The experimental results reflect the trade-off between energy and system cost in terms of resource and speed. Secondly, we compare our energy-directed design flow to other slightly different approaches, easier to implement. These methods attempt to minimize the energy in an indirect way, by tuning parameters influencing the power or energy consumption. We present two of these, targeting communications and resource utilization, respectively. As proven by the experiments, these methods are indeed successful, but only in particular

situations. Finally, the results show that the most successful is our complete energy-directed design flow.

The Speed-Cost-Energy Tradeoff

In order to illustrate the design alternatives from the energy consumption point of view, we used an example introduced in [PP91]. The task-graph containing nine tasks and eight communications is depicted in Figure 6.4. The available resource library contains three types of processors with costs $\{4, 5, 2\}$, having the average power consumptions $\{4, 6, 5\}$, and one bus. We assumed that a bus in use consumes around twice the power of a processor, while all communications take 1 time unit. Also, the tasks have different execution delays on different processor types, yielding different energy consumptions. The 3D graph in Figure 6.4 depicts the design space for the example described above, obtained with a complete search. The bars represent designs having minimal energy consumption for the constraints imposed on both cost and execution time. From these results we conclude that energy savings can be obtained if the limits on deadline and/or maximal cost are relaxed. In principle, more processors and longer execution deadlines make it possible to select better alternatives from the energy point of view.

Optimization Scenarios

The constraint based modeling method presented earlier gives us the possibility to explore and compare different energy optimization alternatives. We considered three energy optimization techniques: the first approach focuses only on the processor energy, the second targets only the communication energy, and, finally, the third combines processor and communication energy consumption.

The first optimization scenario, what we call *the naive designer's method*, is presented in Figure 6.5. It uses a straightforward design method, selecting processors in a greedy manner. The designer starts with an empty set of processors and keeps adding new processors taken from the available resource pool, until a feasible schedule is found for the imposed time limit. The processor to be extracted from the available set and added to the current allocation set is always the processor which has the lowest power consumption. The function **do_synthesis** in Figure 6.5 consists of a branch-and-bound with credit-search (see Section 6.2.2), which tries to assign tasks to processors and to time slots. If no solution is possible with a certain partial assignment, the algorithm backtracks and tries a different assignment until the first possible schedule and binding are found, or there are no more possible partial assignments.

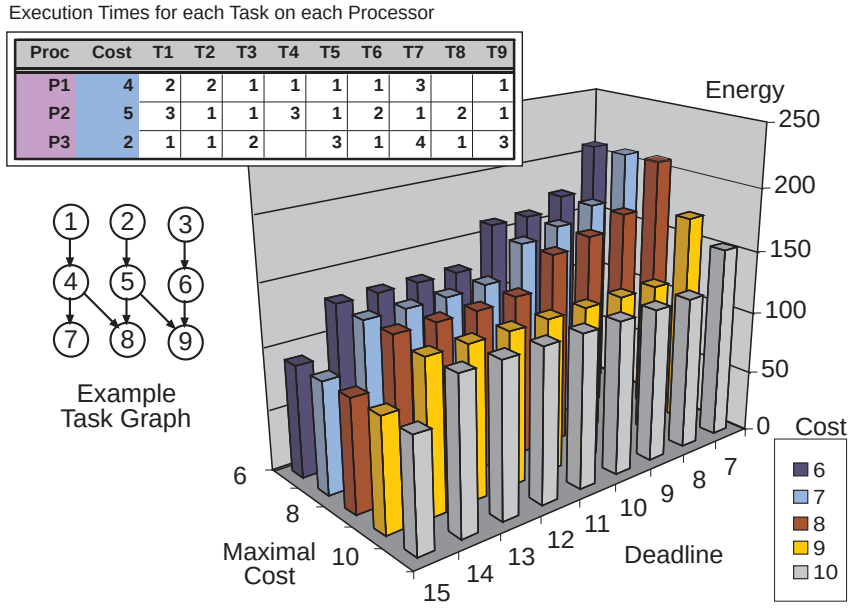


Figure 6.4: An example task graph and possible solutions depicted in a time-cost-energy design space. The bars with the same shade represent designs with the same cost limit.

In the second scenario, we minimize the amount of communication between processors by assigning as many dependent tasks as possible on the same processor. This is carried out by grouping the computational tasks into clusters representing processors, while trying to minimize the cut between clusters. As in the previous scenario, we used the branch-and-bound (B&B) algorithm with credit-search, yet we attempt to minimize the total communication energy. The B&B algorithm determines the resource binding such that the communication on the buses is minimal. Thus, the power consumption of the processors is not directly considered.

In the third scenario, both communication energy and computation energy are considered during optimization. The function to be minimized is the total energy as given in equation 6.8. Again, we use a B&B algorithm with credit-search as in the previous scenarios, but the goal is total energy minimization. The algorithm tries to allocate processors, bind tasks to processors, and schedule the task graph. The B&B algorithm assigns tasks to processors with the minimal power consumption first and then tries to find a schedule for this assignment. The algorithm backtracks if no solution can be found.

```
naive_designer(ResourcePool, Deadline) {
  Allocation ← ∅;
  SolutionFound ← FALSE;
  do {
    (Schedule, Binding) ← do_synthesis(Allocation, Deadline);
    if(valid_solution(Allocation,Schedule,Binding))
      SolutionFound ← TRUE;
    else {
      NewProcessor ← lowest_power(ResourcePool);
      Allocation ← Allocation ∪ {NewProcessor};
      ResourcePool ← ResourcePool \ {NewProcessor}
    }
  } while(!SolutionFound);
  return (Allocation, Schedule, Binding);
}
```

Figure 6.5: The *naive designer's method* pseudo-code.

Note that none of these approaches guarantees optimal solutions, since they are based on partial search of the design space. Yet, they perform quite well, as the experiments presented next are showing.

As input for the experiments, we randomly generated task graphs of different sizes, for two different situations. First, we assumed that the bus (or communication) power is five to ten times lower than the power consumption of the processors. For the second situation, we assumed that the bus power consumption is five to ten times larger than the processor power. Although these two alternatives are extremes in practice, they illustrate the two basic situations we are interested in:

1. when the computation energy is dominant (true mainly for systems-on-chip) and

2. when the communication energy is dominant (true for systems containing expensive communication channels).

For each of the three optimization techniques presented before, we plot two curves, one for each of the two situations described above. For each graph, we determined the shortest possible schedule with the available resources, running separate experiments based on the methods presented in [Kuc99], which yield near optimal results. Relaxing this deadline and applying each of the three methods, we found better designs from the energy consumption point of view. The plotted values are average energy values relative to the energy lower bound as formulated in equation 6.9.

Figure 6.6 presents these experimental results for twenty random graphs with 20 tasks and communications. For these experiments, the resources available were five processors with the power consumptions {6, 7, 8, 9, 10}, and one bus. The bus power consumption is 1, for the naive1, bus1, and all1 curves and 50 for the naive50, bus50 and all50. The naiveX curves are the solutions obtained by the *naive designer*, the busX curves are the ones obtained optimizing only the bus energy consumption, and the allX are the solutions obtained with our unified approach. We also considered large graphs,

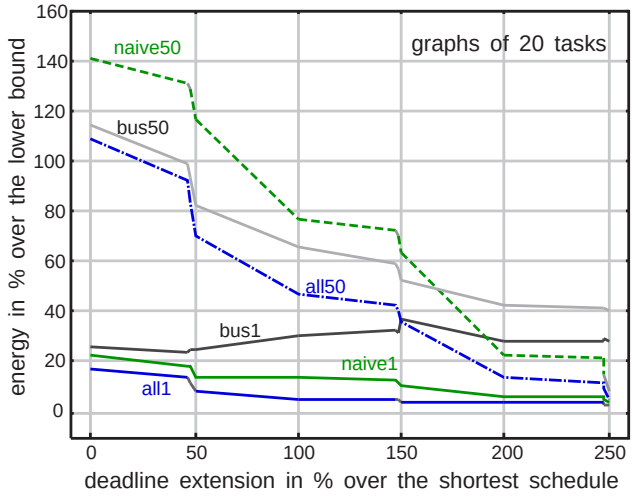


Figure 6.6: Energy consumption vs. schedule length for the graphs of 20 tasks.

comparable to the most complex practical systems. Figure 6.7 presents the curves for twenty random graphs with 130 computational tasks and communications. The resources available in this case were four buses and fifteen processors, three of each kind of the five processors used in the previous experiment. All the experiments were run with the CHIP 5.1 system, on a Sun SPARC center 2000 (50MHz). The time for finding one solution was limited to 10 minutes for the 20 nodes graphs and 20 minutes for the 130 nodes graphs.

The *naive designer* algorithm requires little time, since the first solution found is the one reported. From the diagrams, we can deduce that the *naive designer* approach, although very simple, performs better than the bus oriented approach when the communication power is low. The situation changes when the communication power becomes significant. It is interesting to notice that, for relaxed deadlines, independent of the communication power, the *naive designer* still performs better. This can be explained by the fact that few processors and long deadlines force fewer communications, since more

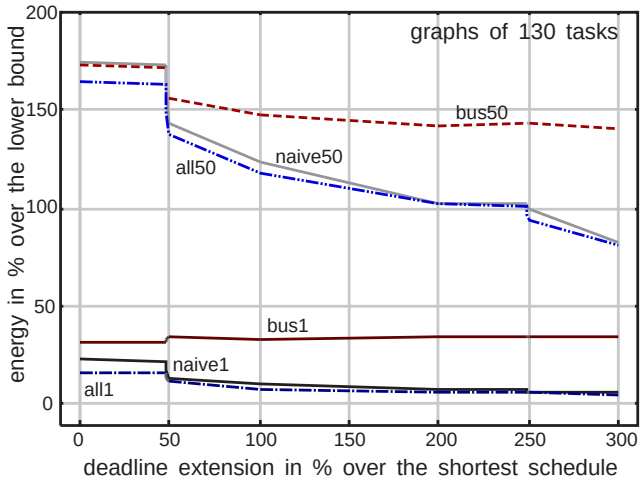


Figure 6.7: Energy consumption vs. schedule length for the graphs of 130 tasks.

and more processes execute on the same processor.

Finally, considering both communications and tasks during optimization is obviously the best strategy. It performs better than the two other techniques, regardless of the communication power and of the deadlines. In real cases, the designers try to obtain rather fast designs, so they focus mainly on tight deadlines, represented somewhere in the left area of our graphs. As shown by the experiments, for these tight deadlines, the unified method obtains much better solutions than any of the other two.

6.3 Variable Speed Processor Architectures

Using a single step approach for architectures with variable speed processors is impractical, since the complexity of the problem grows exponentially. Even on dual-speed processors, tasks may in principle execute at an enormous number of virtual speeds between the two actual clock speeds. For this reason, we adopt a different design methodology, consisting of a two sequential steps, assignment generation followed by fast scheduling (see Figure 6.8), enclosed in the energy-aware optimization loop. Granted such an approach may find fewer good solutions, its speed makes it more suitable as an interactive design space exploration method. To get an energy-efficient design, all the steps in the design flow should get feedback from the subsequent steps. If these steps are performed independently, only local optima can be found.

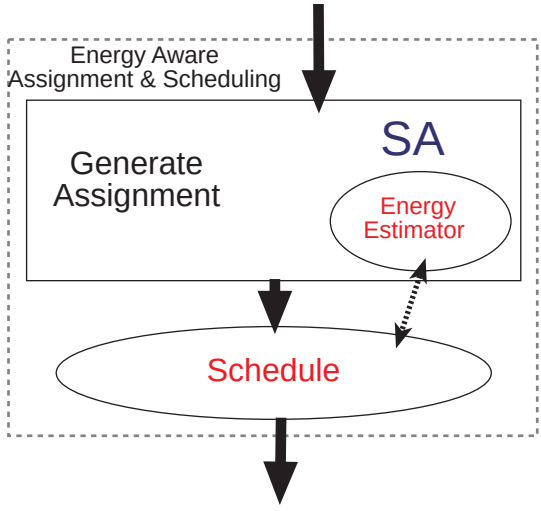


Figure 6.8: A two step design process. The assignment step, guided by a fast energy consumption estimator, is followed by a scheduling step.

For this reason, in the assignment step we use a heuristic search (*Simulated Annealing*, SA, in our implementation) guided by a cost function, estimating the final outcome of the synthesis. In particular, we are interested in estimating the energy consumption after scheduling, therefore the estimator has to take into consideration the scheduling method. We present next two design approaches that differ in the estimation method and scheduling step.

6.3.1 The "Speed-Up and Stretch" Approach

The first method is what we consider one of the simplest, although not trivial, approaches to low energy system design (Figure 6.9 a). It is based on the idea that one can trade execution speed for low power/energy. In principle the task graph is assigned in such way that the schedule is as short as possible, considering all processors run at the maximum speed. This assignment is performed using simulated annealing, mentioned before and detailed later on. Then, this tightest schedule is stretched, by lowering the supply voltage selectively for each processor, such that it covers all the time interval to the desired deadline. For this reason, we call this design flow *Speed-Up and Stretch* (S&S). Scheduling is performed using list-scheduling with a priority function based on critical path. The combined scheduling and stretching method is the same one described in Section 5.3, as *List-Scheduling with Proportional Stretch* (LS-PS). Note that this approach does not require any infor-

mation about the energy consumed by the processors. It uses the assignment and scheduling techniques from a classic, energy-unaware approach.

The idea behind finding the fastest possible schedule comes from the two observations. First, the critical paths become shorter, allowing lower speeds, and thus lower energy consumption. Secondly, the processor load gets balanced and close to the maximum 100%. Consequently, the idle intervals on all processors get smaller. Ideally, for the tightest schedule all processors would be busy 100% of their time. The following, proportional stretch step assigns the lowest (and same) speed for all tasks. Shorter schedule means lower speed.

Finally, the LS-PS scheduling method used by the S&S approach is fast enough for the energy estimator. Once a schedule is found for the given assignment, the energy can be computed in a simple manner, considering the speed is common for all tasks.

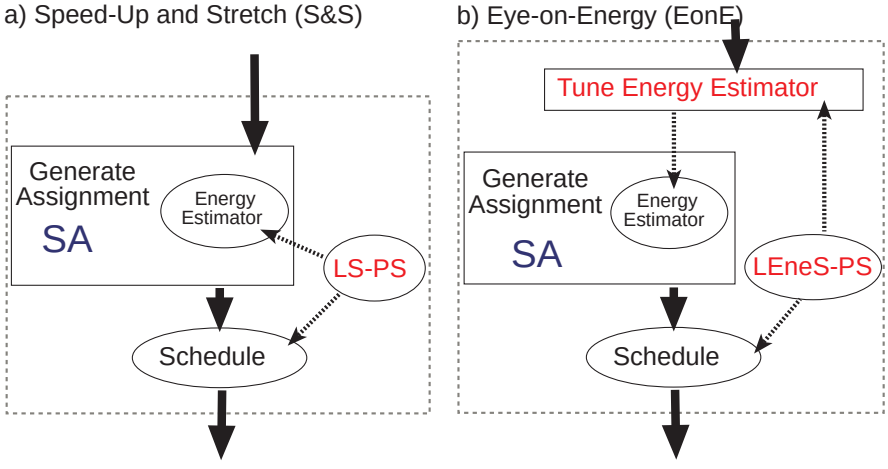


Figure 6.9: Two energy-aware design flows.

6.3.2 The "Eye-on-Energy" Approach

One of the problems with the previous approach is that the scheduling strategy, discussed in Section 5.3, performs badly on heterogeneous sets of processors or for task graphs with unbalanced paths. In Section 5.4 we presented LEneS, a better scheduling algorithm, able to handle the above mentioned situations.

For instance, LEneS is able to reduce the energy consumption by 25% for certain assigned task/graphs, without any performance loss. Although this

improved algorithm saves energy compared to the simpler LS-PS, it takes longer time to find a schedule. For example, scheduling a task-graph with 56 tasks on eight processors takes around 5 minutes. This is far too long to include in an energy estimator contained by the Simulated Annealing loop, required by our design flow. To overcome this problem, we built a separate function, which can quickly estimate the energy consumption when using our LEneS scheduling strategy. A more detailed description of the estimator is presented later on. Finally, using our LEneS strategy enhanced with Proportional Stretch (scheduling algorithm referred to as LEneS-PS in Section 5.4), the new estimation method, we developed an improved design flow.

Our new design flow, called *Eye-on-Energy* (EonE), is depicted in Figure 6.9.b, and works as follows. At first, the energy estimator is tuned for the current task-graph by fitting its estimates to the energy values obtained after a complete (and time-consuming for that matter) scheduling with LEneS-PS. Then we perform a heuristic search using simulated annealing (SA), as in the S&S approach, except this time the search is directed by the estimated energy, as opposed to schedule length. Finally, with the best assignment found by SA, the design is scheduled using LEneS-PS.

The Energy Estimator

Our EonE design flow requires an estimate for the energy consumption of the design obtained assignment and the LEneS scheduling algorithm. This estimate must be computed fast enough to be useful inside the SA search heuristic. The following function was chosen for computing the estimate:

$$\widehat{E} = a\left(\frac{E_{max}}{N_{proc}}\right)\left(\frac{T_{min}}{D}\right)^2 + b\delta_E + c \tag{6.10}$$

where E_{max} is the maximal energy consumption for the given assignment, obtained when all the tasks run as fast as possible. N_{proc} is the number of processors used by the assignment. T_{min} is the length of the fastest schedule, obtained through a classic list-scheduling with a critical-path priority function. D is the required deadline. δ_E is the energy square deviation taken over each processor. Finally, a, b, and c are constants, specific for every task graph. The reason behind choosing such function is the following. The first ratio gives in principle the average maximal energy consumed by each processor. The bigger this value is, the larger the final energy consumption gets. The second ratio describes the scaling ability of the schedule. If this factor is high, meaning that even the tightest schedule needs a long time to complete, the more likely there will be very little extra time for scaling. The square power comes from the energy/delay dependency. The second term in the estimator expression describes how well balanced the energy is among the available pro-

cessors, for the tightest schedule. Several of our experiments suggested that a smaller value for this term leads to a lower final energy consumption.

Parameters a, b, and c are tuned by fitting the function to the energy values obtained for several random assignments, that are then scheduled with LEneS-PS. Given the function shape, we use non-linear regression to determine a, b, and c. In particular, in our implementation we used the non-linear least square fit provided by the *newmat* C++ library [Dav02].

The algorithmic complexity required to compute $\widehat{E}$ for a certain assignment is given by the complexity of determining its components. The most costly of all is T_{min}, the length of the tightest schedule, which is determined through a classic list-scheduling. Thus, the complexity of computing the cost of an assignment is the same as in the S&S approach, which uses LS-PS.

Next we evaluate the accuracy of the energy estimator $\widehat{E}$. This implies comparing the values returned by the estimator against those resulted after LEneS, for a number of different task graphs and assignments. For this, we used one hundred random graphs of thirty nodes, with a dependency depth no larger than five tasks. As a resource pool, we assumed six processors with different average power consumption. The tightest possible deadline with the given resources was obtained using SA, with the schedule length as cost function. First, we fitted the a, b, and c parameters of the estimator using 30 different random assignments for each task-graph. Then, for other 30 random assignments, we examined how well the estimates fit the actual energy values for these assignments. The standard deviation (average, minimum, and maximum) of the estimates for both sets of assignments is depicted in Figure 6.10. Although the estimates for the new assignments (not used for function fitting) are less accurate, the difference is rather small (1-2%). Thus, using only 30 assignments, the estimator parameters are good enough to fit other, random assignments. We conclude that the estimator tuning step in the EonE approach can use only 30 assignments, without significant loss in accuracy. Finally, these experiments show that the average standard deviation of the estimates from the actual energy values obtained by LEneS is around 8%, sufficiently accurate to be used as an estimator in the SA search.

6.3.3 Simulated Annealing as Assignment Search

Finding the best task-to-processor assignment is an NP-hard problem. For this reason we use a well known and easy to implement heuristic search method: simulated annealing. There are several aspects which define a specific implementation of a simulated annealing algorithm, differing between our two design flows as follows.

The neighborhood is defined in terms of one type of move, which is reassigning one randomly selected task to a different processor. In the SA implemented in the S&S approach, a random task is assigned to the processor

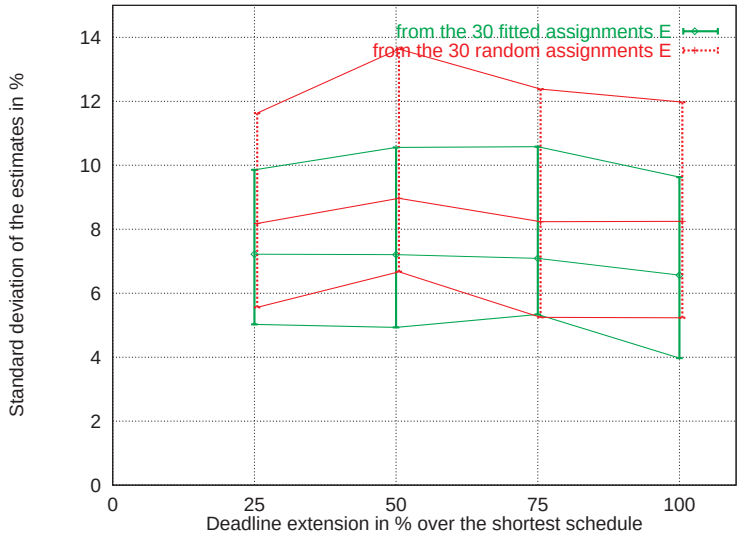


Figure 6.10: The accuracy of the $\widehat{E}$ estimator. The figure depicts average, minimum, and maximum standard deviations of estimates from the LEneS yield energy consumption. Both deviations from the fitted and random assignments are shown.

which yields the fastest execution time for that very task. In the SA implemented in the EonE approach, a random task is assigned to the processor which yields the lowest energy consumption for that task. To allow the search to exit local minima, whenever the selected task is already assigned on the best processor (best local assignment), a random assignment is performed for that task.

The parameters of the SA cooling scheme can be used to speed up the search with the risk of obtaining worse solutions, as shown in Figure 6.11. The number of iterations at one temperature is dynamically computed depending on the problem size and temperature. The stopping condition is given by the dynamics of the solution. If the solution remains unchanged (within a 5% range) in the last three temperatures, the search stops. The SA cost function in the EonE approach is the estimated value of energy. For the S&S approach, the cost function is the schedule length.

6.3.4 S&S vs. EonE Comparison

The most interesting aspect of the two design flows, S&S and EonE, is, of course, their efficiency in producing low energy designs. In the following two

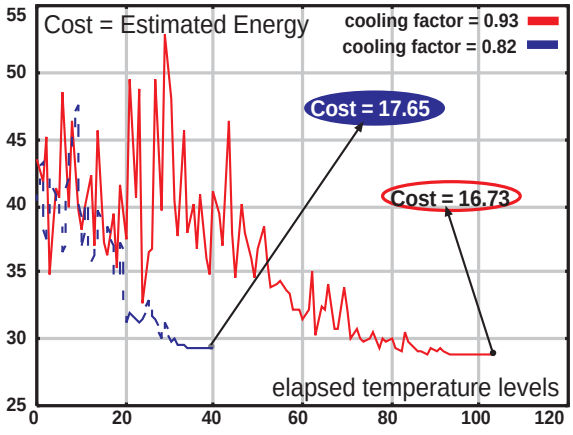


Figure 6.11: Assignment energy evolution during SA search for two different parameter settings.

experiments, we compare the previously mentioned design flows both for artificially created task graphs, and for a real life application.

For the first experiment, we used one hundred graphs with twenty nodes and with a dependency depth no larger than four tasks. The reference deadline is the tightest possible deadline with the available resources. For each task graph, this value was obtained by running SA with the schedule length as cost function. We applied then both S&S and EonE design flows on all these task graphs and recorded the energy consumption of the reported solutions. Figure 6.12 depicts the energy saved by the EonE approach compared to the S&S approach, as average, minimal, and maximal values, for various deadline extensions. On average, EonE can save around 15% energy in comparison to S&S. In some cases, EonE can save as much as 34% energy. The negative values appear because of the energy estimation error (that can be as much as 13% as shown in Figure 6.10), which results in the S&S approach to behave better in some cases.

Next, we present the actual energy saving capabilities of the two design flows introduced in this paper. For this, we chose a real-life application consisting of a sub-system of an Unmanned Aerial Vehicle (see [DGK+00]). The sub-system we are interested in is an optical flow detection (OFD) algorithm, which is part of the traffic monitoring system. In the current implementation, the optical flow algorithm consists of 32 tasks, running on ADSP-21061L digital signal processors. Limited by other tasks, OFD can process 78x120 pixels at a rate up to 12.5Hz. In many cases, a much lower rate is sufficient. Be-

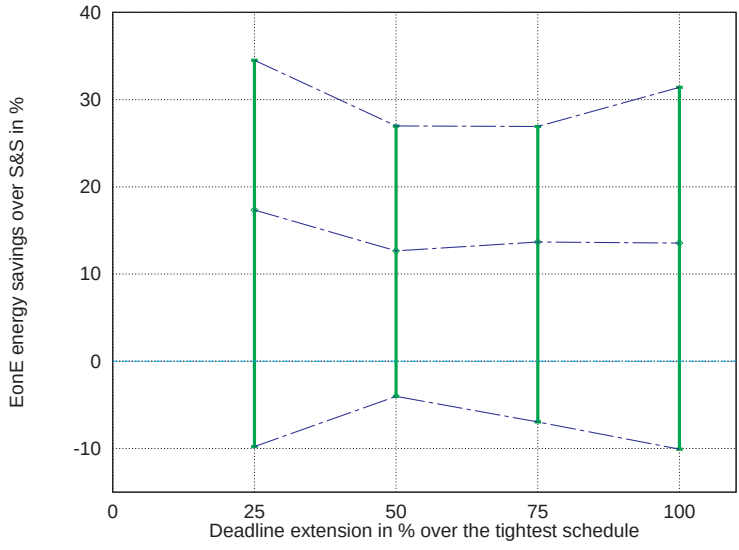


Figure 6.12: The energy saved by applying EonE versus using the S&S design flow. The figure depicts maximum, average, and minimum values for a hundred randomly generated task graphs.

cause of this dynamic behavior, important energy savings would be obtained if the design were to use processors supporting multiple voltages. Depending on the wanted rate, the processors can be run faster or slower, in order to reduce the energy consumption. Assuming we can run the DSPs at 3.3V, 2.5V, 1.7V, or 0.9V, we applied our two low energy design flows for the OFD. For two different processing rates, lower than 12.5Hz (column 1 in Table 6.1), we assumed different pools of available resources: with two, three, and four processors respectively (column 2 in Table 6.1). For each of these configurations we considered three design methods. First, we considered the real current situation, when no voltage scaling can be performed (the *Single* V_{dd} column), but the processors are shut down whenever they are idling. This energy is always the same, since the processors are identical and always execute at the same speed. This is the reference we compare the other methods to. Then, we applied the low energy design flows described before and compared them to the reference energy (the *S&S* and *EonE* columns respectively). As reflected by Table 6.1, there is a trade-off between cost (number of processors) and low energy consumption. Note also that even for only 50% slower processing rate the energy consumption can be almost halved. The simple S&S approach performs fairly well, but for very low energy applications the EonE approach is recommended.

Table 6.1: S&S and EonE solutions energy for two rates of the OFD algorithm.

Processing Rate	Number of Processors	Energy in % of the *Single* V_{dd} case		
		Single V_{dd}	S&S	EonE
6.25 Hz (half rate)	2	100	49.47	48.95
	3	100	45.16	41.18
	4	100	42.07	39.92
8.33 Hz (2/3 rate)	2	100	71.33	69.64
	3	100	60.13	57.04
	4	100	55.86	52.77

6.4 Chapter Summary

The current chapter addressed scheduling in conjunction with architecture selection, as a parts of several energy-centric design flows. In this context, we focused on static scheduling of task graphs, first on fixed speed processor architectures and then on variable speed processor architectures.

For architectures with fixed speed processors, we presented a single step approach for assigning tasks to processors (binding) and to time moments (scheduling). The problem, formulated as a time-and-resource constrained optimization, is modeled using constraints and solved through a branch-and-bound search. We presented a couple of simple heuristics for minimizing the energy consumption, through reducing only the communications or alternatively selecting low power processors. Finally, we presented a unified communication-computation energy minimization strategy. The experimental results show that although simple optimization heuristics perform fairly good, the best results can be obtained only by using the total energy as a minimization goal.

For architectures with variable speed processors, we described a two step design flow, consisting of energy estimate driven binding followed by scheduling. In this context, we presented two variations of this design flow. The first one indirectly minimizes the overall energy consumption, by trying to uniformly load the processors, while finding a short schedule. The second uses our LEneS scheduling method at its core, and is driven by a fast energy estimator. Finally, the experimental results show that the first approach, although very simple, behaves surprisingly well, achieving low energy designs. Still, the second approach, employing especially designed energy-centric techniques, performs best.

CHAPTER 7

FINAL REMARKS

THIS CONCLUDING CHAPTER presents a summary of the work contained in this thesis, together with a few important conclusions that can be drawn from our results. It also highlights some of the possible future developments and trends that could shape the area of low power and low energy digital system design. Finally, the chapter closes with a brief enumeration of the possible directions for our research work.

7.1 Summary and Conclusions

Energy consumption is today an issue emerging in increasingly many applications, ranging from battery powered portable devices to desktop computers. Some of these applications are time critical, falling under the umbrella of real-time systems. For these types of systems, task scheduling is an essential design aspect. Classic real-time design techniques do not address the scheduling problem from the energy efficiency point of view. Conversely, the typical methodologies for designing low energy systems often have a hard time in fulfilling the constraints of a real-time system. Energy efficiency usually means carrying out operations at a lower pace, on slower but power-efficient resources. Therefore, the choice of resources and the schedule influences the energy consumed by the system. With the advent of the variable speed processors — whose speed, thus energy consumption may be adjusted at run-time — the connection between energy efficiency and scheduling becomes even stronger. Scheduling in this case implies also selecting the right speeds for each task in each time moment.

The current thesis presents several speed scheduling algorithms for systems containing variable speed processors, yielding energy-efficient designs,

while still fulfilling the hard real-time constraints.

The thesis starts with a presentation of the relevant background, followed by a review of related research. The task, task group, and resource models used throughout the thesis, are then introduced.

Next, a set of speed scheduling choices at task level were described and compared, including stochastic scheduling. This is our own task level scheduling method, that minimizes the expected case energy consumption by employing the task execution pattern probability distribution. Stochastic scheduling is examined side by side with complier-assisted scheduling, from various points of view, such as energy efficiency, run-time overhead, and implementation complexity. The experimental results obtained on a hardware platform with a variable speed processor (see Appendix B) shown at this point, confirm the validity of our starting assumptions and the success of our method. We conclude that the best task level scheduling strategy is highly case dependent, but can be detected using careful analysis.

A wide range of task group scheduling policies were described next. First, we focused on static scheduling for task graphs on variable speed processors. A few simple low energy scheduling heuristics were examined, employing simple proportional scaling transformations. We then presented LEneS, a more advanced algorithm based on list-scheduling with an energy-sensitive priority function. Briefly, LEneS reduces the energy by assigning lower speeds to non-critical tasks. Next, we focused on sets of independent tasks running on single processor systems. We presented methods for computing the minimum required speeds for each task, under rate-monotonic and earliest deadline first scheduling schemes. Then we described an energy-efficient run-time extension of the rate-monotonic policy, using slack re-distribution. The worst case response time analysis presented at this point, shows that our method behaves as well as than the classic rate-monotonic scheduling policy, while achieving important energy savings. The task group scheduling chapter concludes with the description of an ordering policy built on top of the earliest deadline first scheduling, entitled Uncertainty-Based Scheduling. The policy, designed for tasks with variable execution pattern, uses stochastic information about the tasks to derive orders that yield, at run-time, minimal energy consumption in the long run. The evaluation presented there, showed that the ordering algorithm is much faster and, at the same time, as efficient as a simulated annealing heuristics. The energy measurements performed on the available hardware system confirmed both the importance of a good ordering and the efficiency of our own policy.

Finally, we examined task graph scheduling in conjunction with architecture selection, as parts of low energy system-level synthesis. In this context, we presented several design flows directed at lowering the energy consumption while fulfilling the real-time constraints. First, we described a constraint

programming based approach for solving task mapping and scheduling in a unified manner, while minimizing the system energy. As pointed out, a good methodology for design space exploration reveals the cost vs. low energy vs. performance tradeoff, which offers many choices to a designer. Next, we presented two more system-level design flows for energy-efficient architectures of variable speed processors. One of them is a very simple approach, that takes initially energy-unaware techniques, but combines them into an energy-centric design methodology. The other approach directly targets energy consumption reduction during task mapping and scheduling. A comparison between the two design flows shows that, although the simpler method performs fairly well, the best results are always obtained using our specially designed energy-aware techniques.

In conclusion, even hard real-time systems can be made energy efficient, if one employs methods as the ones presented in this thesis. Performing computations as slow as possible, while fulfilling the deadlines, is the key to low energy real-time systems. In this sense, architectures composed of variable speed processors seem to offer the best support platform for energy efficiency. Furthermore, better scheduling algorithms can be designed if stochastic data about the tasks execution time is taken into consideration. Additionally, these algorithms may be implemented at operating system level, improving the software portability.

7.2 Future Trends

The importance of any research results concerning speed scheduling is best examined in the context of future possible developments in the digital appliances area. Variable speed systems offer today the best combination of performance and energy efficiency. Highly configurable or programmable hardware support are preferred because of their flexibility. In this sense, variable speed processors are the best solution for a wide area of applications. Such processors already make their way into desktop computers, laptops, PDAs[1], and all sorts of embedded systems. Judging by their huge potential to reduce the system cost (energy consumption, cooling) while maintaining performance, it is our conviction that in the near future most processors will be variable speed processors. Furthermore, the overhead of the speed switch, which accounts for a couple of thousand clock cycles today, will most likely decrease significantly, using architectural solutions similar to the one described in Section 3.3.2. The number of speed settings, today usually between two and eleven, will probably increase slightly, pushing the energy/delay characteristic closer to an ideal one. Although synchronous design is very likely to remain the

[1]PDA: Personal Digital Assistant

basis for the majority of processors, asynchronous designs showing promising power-performance characteristics have already been presented [EGT01] and could some day replace the synchronous processors. However, even asynchronous processors can be designed to support multiple speed settings by varying the supply voltage. Therefore, speed scheduling methods remain necessary, regardless of the synchronous-asynchronous choice.

In the future, operating system level speed management is expected as opposed to task/user level speed scheduling carried out at compile time. This is because the operating system has, in principle, a better view and control over all system resources. Another reason is portability and code mobility among systems with different variable or fixed speed processors.

To take advantage of the intrinsic parallelism of some applications and consume as little energy as possible, multi-processor systems seem to be the best option. The increasing number of distributed applications will require a system level energy manager, that will take global rather than local choices. Such a manager should adaptively choose the best distribution of speeds for each component, together with the required computation and communication amount to fulfill the system functionality in an energy efficient manner. An example of this kind of application are wireless networks.

7.3 Future Work

Regarding possible improvements or extensions of our current work, we would like to examine more aggressive methods to incorporate stochastic data into existing real-time scheduling strategies. In this sense, we plan to evaluate in detail our Uncertainty-Based Scheduling policy for more general cases than those presented in this thesis. In principle, we are interested in examining the influence of the overhead introduced by the additional context switches required by UBS.

To further verify the efficiency of our methods, we intend to implement some of these inside the IVM[2], a real-time Java Virtual Machine designed for embedded systems, developed in our department [Ive03]. We plan to use the already familiar XScale evaluation board to examine the energy consumption for the enhanced IVM.

Finally, since wireless networks seem to become more common, in particular sensor networks, we intend to look more closely into energy management for distributed systems. Problems that are especially interesting in this case are adaptive functionality partitioning depending on the locally available energy, coupled with task migration, routing, and communication for low energy.

[2]IVM: Infinitesimal (Java) Virtual Machine

BIBLIOGRAPHY

[ADI] ADI Engineering. *80200EVB Reference Platform*. http://www.adiengineering.com/product80200EVB.html.

[AMD00] AMD. *AMD PowerNow™ Technology Dynamically Manages Power and Performance, Rev. A*, November 2000. Informational White Paper No. 24404.

[AMMMA01] H. Ayidin, R. Melhem, D. Mossè, and P. Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 225–232, 2001.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Axe97] J. Axelsson. *Analysis and Synthesis of Heterogeneous Real-Time Systems*. PhD thesis, Linköping Technical University, 1997. No. 502.

[BBSC97] N. Beldiceanu, E. Bourreau, H. Simonis, and P. Chan. Partial search strategy in CHIP. In *Proceedings of the 2nd Metaheuristic International Conference*, 1997.

[BD99] L. Benini and G. DeMicheli. System-level power optimization: Techniques and tools. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, pages 288–293, 1999.

[BDM+97a] L. Benini, G. DeMicheli, E. Macii, M. Poncino, and S. Quez. System-level power optimization of special purpose applications: the Beach solution. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pages 24–29, 1997.

[BDM+97b] L. Benini, G. DeMicheli, E. Macii, D. Sciuto, and C. Silvano. Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems. In *Proceedings of the Seventh Great Lakes Symposium on VLSI*, pages 77–82, 1997.

[BMR90] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard real-time sporadic tasks on one processor. In *Proceedings of IEEE Real-Time Symposium*, 1990.

[Bor99] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July-August 1999.

[BPSB00] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid State Circuits*, 35(11), November 2000.

[BRH90] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2, 1990.

[Bur91] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, pages 116–128, May 1991.

[BW01] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Adison-Wesley, 3rd edition, 2001.

[CBD01] E.-Y. Chung, L. Benini, and G. DeMicheli. Automatic source code specialization for energy reduction. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Desing*, pages 80–83, 2001.

[CC01] K.-W. Choi and A. Chatterjee. Efficient instruction-level optimization methodology for low-power embedded systems. In *Proceedings of the 14th International Symposium on System Synthesis*, pages 147–152, 2001.

[CGX96] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: An approach for energy efficient computing. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, pages 347–352, 1996.

[Cof76] E. G. Jr. Coffman. Introduction to deterministic scheduling theory. In *Computer and Job-Shop Scheduling Theory*. Wiley, New York, 1976.

[Com02] Compaq, Intel, Microsoft, Phoenix, and Toshiba. *Advanced Configuration & Power Interface Specification, Revision 2.0a*, March 31 2002.

[COS96] COSYTEC. *CHIP, System Documentation*. COSYTEC, 1996.

[CP97] J.-M. Chang and M. Pedram. Energy minimization using multiple supply voltages. *IEEE Transactions on VLSI Systems*, 5(4):436–443, December 1997.

[Cyg] Cygnus. *GNUPro Tools for Intel® XScale™*. Cygnus, Cygnus Solutions, 1325 Chesapeake Terrace, Sunnyvale, CA 94089, USA. http://www.cygnus.com/.

[Dav02] R. B. Davies. *Documentation for newmat10A, a matrix library in C++*, October 2002.

[DeM94] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[DGK+00] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The WITAS unmanned aerial vehicle project. In *Proceedings of the 14th European Conference on Artificial Intelligence*, 2000.

[Dic02] R.P. Dick. *Multiobjective Synthesis of Low-Power Real-Time Distributed Embedded Systems*. PhD thesis, Princeton University, Dept. of Electrical Engineering, November 2002.

[DJ97] R.P. Dick and N.K. Jha. Mogac: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. In *Proceedings of the 1997 International Conference on Computer Aided Design*, pages 522–529, 1997.

[DKV+02] V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, and I. Kolcu. Compiler-directed array interleaving for reducing energy in multi-bank memories. In *Proceedings of the 15th International Conference on VLSI Design*, 2002.

[DLJ99] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems. *IEEE Transactions on VLSI Systems*, 7(1):92–104, March 1999.

[EGT01] A. Efthymiou, J.D. Garside, and S. Temple. A comparative power analysis of an asynchronous processor. In *Proceedings of the 11th International Workshop — Power And Timing Modeling, Optimization and Simulation*, pages 10.1.1–10, September 2001.

[EKP98a] P. Eles, K. Kuchcinski, and Z. Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, 1998.

[EKP+98b] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings of the 1998 Design, Automation and Test in Europe*, pages 132–138, 1998.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, 1979.

[Fle01] M. Fleischmann. LongRun power management - dynamic power management for crusoe processors. Technical report, Transmeta Corporation, January 17, 2001.

[GCW95] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *ACM International Conference on Mobile Computing and Networking*, pages 13–25, 1995.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[GK83] D. D. Gajski and R. Kuhn. Guest editors' introduction: New VLSI tools. *IEEE Computer*, 6(12):11–14, December 1983.

[GK99] F. Gruian and K. Kuchcinski. Low energy architecture selection and task scheduling for system-level design. In *Proceedings of the 25th EuroMicro Conference*, volume 1, pages 296–302, September 1999.

[GK01] F. Gruian and K. Kuchcinski. LEneS: Task-scheduling for low-energy systems using variable voltage processors. In *Proceedings of the 2001 Asia South Pacific – Design Automation Conference*, pages 449–455, January 30 – February 2 2001.

[Gru00a] F. Gruian. Energy-aware design of digital systems. Licentiate Thesis 809, Linköping Technical University, IDA, March 2000.

[Gru00b] F. Gruian. System-level design methods for low-energy architectures containing variable voltage processors. In B. Falsafi and T.N. Vijaykumar, editors, *Lecture Notes in Computer Science*, number 2008, pages 1–12. Springer, 2000. First International Workshop on Power-Aware Computer Systems.

[Gru01a] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 46–51, August 6–7 2001.

[Gru01b] F. Gruian. On energy reduction in hard real-time systems containing tasks with stochastic execution times. In *IEEE Workshop on Power Management for Real-Time and Embedded Systems*, pages 11–16, May 29 2001.

[HK01] C.-H. Hsu and U. Kremer. Compiler-directed dynamic voltage scaling based on program regions. Technical Report DCS-TR461, Rutgers University, November 2001.

[HK02] C.-H. Hsu and U. Kremer. Single region vs. multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches. In *Workshop on Power-Aware Computer Systems*, 2002.

[HKQ+98] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 1998 Design Automation Conference*, pages 176–181, 1998.

[HPS98] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Digest of Technical Papers of ICCAD'98*, pages 653–656, 1998.

[Int00] Intel. *Intel® XScale™ Core Developer's Manual*, December 2000. Order Number: 273473-001.

[Int01] Intel. *Intel® 80200 Processor based on Intel® XScale™ Microarchitecture Datasheet*, September 2001. Order Number: 273414-003.

[Int02] Intel. *Mobile Intel® Pentium® 4 Processor-M with 512KB L2 Cache on .13 Micron Process at 1.6 GHz and 1.7 GHz Datasheet*, April 2002. Order Number: 250686-002.

[Ive03] A. Ive. Implementation of an embedded real-time Java virtual machine prototype. Licentiate thesis, Dept. of Computer Science, Lund University, 2003. To be published.

[IY98] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 197–202, 1998.

[JG02] R. Jejurikar and R. Gupta. Energy aware task scheduling with task synchronization for embedded real time systems. In *Proceedings of the 2002 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 164–169, 2002.

[Jha01] N. K. Jha. Low power system scheduling and synthesis. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer Aided Design*, pages 259–263, 2001.

[KB99] A. Kumar and M. Bayoumi. Multiple voltage-based scheduling methodology for low-power in the high-level synthesis. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, volume 1, pages 371–374, 1999.

[KKM02] W. Kim, J. Kim, and S. L. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 788–794, 2002.

[KP97] D. Kirovski and M. Potkojak. System-level synthesis of low-power hard real-time systems. In *Proceedings of the 34th Design Automation Conference*, pages 697–702, 1997.

[KRH+96] N. Kim, M. Ryu, S. Hong, M. Saksena, C.-H. C.-H. Choi, and H. Shin. Visual assessment of a real-time system design: A case study on a CNC controller. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 300–310, 1996.

[KSY+02] W. Kim, D. Shin, H. S. Yun, J. Kim, and S. L. Min. Performance comparison of dynamic voltage scaling algorithms for hard real-time systems. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.

[Kuc99] K. Kuchcinski. Synthesis of distributed embedded systems. In *Proceedings of the 25th EuroMicro Conference*, volume 1, pages 22–28, September 1999.

[Kuc01] K. Kuchcinski. Constraints driven design space exploration for distributed embedded systems. *Journal of Systems Architecture*, (47):241–261, 2001.

[KW01] K. Kuchcinski and K. Wolinski. Synthesis of conditional behaviours using hierarchical conditional dependency graphs and constraint logic programming. In *Proceedings of the 2001 Euromicro Symposium on Digital Systems Design*, pages 220–227, 2001.

[Lab02] Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, 3.9.1 edition, June 2002.

[LJ00] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time systems. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer Aided Design*, pages 357–364, 2000.

[LJ02] J. Luo and N. K. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proceedings of the 15th International Conference on VLSI Design*, pages 719–726, 2002.

[LK99] Y.-H. Lee and C. M. Krishna. Voltage-clock scaling for low energy consumption in real-time embedded systems. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 272–279, 1999.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprograming in a hard real time environment. *JACM*, 20(1):46–61, 1973.

[LLM+01] M. Lorenz, R. Leupers, P. Marwedel, T. Drager, and G. Fettweis. Low-energy DSP code generation using a genetic algorithm. In *Proceedings of the 2001 International Conference on Computer Design*, pages 431–437, 2001.

[LM80] J.Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3), 1980.

[Loc92] C. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–35, 1992.

[Lor01] J. R. Lorch. *Opertaing Systems Techniques for Reducing Processor Energy Consumption*. PhD thesis, University of California Berkeley, 2001.

[LRT92] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the 1992 Real Time Systems Symposium*, pages 110–123, 1992.

[LS00a] S. Lee and T. Sakurai. Run-time power control scheme using software feedback loop for low-power real-time applications. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, pages 381–386, 2000.

[LS00b] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 2000 Design Automation Conference*, pages 806–809, 2000.

[LS01] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of ACM SIGMETRICS 2001 Conference*, pages 50–61, 2001.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the 1989 Real Time Systems Symposium*, pages 166–171, 1989.

[LVM91] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in Ada: a case study. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 181–189, 1991.

[MACM00] D. Mossè, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power*, October 2000.

[Mar00] D. Marculescu. Power efficient processors using multiple supply voltages. In *Workshop on Compilers and Operating Systems for Low Power*, 2000.

[MAX00] MAXIM. *MAXIM MAX1855 Evaluation Kit*. MAXIM, first edition, 2000. http://www.maxim-ic.com/.

[MC00] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *Proceedings*

of the 2000 IEEE International Conference on Acoustics, Speech, and Signal Processing, volume 6, pages 3239–3242, 2000.

[MC01] A. Manzak and C. Chakrabarti. Variable voltage task scheduling algorithms for minimizing energy. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 279–282, 2001.

[MC02] A. Manzak and C. Chakrabarti. A low power scheduling scheme with resources operating at multiple voltages. *IEEE Transactions on VLSI Systems*, 10(1):6–14, February 2002.

[MLD92] P. Michel, U. Lauther, and P. Duzy, editors. *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, 1992.

[MOI96] H. Mehta, R. M. Owens, and M. J. Irwin. Some issues in gray code addressing. In *Proceedings of the Sixth Great Lakes Symposium on VLSI*, pages 178–181, 1996.

[MOI+97] R. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pages 72–75, 1997.

[OIY98] T. Okuma, T. Ishihara, and H. Yasuura. Real-time scheduling for a variable voltage processor. In *Proceedings of the 12th International Symposium on System Synthesis*, pages 25–29, 1998.

[OIY01] T. Okuma, T. Ishihara, and H. Yasuura. Software energy reduction techniques for variable voltage processors. *IEEE Design & Test of Computers*, pages 31–41, March-April 2001.

[PBB98] T. A. Pering, T. D. Burd, and R. W. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, 1998.

[PBB00] T. A. Pering, T. D. Burd, and R. W. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 96–101, 2000.

[PD99] P. R. Panda and N. D. Dutt. Low-power memory mapping through reducing address bus activity. *IEEE Transaction on VLSI Systems*, 7(3):309–320, September 1999.

[Ped01] M. Pedram. Power optimization and management in embedded systems. In *Proceedings of the 2001 Asia and South Pacific – Design Automation Conference*, pages 239–244, 2001.

[PKVI01] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. VLIW scheduling for energy and performance. In *Proceedings of the 2001 IEEE Computer Society Workshop on VLSI*, pages 111–117, 2001.

[PLS01] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, pages 28–33, 2001.

[PP91] S. Prakash and A. C. Parker. Synthesis of application-specific multiprocessor architectures. In *Proceedings of the 28th Design Automation Conference*, pages 8–13, 1991.

[PP92] S. Prakash and A. C. Parker. Synthesis of application-specific heterogeneous multiprocessor systems. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, page 434, 1992.

[PS01] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 2001 Symposium on Operating System Principles*, pages 89–102, 2001.

[QH01] G. Quan and X. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proceedings of the 2001 Design Automation Conference*, pages 828–833, 2001.

[QH02] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processors. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 782–787, 2002.

[QKPS99] G. Qu, D. Kirovski, M. Potkonjak, and M. B. Srivastava. Energy minimization of system pipelines using multiple voltages. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, volume 1, pages 362–365, 1999.

[RJ98] J. T. Russell and M. F. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors.

In *Proceedings of the 1998 International Conference on Computer Design: VLSI in Computers and Processors*, pages 328–333, 1998.

[RP96] J. M. Rabaey and M. Pedram, editors. *Low Power Design Methodologies*. Kluwer Academic Publishers, 1996.

[SAH01] M. T. Schmitz and B. M. Al-Hashimi. Considering power variations of dvs processing elements for energy minimization in distributed systems. In *Proceedings of the 14th International Symposium on System Synthesis*, pages 250–255, 2001.

[SAHE02] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 514–521, 2002.

[SB95] M. R. Stan and W. P. Burleson. Bus-invert coding for low-power I/O. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1):49–58, March 1995.

[SBD01] T. Simunic, L. Benini, and G. DeMicheli. Energy-efficient design of battery-powered embedded systems. *IEEE Transactions on VLSI Systems*, 9(1):15–28, February 2001.

[SC98] W.-T. Shiue and C. Chakrabarti. Low-power scheduling with resources operating at multiple voltages. In *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, volume 2, pages 437–440, 1998.

[SC99] Y. Shin and W. Choi. Power conscious fixed priority scheduling for real-time systems. In *Proceedings of the 36th Design Automation Conference*, pages 134–139, 1999.

[SC01a] A. Sinha and A. P. Chandrakasan. Energy efficient real-time scheduling [microprocessors]. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer Aided Design*, pages 458–463, 2001.

[SC01b] V. Swaminathan and K. Chakrabarty. Investigating the effect of voltage-switching on low-energy task scheduling in hard real-time systems. In *Proceedings of the 2001 Asia and South Pacific – Design Automation Conference*, pages 251–254, 2001.

[SCS00] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedigns of the 2000 IEEE/ACM International Conference on Computer Aided Design*, pages 365–368, 2000.

[SGK99] R. Szymanek, F. Gruian, and K. Kuchcinski. Application of constraint programming to digital system design. In *Proceedings of the 1st Workshop on Constraint Programming for Decision and Control*, pages 57–64, 1999.

[SGK00] R. Szymanek, F. Gruian, and K. Kuchcinski. Digital system design using constraint logic programming. In *Proceedings of PACLP*, pages 10–12, 2000.

[Shi01] W.-T. Shiue. Energy-efficient backend compiler design for embedded systems. In *Proceedings of IEEE Region 10 International Conference on Electrical and Electronic Technology*, volume 1, pages 103–109, 2001.

[SKJ+02] D. Shin, W. Kim, J. Jeon, J. Kim, and S. L. Min. SimDVS: An integrated simulation environment for performance evaluation of dynamic voltage scaling algorithms. In *Workshop on Power-Aware Computer Systems*, 2002.

[SKL01] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2), March-April 2001.

[SKS01] Y. Shin, H. Kawaguchi, and T. Sakurai. Cooperative voltage scaling (CVS) between OS and applications for low-power real-time systems. In *Proceedings of the 2001 IEEE Custom Integrated Circuits Conference*, pages 553–556, 2001.

[SR99] M. Sarrafzadeh and S. Raje. Scheduling with multiple voltages under resource constraints. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, volume 1, pages 350–353, 1999.

[SSRB98] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[TMW94a] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of the 1994 IEEE Symposium on Low Power Electronics*, pages 38–39, 1994.

[TMW94b] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transaction on VLSI Systems*, 2(4):437–445, December 1994.

[Wol97] W. H. Wolf. An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(2):218–229, 1997.

[WWDS94] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[YDS95] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995.

[YWM+01] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Design & Test of Computers*, 118(5):46–58., 2001.

[ZHC02] Y. Zhang, X. Hu, and Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 2002 Design Automation Conference*, pages 183–188, 2002.

APPENDIX A

PROOFS

A.1 Stochastic schedule energy lower bound

We have to prove that the average energy $\overline{E}$ as described by equation 4.9 has the lower bound E_{LB} as given by equation 4.10. For the sake of clarity, we will substitute now x by i, WCE by N, $\sqrt[3]{(1 - cdf(x))}$ by a_i and k_x by b_i. The relation to prove becomes then:

$$\sum_{i=1}^{N} \frac{a_i^3}{b_i^2} \geq \frac{\left(\sum_{i=1}^{N} a_i\right)^3}{\left(\sum_{j=1}^{N} b_j\right)^2} \quad \forall i, j : 0 \leq a_i < b_j \tag{A.1}$$

We will prove this inequality by mathematical induction. First we show that the above holds for the simple case of $N = 2$. Moving everything to the left side, we obtain:

$$\begin{aligned} \frac{a_1^3}{b_1^2} + \frac{a_2^3}{b_2^2} - \frac{(a_1 + a_2)^3}{(b_1 + b_2)^2} &\geq 0 \quad \Longleftrightarrow \\ \frac{(a_2 b_1^2 + 2a_2 b_2 b_1 + 2a_1 b_1 b_2 + a_1 b_2^2)(a_2 b_1 - a_1 b_2)^2}{(b_1 b_2 (b_1 + b_2))^2} &\geq 0 \end{aligned} \tag{A.2}$$

which is obviously true, since all the terms in the left side are non-negative. Next, assuming the inequality holds for $N = k$, we show that it also holds for $N = k + 1$. We have thus equation A.1 for $N = k$ given as true:

$$\sum_{i=1}^{k} \frac{a_i^3}{b_i^2} \geq \frac{\left(\sum_{i=1}^{k} a_i\right)^3}{\left(\sum_{j=1}^{k} b_j\right)^2} \tag{A.3}$$

Adding $\frac{a_{k+1}^3}{b_{k+1}^2}$ to both sides we obtain:

$$\sum_{i=1}^{k+1} \frac{a_i^3}{b_i^2} \geq \frac{\left(\sum_{i=1}^{k} a_i\right)^3}{\left(\sum_{j=1}^{k} b_j\right)^2} + \frac{a_{k+1}^3}{b_{k+1}^2} \tag{A.4}$$

Furthermore, equation A.2 can be applied to the right hand side:

$$\frac{\left(\sum_{i=1}^{k} a_i\right)^3}{\left(\sum_{j=1}^{k} b_j\right)^2} + \frac{a_{k+1}^3}{b_{k+1}^2} \geq \frac{\left(\left(\sum_{i=1}^{k} a_i\right) + a_{k+1}\right)^3}{\left(\left(\sum_{j=1}^{k} b_j\right) + b_{k+1}\right)^2} \tag{A.5}$$

From the last two inequalities, A.4 and A.5, it results:

$$\sum_{i=1}^{k+1} \frac{a_i^3}{b_i^2} \geq \frac{\left(\sum_{i=1}^{k+1} a_i\right)^3}{\left(\sum_{j=1}^{k+1} b_j\right)^2} \tag{A.6}$$

which is exactly the form for $N = k + 1$. We have shown that the case for $N = k + 1$ is true, assuming the form for $N = k$ is true. Finally, together with A.2, this means that the initial inequality A.1 holds for any value of N, by mathematical induction. Q.E.D.

This result can be applied to equation 4.4 from Section 4.2. Examining the right hand side of equation 4.4 and using A.1, we can write:

$$E = \mathcal{K} \sum_{i=1}^{C} \frac{1^3}{A_i^2} \geq \mathcal{K} \frac{C^3}{\left(\sum_{i=1}^{C} A_i\right)^2} \tag{A.7}$$

Considering that $\sum_{i=1}^{C} A_i = A$ we obtain the energy lower bound invoked in Section 4.2:

$$E \geq \mathcal{K} \frac{C^3}{A^2} \tag{A.8}$$

A.2 Optimal order for UBS

We are interested in finding the optimal task order, defined as a permutation $\pi(1, 2, \ldots, n)$, such that the following measure (energy consumption) is minimized:

$$\begin{aligned} E &= x_{\pi_1} + x_{\pi_2}\left(\frac{\sum_{k=1}^{n} c_{\pi_k} - c_{\pi_1}}{A - x_{\pi_1}}\right)^2 + \cdots \\ &\cdots + x_{\pi_i}\left(\frac{\sum_{k=1}^{n} c_{\pi_k} - \sum_{j=1}^{i-1} c_{\pi_j}}{A - \sum_{j=1}^{i-1} x_{\pi_j}}\right)^2 + \cdots + x_{\pi_n}\left(\frac{c_{\pi_n}}{A - \sum_{j=1}^{n-1} x_{\pi_j}}\right)^2 \end{aligned}$$

We transform now this problem into another problem, from the Network Flow class [Eve79]. Initially, any task may start executing, using the same initial speed. Subsequently, any new choice has to be made from the tasks waiting to execute. Note that the speed after certain k tasks is dependent only on the set of executed tasks, but not on their order:

$$s(\{\pi_1, \ldots, \pi_k\}) = \frac{\mathcal{C} - \sum_{i=1}^{k} c_{\pi_i}}{A - \sum_{j=1}^{k} x_{\pi_k}}$$

where $(C) = \sum_{i=1}^{n} c_{\pi_i}$. We model each such state as a graph node, in an acyclic polar graph depicted in Figure A.1. The root node, having no predecessors represents the initial state, where the set of executed task is $\emptyset$. The sink node represents the state where all tasks have been ordered. The number of nodes on the level k, after executing k tasks, is equal to C_n^k, the number of combinations of k from n. From each node on level k, there are $n - k$ outgoing edges, since any of the remaining tasks may be scheduled next. Each edge between level k and $k + 1$ can be augmented with the energy consumption yielded by scheduling task π_{k+1} after the set of previous k tasks. More precisely, each edge from a node representing a set $\mathcal{S}$ to a node representing a set with an additional node $\mathcal{S} \cup \{p\}$ will have a label:

$$l_{\mathcal{S},p} = x_p s(\mathcal{S})^2$$

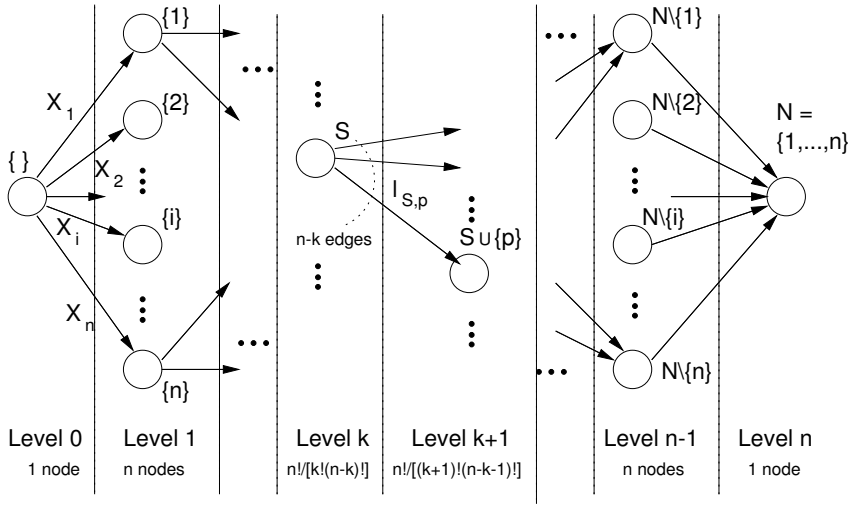


Figure A.1: The acyclic polar digraph obtained from the initial problem. Each route from the root node to the sink is a possible order of execution for the n tasks.

It is obvious now that every path from the initial state $\emptyset$ to the final state represents an order, while the sum of edge labels represents the energy consumption for that order. Finding the order with the minimal energy is a problem similar to computing the minimum completion time in a PERT[1] digraph [Eve79]. The time complexity of such an algorithm can be kept down to $O(|V|)$ where $|V|$ is the number of edges. Note that for our problem the number of edges can be computed as:

$$|V| = nC_n^0 + (n-1)C_n^1 + \ldots + (n-k)C_n^k + \ldots + C_n^{n-1}$$

which is larger than 2^n where n is the number of tasks to be ordered. In other words, finding the optimal order using the transformation just described would require a time complexity of order at least $O(2^n)$. Note that this is not a proof, but rather an indication that our initial problem might be NP-hard.

[1] PERT: Program Evaluation and Review Technique

APPENDIX B

THE TEST SYSTEM

THE SYSTEM USED TO EVALUATE our dynamic speed scaling algorithms was provided by Intel, and consists of three separate boards. The main board, ADI 80200EVB (also referred to as LRH), home of the Intel 80200 XScale processor, an external voltage supply board for the 80200 core, MAX1855 EV kit, and an adapter board between the LRH and the MAX1855.

B.1 ADI 80200EVB

The ADI Engineering's 80200EVB evaluation board [ADI] for Intel 80200 (733MHz) XScale features, along with the processor, a 80200 FPGA Companion Chip (a Xilinx® XC2S200 Spartan-II™) handling the SDRAM control, processor interface, peripheral interface, system clock generation, and a planned PCI interface. The on-board memory consists of 32MB PC100 SDRAM (x64 configuration) and 4MB Flash (x8 configuration). The peripheral interface is an RS232, used mainly for uploading/debugging software. Additionally, on the 80200EVB board there are a 7-segment LED display and a JTAG (hardware testing) port.

The 80200 companion chip generates the 100MHz memory bus clock, the processor core clock (5 ... 11 multiples of 66.666MHz), and the serial port baud clock, up to a 57.6K rate. Since the processor requires the core clock to be at least 3 $\times$ faster than the memory clock, the lowest frequency for the 80200 is 333MHz (5 $\times$ 66.6MHz). The core clock frequency can be changed according to the Intel 80200 XScale core specification [Int00], by accessing co-processor 14, register 6. On the other hand, the core voltage is handled by the companion boards, described further on. Note that there are no dedicated

real-time clocks or *watchdog* timers on the LRH, which makes it almost impossible to accurately keep track of time exclusively on the processor. First, there is no way to determine the duration of a frequency switch on the processor, but only from outside. Further more, to wake up the processor from a low power state, an external signal is required, such as an interrupt or a reset. For these reasons we used an external signal generator (HP 3312A Function Generator), that generates a nIRQ whenever new tasks are supposed to arrive. Although the nIRQ is masked and therefore not actually handled, it still wakes up the processor from the sleep state. This signal is also used to synchronize the I_{cc} and V_{cc} signals in the oscilloscope, which makes it possible to sample and report averages over hyper-periods. For the software part, we used the GNUPro Toolkit from CYGNUS [Cyg]. All code was compiled for XScale on a Sun SPARC Solaris host, using an ELF format.

B.2 MAXIM MAX1855 EV kit

The MAX1855 evaluation kit (EV kit) [MAX00] demonstrates a dynamically adjustable notebook CPU power supply application circuit. The DC-DC converter steps down high-voltage or AC adapters, generating a precision low-voltage CPU core V_{cc}. The output voltage, adjustable through a 5-bit DAC, ranges between 0.6V and 1.75V, with up to 18A current load. This output voltage is used for the Intel 80200 core voltage, V_{cc}. The selection can be done through a program write at address 0x0060.0000, for which only the lower 4 bits are valid. These are translated to the 5 bits D[4:0] required by the EV kit via a look-up table located on the adapter board. In our configuration, the value at address 0x0060.0000 can be in the $3 \ldots 9$ interval, corresponding to voltages in the 1.6V ... 1.0V interval. The EV kit operates at 300kHz switching frequency for voltages around 1.35–1.6V and slightly slower for lower voltages. Note that while the V_{cc} is changed by the EV kit, the processor still operates. Recall that the clock frequency switch stalls the processor until the PLL re-locks on the new frequency. The voltage and frequency are thus adjusted separately. This requires careful design of a switch, since the voltage must support the core clock frequency. In particular, when increasing the processor speed, first one has to increase the voltage and only after that increase the frequency. The reverse must be done when decreasing the speed, namely first lower the frequency and then the voltage.

B.3 Measurements Setup

All the measurements were carried out using a Tektronix TDS 340A two channel digital real-time oscilloscope. Obtaining the core energy consumption

meant tracing the power consumption P_{cc} or the core voltage-current product $V_{cc} \times I_{cc}$ over time. The core voltage is easily accessible on the LRH board. The core current I_{cc} was measured indirectly by measuring the voltage drop on a 0.1Ω resistor in series with the 80200 core, already present on the LRH board. Since this drop was very small, we used the amplifier from Figure B.1, to increase this measure ten times. Finally, at the output of the amplifier we obtained a voltage computed as $\approx I_{cc} \times 0.1\Omega \times 10$, or exactly the core current. Once we were able to measure the core power, it was possible to average the

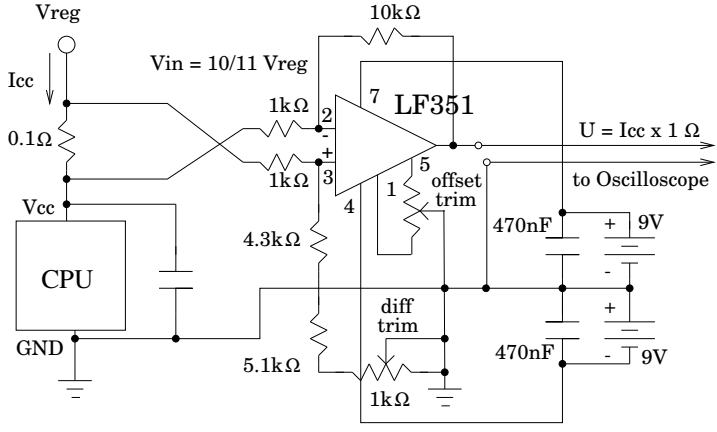


Figure B.1: The setup for measuring the core current I_{cc}. The voltage drop on the 0.1Ω resistor is amplified using an LF351 operational amplifier.

power over several task set hyper-periods or task periods. Throughout the thesis there are often oscilloscope screen dumps with such waveforms. Whenever we had to measure the energy consumption, which is the area under the power waveform, we could output the waveform to a file, in *time-power* pairs of values. These files were then processed on a workstation to obtain the exact energy figures for the setup.

APPENDIX C

A TASK LEVEL STOCHASTIC SCHEDULE ON 80200EVB

The following listing contains the code of a task with normal distribution pattern, that can be scheduled both using the stochastic approach and the WCE-stretch method, used in Example 4.3. The code as given compiles for a stochastic schedule. To obtain a WCE-stretch schedule, one needs to simply comment the 4th line. For a stable oscilloscope trace we generated nIRQs with a $200ms$ period, meant to wake up the processor from the idle state, as detailed in Appendix B.1.

```
#include "../H-sparc-sun-solaris2.5/xscale-elf/include/math.h"
#include "../H-sparc-sun-solaris2.5/xscale-elf/include/stdlib.h"

#define STOCH
#define TOLOOP2 10000

#define cpwait asm("mrc p15,0,r0,c2,c0,0; mov r0,r0; sub pc,pc,#4")

#define MHz733 9
#define MHz666 8
#define MHz600 7
#define MHz533 6
#define MHz466 5
#define MHz400 4
#define MHz333 3

inline void switch_frequency(unsigned number) {
  /*
     9 -> clk mult factor 11 -> 733 MHz
     ...
```

```
    3 -> clk mult factor 5 -> 333 MHz
  */
  asm("ldr r1, %0; mcr p14, 0, r1, c6, c0, 0 " : : "m" (number));
}

#define VCC733 4
#define VCC666 4
#define VCC600 5
#define VCC533 6
#define VCC466 7
#define VCC400 8
#define VCC333 9

inline void switch_voltage(unsigned number) {
  asm("ldr r0, %0; mov r1, #0x00600000; str r0,[r1]": : "m" (number));
}

inline void go_to_idle() {
  unsigned intctlstat;
  /* enable write to cp13 (and all others) */
  asm("ldr r0, =0x3FFF; mcr p15, 0, r0, c15, c1, 0");
  cpwait;
  /* enable external IRQs via INTCTL */
  asm("mov r1, #2; mcr p13, 0, r1, c0, c0, 0");
  /* disable IRQs */
  asm("mrs r0, CPSR; orr r0,r0,#0xd0; msr CPSR_c, r0");
  /* go to idle mode 1, drowsy 2 */
  asm("mov r0, #1; mcr p14, 0, r0, c7, c0, 0");
}

/* Allowed: 2775.0 Period: 2775.0 Wcet=1850.0
   VDD:[ 186 640 181 109 79 62 593 ] */
inline int delayaction(float n) {
  unsigned i,j,k,z;
  for(i=0;i<n;i++) {
    switch(i) {
#ifdef STOCH
    case 186:
      switch_voltage(VCC400);
      switch_frequency(MHz400);
     break;
    case 826:
      switch_voltage(VCC466);
      switch_frequency(MHz466);
      break;
    case 1007:
      switch_voltage(VCC533);
      switch_frequency(MHz533);
     break;
    case 1116:
      switch_voltage(VCC600);
      switch_frequency(MHz600);
      break;
```

```
    case 1195:
      switch_voltage(VCC666);
      switch_frequency(MHz666);
      break;
    case 1257:
      switch_voltage(VCC733);
      switch_frequency(MHz733);
      break;
#else
    case 1150:
      switch_voltage(VCC533);
      switch_frequency(MHz533);
     break;
#endif
    default:;
    }
    for(j=0;j<TOLOOP2;j++) {
      k++;
      z=z+z;
    }
  }

#ifdef STOCH
  switch_frequency(MHz333);
  switch_voltage(VCC333);
#else
  switch_frequency(MHz466);
  switch_voltage(VCC466);
#endif

  return z;
}

inline float polarBoxMuller() {
  float x1, x2, w, y1, y2;
  do {
    x1 = 2.0 * rand()/(float)RAND_MAX - 1.0;
    x2 = 2.0 * rand()/(float)RAND_MAX - 1.0;
    w = x1 * x1 + x2 * x2;
  } while ( w >= 1.0 );

  w = sqrtf( (-2.0 * logf( w ) ) / w );
  y1 = x1 * w;
  /* y2 = x2 * w; */

  /* just make sure it's between -3 and 3 (std=1, mean=0) */
  if(y1>3.0) y1 = 3.0;
  if(y1<-3.0) y1=-3.0;
  return y1;
}

int main() {
  float repeats;
```

```
  /* get to the right voltage and frequency */
#ifdef STOCH
  switch_frequency(MHz333);
  switch_voltage(VCC333);
#else
  switch_frequency(MHz466);
  switch_voltage(VCC466);
#endif

  while(1) {
    repeats = 950.0+polarBoxMuller()*300.0;
    /* repeats = 1850; */
    delayaction(repeats);
    go_to_idle();
  }
}
```