Technical University of Denmark

Rubathas Thirumathyam

Online Dynamic Voltage Scaling in Embedded Real-Time Systems

An extensible simulation framework

Master's Thesis, January 2012

DTU Informatics
Department of Informatics and Mathematical Modelling

Rubathas Thirumathyam

Online Dynamic Voltage Scaling in Embedded Real-Time Systems

An extensible simulation framework

Master's Thesis, January 2012

Online Dynamic Voltage Scaling in Embedded Real-Time Systems: *An extensible simulation framework*

This thesis was prepared by
Rubathas Thirumathyam, s052507

Supervisor
Paul Pop, Associate Professor
Embedded Systems Engineering

DTU Informatics
Department of Informatics and Mathematical Modelling (IMM)
Technical University of Denmark
Richard Petersens Plads Building 321 DK-2800 Kgs. Lyngby
Denmark

http://www.imm.dtu.dk
Tel: (+45) 45 25 33 51
Fax: (+45) 45 88 26 73
EAN: 5798000430204
E-mail: reception@imm.dtu.dk

Release date:	20th of January 2012
Thesis number:	IMM-M.Sc.-2012-xx
Edition:	First (public)
Comments:	This thesis is part of the requirements to achieve the degree, Master of Science in Engineering (MScE), at the Technical University of Denmark. This report represents 30 ECTS points.

Preface

This thesis of 30 ECTS points has been written at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark.

The idea of working with this topic came from following the course *Fundamentals of Modern Embedded Systems*, which introduced the various scheduling problems in real-time systems. Hence, I saw the potential in dynamic voltage scaling and later the necessity of a simulation framework.

I would like to sincerely express my gratitude towards my supervisor, Paul Pop, for not only giving me the opportunity to work in this field, but for also being helpful, understanding and encouraging throughout the whole project. Thanks to PhD-student Junhe Gan for her time explaining her work within DVS.

Furthermore, I would like to thank my family and friends who accepted the time I could spare - in many situations it was lesser than they deserved. Thanks to Odense Universitetshospital and in general the Danish Healthcare system for their professional service, kindness and not giving up on people.

Finally, I would like to quote a person, which philosophy this thesis follows:

That's been one of my mantras - focus and simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there you can move mountains. — Steve Jobs

- Rubathas Thirumathyam

Abstract

Embedded systems are everywhere, from mobile phones to medical devices. They are becoming more complex, and have to fulfill competing requirements, in terms of performance (timing constraints), reliability and energy consumption (long battery life). Due to the competing constraints, their design is becoming increasingly difficult. In this thesis we consider hard real-time applications mapped on distributed heterogeneous architectures. The applications are modeled as a set of tasks, which are characterized by a worst-case execution time and a deadline. The processors in the heterogeneous architecture have have multiple operating modes, consisting of a given frequency and voltage. We assume that we know the reliability of the architecture components. Embedded systems are often designed using static approaches, where the implementation is derived offline. However, many applications require more flexible solutions. Hence, researchers are advocating adaptive approaches, which can change the system configuration in response to changes in the requirements and the environment.

The objective of this thesis is to design and implement online adaptive scheduling algorithms, which are able to successfully address competing design objectives in terms of performance, energy consumption and reliability. We have adapted performance, energy and reliability model from the literature. The trade-off between performance and energy consumption has been addressed using dynamic voltage scaling (DVFS), i.e., reducing the dynamic power consumption by scaling down operational frequency and circuit supply voltage. However, lowering the voltage to reduce the energy consumption will impact negatively the reliability, and we plan to investigate also this energy/reliability trade-off.

We have designed and implemented a simulation framework. Two main algorithms have been designed, implemented and compared: Low-Power Priority-Based Rate Monotonic (LPP) and Cycle-Conserving Rate Monotonic (CCRM). The simulation framework has been implemented using Java and multiple third party libraries. The framework is extensible, and we have shown that it can used to successfully evaluate the quality of online scheduling algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The common modern society has past the last couple of decades become more and more dependent on embedded mobile devices. Examples of such devices in the private consumer market are the legendary and popular products from Apple (iPod, iPhone, iPad). While in the industry they tend to be devices, such as control devices, medical aids, sensors and security measurements. These devices are then placed in bigger systems, like airplanes, cars, buildings and so on. The very purpose of these devices ranges from making everyday life easier, optimize business processes, enjoy entertainment to ensure healthiness, safety and security. Each device can have different and multiple purposes, target audiences, usabilities and requirements.

However, one thing they have in common is being mobile and consuming power from a limited resource, the battery. The vendors will try to increase the consumer's dependence to their device, so they can sustain their market positions or enter new markets, since their primary goals are to make profit and please the shareholders. This can be accomplished in many ways, but mainly done by increasing the functionalities on the device, which again heavily affects the power consumption on the device. Hence, it often becomes a dilemma for the vendors when taking decisions regarding functionality versus power consumption. In the literature there has been made extensive research to overcome this dilemma. Some researchers had made their focus in efficiency when implementing the functionalities, while others centre around the power consumption of the *Central Processing Unit* (CPU). In the latter case, especially, *Dynamic Voltage Scaling* (DVS) has taken up a lot of attention.

1.1 Dynamic Voltage Scaling

Dynamic Voltage Scaling (DVS) can be explained as a design technique to adjust the utilized supply voltage on the CPU during a system's execution. A CPU's energy consumption has a quadratic dependency on the supply voltage. Hence, it is of great advantage to be able to reduce the supply voltage when possible, since it can reduce the energy consumption dramatically. A reduction in energy consumption can prolong the lifetime of the device and help overcome the previous mentioned dilemma the vendors are faced with.

However, there exists a correlation between a CPU's voltage and it's operating frequency. Whenever the supply voltage is reduced it will, beside

a reduced energy consumption, most often result in a lower operating frequency on the CPU. The main objective of a CPU is to handle and execute tasks, and the operating frequency of the CPU determines how quickly (or slow) a given task is executed. Thus, a decrease in the supply voltage and frequency comes with the prize of longer execution times of tasks.

1.1.1 DVS algorithm

A strategy describing how much and when the voltage should be altered based on relevant input-data is characterized as a *DVS algorithm*. As all algorithms, the purpose of developing an algorithm is to solve a (computational) problem. In this present context, the DVS algorithm is integrated into a *scheduler*. The scheduling algorithm within the scheduler decides which task is to be allowed execution on the CPU, while the DVS algorithm decides which voltage level - thereby which frequency level - the CPU should execute the task with. Based on the known relevant information regarding the tasks in the system the DVS algorithm should produce an optimal voltage level decision, which gives energy savings and still satisfy any constraints in the system.

For the past decade there has been, through research, developed several interesting DVS algorithms. Each algorithm has been extensively described in each of their respective academic published paper. Nevertheless, throughout the years these algorithms have become more and more "outdated". Mainly, the assumptions are no longer realistic and new constraints have been put forward.

A bigger problematic is that a profound comparison of the DVS algorithms is difficult to setup. Often a new DVS algorithm is proposed, because the proposer intend to have a better strategy of how to compute the optimal voltage level, including altering the existing assumptions or making new constraints. In any case, it put a great lot of preliminary work upon the proposer to actually prototype the proposed DVS algorithm. Another hassle is the fact to re-implement other major DVS algorithms to make comparisons and an in-depth evaluation of the proposed DVS algorithm.

1.2 Arising issues and challenges

An evaluation of a DVS algorithm based on simulation runs is crucial. The evaluation is an empirical analysis of the outlined strategy in the algorithm and see how the algorithm behaves under various circumstances and scenarios. However, setting up an environment to generate simulation runs and make an evaluation is time consuming and far too often the solution is a setup, which just satisfies the needs of that specific concerned DVS algorithm.

1.2.1 Simulation framework

To sustain a high level of quality and integrity in academia, a key factor is to be able to re-produce published results. In the process of either accepting or rejecting proposed hypotheses, elements of creativity can lead to more optimal solutions and new hypotheses can arise. This essence of science and research should also exist in the development of DVS algorithms.

A general simulation framework is necessary, so it can be straightforward to implement a proposed DVS algorithm. The framework will make it easy to compare a DVS algorithm with other algorithms.

1.2.2 Constraints on the DVS processor

Dynamically reducing the voltage and hereby obtaining energy reduction, is *not* supported by all available CPUs for mobile embedded devices. It is only lately that the CPU vendors have started targeting the mobile embedded devices with DVS CPUs, and enabled the support for changing the voltage while the CPU is on and executing tasks. When developing a DVS algorithm there has to be made considerations concerning the consequences which emerges, when the supply voltage is changed.

Overheads

The consequences are not only longer response times in the system. A change in the voltage induces overheads in two ways. One overhead is in the form of physical time. When the CPU is informed to change the supply voltage, there is a period where it has to adjust to the new supply voltage and operating frequency, before it can start executing tasks again. Another overhead is in the form of extra energy spend, since every change in the supply voltage requires energy to make the transition.

Discrete vs. Continuos frequency levels

The majority of the DVS algorithms should be renamed to Dynamic Frequency Scaling (DFS) algorithms, since they output a frequency and not a supply voltage value. These algorithms pass the responsibility of transforming the frequency into voltage to the scheduler, which there is no harm in. The concern is these algorithms have based their strategy upon continuos frequency levels. There exists no CPU, which supports continuous voltage and frequency scaling. In practice, every DVS CPU has a finite number of supply voltage levels, with corresponding frequency levels, which are interchangeable. Assuming a CPU can operate in any given frequency makes the energy saving computed by the algorithm an upper bound for the actual energy saving, since in practice the chosen frequency will be the

”discrete” available frequency level above the suggested frequency outputted by the DVS algorithm. Hence, the actual energy saving will be less than the originally presumed energy saving.

However, the idea of having an utopian CPU, which can operate at any given ”continuos” supply voltage and frequency level, is reasonable. Decoupling the practicality of the CPU architecture from the decision problem is a good way of simplifying and concentrating on a partial solution.

Reliability

Recently, preeminent work (5) has shown a noncritical approach towards DVS algorithms can have fatal consequences. The consequences when dynamically changing the supply voltage are not limited to overheads in the system. The fault-rate of transient errors increases, when reducing the supply voltage on the CPU. (5) suggests to add an extra constraint in the system, namely a reliability threshold. The function of this extra constraint is to guarantee a certain reliability in the system. Adapting this constraint into the DVS algorithm, makes the algorithm more *pessimistic*, because potential energy savings are not exploited due to possible infringement of the reliability threshold. Hence, the energy savings will be less, however, the reliability and availability of the system is maintained. Fault-tolerance of the system could be extended further. Tolerating transient faults by *replication* or *re-execution* of critical tasks can increase the reliability of the system.

Multiple constraints and assumptions are made in the literature regarding DVS algorithms. When a new DVS algorithm is proposed the assumptions are not necessary identical with other DVS algorithms. Additionally, new constraints could have been developed.

1.3 Related work

Dynamic voltage scaling per se is not a new problem. In spite of the area has been well-studied, only few has tried to establish a simulation framework to operate from as a baseline. The most prominent work has been done by (9), which is quite extensive and presents sound concepts to model a simulation framework. In the preface of this thesis-project, the ambition was to extend the simulation framework developed by (9), such that trade-offs and impacts could be analyzed and discussed based on chosen strategies in the algorithms. However, in the first preliminary phase of the project it was realized that the research group, had discontinued any further work with the simulation framework. It was not able to capture the source code or any executable of the simulation framework, since the responsible code maintainer Dr. Woonseok Kim had passed away. This was a great set back, since this highly related work now was harder to take advantage of.

Nevertheless, their written published work (9) has been of great help and been taken into consideration, in the process of designing and modeling a new simulation framework.

1.4 Thesis objective

1.4.1 Problem

Online dynamic voltage scaling algorithms are difficult for a researcher to develop and assess singularly through an evaluation, but also evaluate towards other online DVS algorithms.

Problem statement

It is a problem, that dynamic voltage scaling algorithms cannot be evaluated in a streamlined manner, such that the trade-offs and impacts can be thoroughly analyzed.

Questions

- *Which relevant online DVS algorithms exist today?* Describe them and their characteristics briefly.
- *Which requirements and architecture should be established for a simulation framework?* Analyze the requirements and the overall goal is: online DVS algorithms should be easily implemented, evaluated and compared to other online DVS algorithms.
- *How can the outlined algorithms be implemented and evaluated?* The technical assessment of how the algorithms can be implemented.
- *Which extensions to the algorithms are possible to incorporate?* Analyze possible extensions that can improve the outlined algorithms.
- *Why is a unified simulation framework essential?* Discuss the importance of the unified simulation framework.

1.4.2 Limitations

Simplification of problems is an approach to make the problems easier to understand and establish limitations, so the core problem can be addressed and not distracted by heavy ”detouring”. Often projects have resource constraints, e.g. in the form of time and manpower, and it becomes necessary to limit the project to move forward. The choices of limitations and assumptions should be based on justifications, such that *oversimplification* is

avoided. To illustrate which areas exist within real-time systems and DVS, a mind-map is shown in figure 1.1.

DVS has many exciting and potential areas; one area is IntraDVS, where the scaling becomes more advanced and instead of deciding one single voltage level for a given task (InterDVS), IntraDVS makes voltage level decisions based on execution paths via a Control Flow Graph (CFG) for a task. Basically, IntraDVS splits a task into multiple basic blocks of computation and assigns a voltage for each individual block. This thesis scope covers InterDVS, since developing an code-level analysis of all task becomes too extensive and the problem statement can still be sufficiently satisfied by only supporting InterDVS. Other possible areas are inter-task communication and bus power management, which due to constraints not have been addressed in this thesis.

The above-mentioned limitations are general, additional limitations have been made. They are mentioned along with their reasons in their concerning sections, since the context they appear in are necessary to properly understand them.

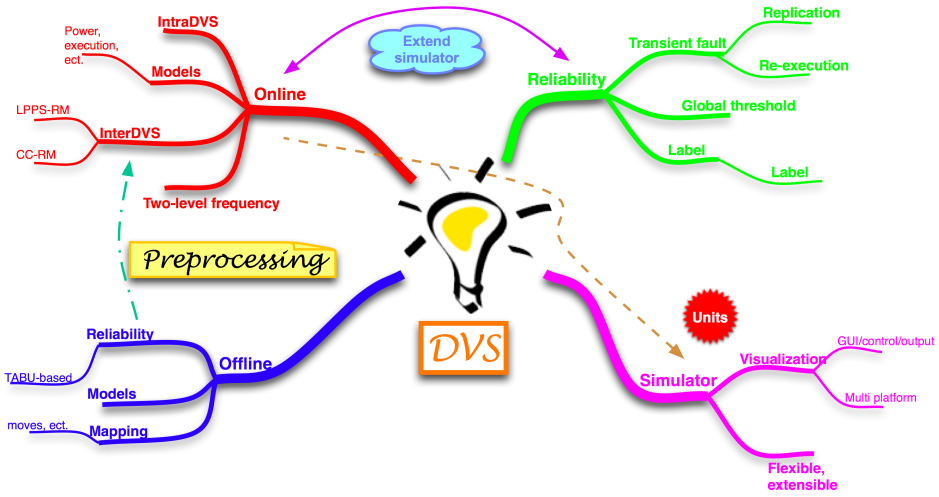


Figure 1.1: DVS mind-map

1.4.3 Method of choice

The applied method, to approach the problem and questions, is more or less accomplished in a deductive manner, where recognized theories and models are identified and a simulation framework is developed. The framework becomes both a resulting product of this thesis and a tool to produce more primary data, based on the outlined and extended online DVS algorithms. The generated data could be used to either accept or reject the hypotheses

made in the analysis.

1.5 Structure of the report

At the outset, chapter 1 contains an introduction to dynamic voltage scaling, followed by a problem, which is transitioned into a problem statement and multiple research questions. These questions will be addressed during the report.

In chapter 2 an analysis is made where all relevant models are defined with their associated theory. A conceptual architecture is explained regarding the software simulator, giving the reader a foundation in how to understand the upcoming chapters concerning the implementation of the simulator and the experiments derived from the simulator.

Hereafter, chapter 3 goes into technical details of the simulator. The implementations of the specified models are untangled along with decision choices and design patterns.

After the reader has got a sense around the problem, central theories, models and implementations, chapter 4 explains the strategy for experiments, and how the simulator is run based on various scenarios and the results are outputted.

Chapter 5 takes the results further. The algorithms and their characteristics are discussed based on the results obtained in the earlier chapter. Interpretations of the results in the light of problem statement, from chapter 1, and theories, from chapter 2, are presented in this chapter.

The final chapter 6 wraps up conclusions made during the earlier chapters and makes way for future work, which could not be enlightened due to the constraints and choices of limitations.

Chapter 2

Analysis

In this chapter the theory and models behind the foundation of the simulation framework is introduced. The conceptual architecture along with the requirements and use case of the framework are explained.

2.1 Real-Time Systems

The comprehension of DVS could be improved by understanding the context within it is used. It is essential to realize that DVS is applied in *real-time systems*, which again is an important part of an *embedded system*. A real-time system contains of a set of *real-time applications*. The objective of a real-time application is to complete tasks and complete them under the given constraints, namely time or resources. A typical real-time application is hidden and difficult for a user, even a technically skilled one, to distinguish from a non real-time application (6). There has previously been mentioned in which devices real-time applications can be found. Often the existence of a real-time system is only given away for a user, when it is not behaving according to it's requirements specification. This accustoming towards the invisibility of real-time systems can have fatal consequences if the real-time system fails to meets it's requirements. In most situations the user will not be able to realize the malfunctioning of the system before it is too late. Even if there is an early detection of the malfunctioning and recovery procedures are not provided by the real-time system, it is difficult for users to overhaul the system. Nonetheless, in the theory of scientific real-time systems there is a specific set of principles and concepts to specify and check the correctness of a given real-time system, *Safety-Critical Embedded Systems*, but this thesis does not go into further details of safety-critical embedded systems.

2.1.1 Reference model of a Real-Time System

Here at the outset, terms and notation should be put in place. A task in a real-time system is characterized as some kind of work, which is scheduled to be executed. Every task has several attributes associated with itself. Especially, worst-case execution time (WCET), release time, deadline, period and priority are significant attributes.

Worst-case execution time

WCET is the absolute maximum execution time a task will run for, then also said that a task not necessary always will run as long as it's WCET. Since the execution time is inversely proportional with the operational frequency of the CPU, it is assumed a task's WCET is when the CPU is run at it's maximum operational frequency. If the frequency is changed at a later point, self-evidently, the task's WCET will change as well.

Release time

Release time is a constant which is only relevant once and could be described as a guaranteed delay. Since not all tasks are interested in being executed immediately after the system starts up, the release time indicates when the task should be made available in the pool of ready tasks to the scheduler after the system start-up.

Deadline

Deadline is more or less self-explanatory. The scheduler must schedule such that the task is finished executing by the CPU before it's deadline, otherwise there is a deadline miss and the consequences can be fatal. If one or more deadline misses arise it is an indication of one of two incidents. Either the set of tasks actually cannot be scheduled or the schedulability analysis of the scheduler has been compromised. In the first case, there is nothing to do than re-do the composition of tasks. In the latter case, the scheduling algorithm is inadequate and has to be re-modeled. A brief introduction to schedulability analysis is further described in section 2.1.2.

Period

The period represents the length of time of recurrence for a periodic task. As with release time, this time only indicates that the task is released and ready to be executed on the CPU. There can be made no assumptions that the task absolutely will be executed at the beginning of the period, since it is the scheduler who has the sovereignty to decide which task in the pool of ready tasks, to a given time, should be selected to be executed. All tasks in this model are periodic tasks.

Priority

In this present model of a real-time system every single task has an assigned priority. The concept of assigning a priority can help the scheduler to determine which of the ready-to-be-executed tasks, should be selected to be executed on the CPU. A real-time system can operate with either static

or dynamic priorities assigned to the tasks. Tasks with static priority are constant and will not change during the system's runtime, while tasks with dynamic priorities can change during the system's runtime. Scheduling algorithms not taking priorities into considerations performs poorly (7, p. 122). All tasks in this model are tasks with static priorities.

Preemption

Some real-time systems run in a non-preemptive manner. Although, in this model preemptions are allowed and a preemption occurs when a task is interrupted before it has finished executing and is replaced by another, generally higher priority, task. The scheduler should make a precaution, so that any preempted task will be resumed and finished before it's deadline to avoid a deadline miss.

Resources

More advanced and complex real-time systems support the possibility of controlling resources. In such systems every task can reserve an amount of various resources. Enabling a task to require other resources than the CPU, can lead to resource contentions. Resource conflicts can arise if one task has reserved a resource, which also is required by a higher priority task to finish execute. Suddenly, the likelihood of phenomenons known from the parallel- and concurrent systems theory are introduced, such as deadlocks, starvation, fairness and live locks. Undergraduate textbooks in real-time systems (7, p. 277) explain how to take advantage of Resource Access Control Protocols to avoid resource contentions and eliminate the newly introduced threads. The model here takes a more simplistic approach towards the management of resources. Every task requires only one single resource, the CPU, and the CPU has no upper limit of how many tasks can request an instance of the CPU. This limitation is necessary to focus on the stated problem, rather than introducing another complexity into the model.

Hard versus Soft Real-Time Systems

Real-time systems are split into two categories, hard- and soft real-time systems. Various definitions of these categories exist in the literature. One interpretation (3) is using the timing constraints to distinguish[1] between the categories, which will be used here. Thus, the main difference is that a hard real-time system takes all deadlines solemnly and even one single deadline miss results in failure of the system. While a soft real-time system is less stringent and can tolerate deadline misses, the exact rigidness depends on that specific soft real-time system.

[1] other measurements could be using the output results

The application of the system has a great impact to decide whether a system should be hard or soft. It is obvious that a system, which displays the temperature on a wall could be a soft real-time system, allowing minor delays in the system once in a while, both with regard to measurements and the final display of the actual temperature. While a flight is highly volatile towards delay in the break system, so the system has to absolutely guarantee the timing constraints established. To emphasize that DVS can be applied in systems, which requires strict timing constraints and be safety-critical, the present model is a hard real-time system.

2.1.2 Schedulability analysis

A schedulability analysis is important to decide if a set of tasks can be scheduled and, if so, how to schedule the tasks. The schedulability analysis is the core element in any scheduling algorithm. A weak schedulability analysis indicates a scheduling algorithm performing poorly. It is common to use a response-time analysis to determine the schedulability of a set of tasks. The idea in this analysis is to compute every task's worst-case response times and compare that with it's deadline. The worst-case response time is not necessary equal to a task's worst-case execution time, since response times take the waiting time when a task is preempted into account.

Static analysis

Static analysis is a simple way in advance to see whether a set of task can be scheduled. It requires information known a priori and the assumption of that the length of the tasks to be their worst-case execution time. Scheduling algorithms is divided into *offline* and *online* scheduling algorithms.

Offline scheduling algorithm

Offline scheduling algorithms use static analysis to schedule all tasks. There is no perception of past and future tasks. All tasks are scheduled according to their worst-case execution times and once they are scheduled, the work of the algorithm is done. The scheduler follows the schedule, if a task during runtime of the real-time system finishes earlier than it's WCET, the CPU does nothing and there will be a period of no activity on the CPU. This period is often referred to as *slack*. The slack will be the difference between the WCET and the actual execution time. The CPU will first begin execute again after the slack. Since the offline scheduling algorithm needs to have all tasks scheduled before the execution of the first task, the algorithm is often very difficult to develop. Typically, because the scheduling problem can be hard to solve.

Online scheduling algorithm

Online scheduling algorithms differ from offline scheduling algorithms, by making their decisions "on-the-fly", while the real-time system is running. Online scheduling algorithms thereby has a history of already executed tasks. However, they do not necessary know about the tasks in the future. Knowing the past and having an idea of which tasks are ready to be executed makes opportunities for the online scheduling algorithm to be more intelligent. The actual execution times become available for the scheduling algorithm, so it can make decisions which take advantage of the slack period, such that the response times of various task get reduced. In contrast to offline scheduling algorithms, online scheduling algorithms need to act fast and make the decision, which task is to be executed, available for the CPU as soon as possible to avoid any delay.

The real-time scheduling chosen in this real-time system model makes an online *priority-driven* approach. The most popular online algorithms in the literature are *Earliest Deadline First* (EDF) *Rate Monotonic* (RM). Both algorithms make use of CPU utilization and individual schedulability tests to guarantee if a given task set is schedulable.

Earliest Deadline First scheduling algorithm

EDF is a dynamic priority-driven scheduling algorithm. The basic strategy is to choose the task with the nearest deadline to the current time of scheduling point. One advantage of EDF is that it has a utilization bound of 100 %.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.1}$$

Where C_i is the WCET and T_i is the deadline for the task, i. If the computed utilization is below 1, EDF can guarantee the set of tasks can be scheduled on the CPU. EDF is not common in industry hard real-time systems due to multiple reasons. Instead the RM scheduling algorithm is widely used.

Rate Monotonic scheduling algorithm

RM is a static priority-driven scheduling algorithm. Like EDF, RM also has a schedulability test, which uses the CPU utilization. A subtle difference is that RM does *not* guarantee schedulability if the CPU utilization is below 100 %. There has been made work (7), which has proven that the schedulability with RM scheduling is dependent on the amount of tasks, which has

to be scheduled. Their work resulted in equation 2.2.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{2.2}$$

If the exact number of tasks is known, it is straightforward to estimate if the CPU utilization is below the bound made by (7). If so, RM guarantees that the set of task can be scheduled. Moreover, they also computed the limiting value of the equation when the amount of task is reaching infinite, which can be seen in equation 2.3.

$$\lim_{n\to\infty} n(\sqrt[n]{2} - 1) = \ln(2) \approx 0.69\ldots \tag{2.3}$$

As it can be seen if the CPU utilization is below the "worst-case" bound 0.69, RM will always guarantee schedulability, but there will be cases where the utilization can be above the bound and still can be scheduled by RM.

2.1.3 Offline Dynamic Voltage Scaling

As mentioned before, offline scheduling algorithms tend to be complex algorithms solving computational hard problems. It is especially troublesome that the problem grows exponential with the size of the input. Hence, offline algorithms make use of combinatorial optimization, to find an optimal scheduling out of many possible scheduling possibilities. Metaheuristics are also used, although these do not guarantee an optimal solution (scheduling), since metaheuristics algorithms are quicker and in many cases more feasible to use.

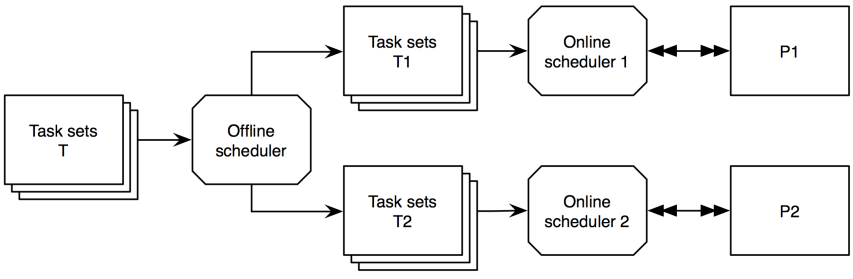


Figure 2.1: Offline and online scheduling interaction with the system

Even though this thesis do not address offline DVS scheduling, it is optimal to have preprocessed the input with an offline algorithm. Especially, in case where the system architecture has more than one processor at disposal. The complexity becomes very high and infeasible for an online algorithm to re-evaluate every decision during execution for the whole system. The

proposed solution is *delegation*. An offline algorithm could initially make the complex computations and find the most optimal mapping of tasks (and also operating frequency) to each processor in the system, and hereafter an online scheduler takes over for each of the processor. During execution the online scheduler continuously makes less complex computations to decide what the actual operating mode should be for the active task on the processor. The flow is shown in figure 2.1.

TABU-based search algorithm

A very interesting offline DVS algorithm is the one developed by (5). It is a TABU Search-based algorithm making use of metaheuristics. They have developed a *cost function*, which has multiple constraints such as energy, reliability and time. This cost function is then trying to be minimized when exploring solutions. One of the advantage of TABU search is that it allows selection of non-improving solutions, such that it can avoid returning solutions from a local optimum. Furthermore, addressing reliability has become a very hot topic recently, since it has been proven that the error of transient faults increases exponentially be reducing the voltage level of the processor. Mapping moves have been illustrated in figure 2.2, after the mapping the algorithm decides which level of frequency should be assigned.

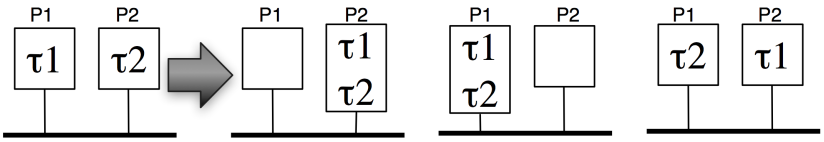


Figure 2.2: Mapping moves during offline scheduling

2.1.4 Online Dynamic Voltage Scaling

The main emphasis in this thesis is making it possible to implement online DVS algorithms into a simulation framework and be able to evaluate them. The first problem question in section 1.4.1 seeks existing online DVS algorithms. In the literature especially two algorithms turn up frequently and those algorithms are: *Low-Power Priority-Based Rate Monotonic* (LPP) and *Cycle-Conserving Rate Monotonic* (CCRM). These are chosen to be described further and are the ”test algorithms” to be applied in the simulation framework.

Low-Power Priority-Based Rate Monotonic algorithm

This LPP algorithm (10) actually consists of both an offline and online component, where the offline (WCET) analysis computes the lowest possible operating frequency all tasks can be run in and which still guarantees

all deadlines. The interesting part, however, is the online analysis which dynamically varies the operating frequency based on exploiting the slack emerged from execution time variations and idle intervals. The algorithm is briefly described with examples from the relevant literature.

	T_i	D_i	C_i	**Priority**
τ_1	50	50	10	1
τ_2	80	80	20	2
τ_3	100	100	40	3

Table 2.1: Task set example (10)

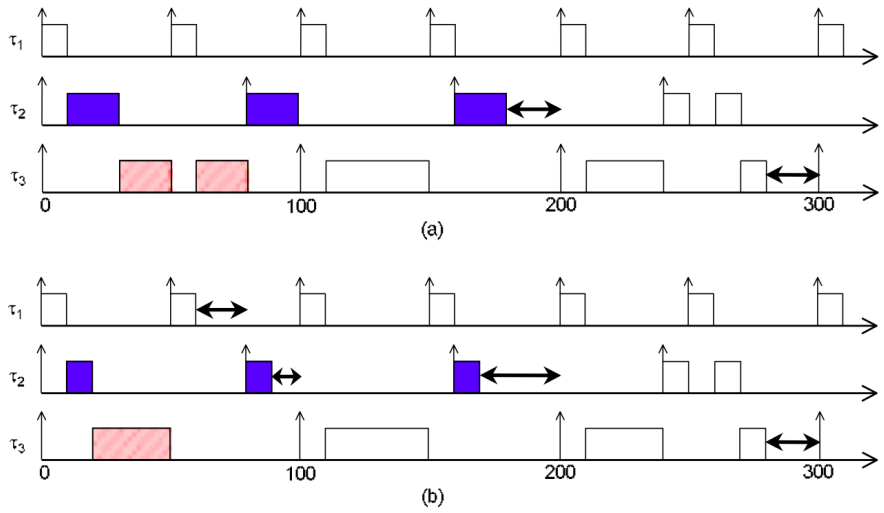


Figure 2.3: Scheduling (10)

LPP requires two queues, one *run queue* and one *delay queue*. The run queue keeps track of all tasks which are ready to run (released) and are ordered according to their priority. The delay queue keeps track of all tasks that have already run and finished executing, so the tasks are not ready to run - be released - before their individual next period occur. They are sorted according to when their release time is due. The task with the highest priority in the run queue is called the *active task*. An example from (10) is reproduced in figure 2.4 with the task from table 2.1.

Whenever the algorithm is invoked it checks the delay queue to see if any tasks are ready to be released, if so they are moved to the run queue. In the process it further checks whether any of the newly released tasks have a higher priority task to be run than the active task running, in that case, the active task is replaced with and this is called a *context switch*. Most

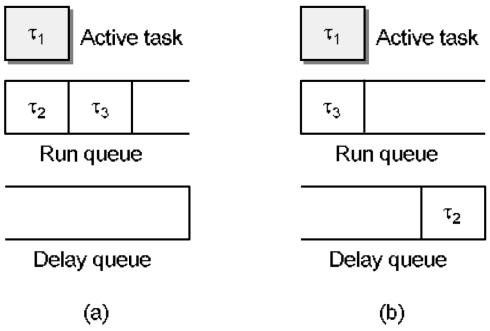


Figure 2.4: State of the queues (a) at time 0 and (b) at time 50

conventional schedulers follows this approach, but LPP get to be interesting when the run queue becomes empty. It divides the scenario up into two:

- All tasks have completed their executions for their respective periods, hence no active task exists either. The processor can then enter power down mode.
- All tasks except the active task has finished their executions for their respective periods. The processor can then prolong the active tasks execution time, by reducing the operating frequency, until the earliest released task of the delay queue.

This is the basic concept of the LPP algorithm, the pseudo code developed by (10) is listed in appendix A. The computation to reduce the active task's the operating frequency, in case it is the only task left outside the delay queue, is given in equation 2.4.

$$(t_a - t_c) \cdot r_{opt} + \frac{(1 - r_{opt})^2}{\rho} = C_i - E_i \tag{2.4}$$

Where C_i is the WCET for the active task, τ_i, and E_i is the already executed time of τ_i. The current time is denoted t_c and the next arrival time of the next task available in the delay queue is t_a. There is a delay of changing the operating frequency, this rate is denoted as ρ. Solving r_opt in equation 2.4 gives equation 2.5.

$$r_{opt} = \frac{-\rho \cdot (t_a - t_c) + 2 + \sqrt{\rho^2 \cdot (t_a - t_c)^2 - 4\rho \cdot (t_a - t_c - C_i + E_i)}}{2} \tag{2.5}$$

Nevertheless, r_{opt} can be computationally expensive to perform for the online scheduler. To overcome this issue (10) has developed a heuristic solution given by equation 2.6, where they neglect the above mentioned delay. The proof that the heuristic solution is an upper bound such that r_{heu} is always larger than r_{opt}, hence not violating the schedulability constraint is given in their appendix.

$$r_{heu} = \frac{C_i - E_i}{t_a - t_c} \tag{2.6}$$

Cycle-Conserving Rate Monotonic algorithm

CCRM is another online DVS algorithm and differs slightly from LPP, especially when it comes down to the slack distribution among tasks. The algorithm is presented in (8) and is briefly explained here.

	T_i	D_i	C_i	**Priority**
τ_1	8	8	3	1
τ_2	10	10	3	2
τ_3	14	14	1	3

Table 2.2: Task set example (8)

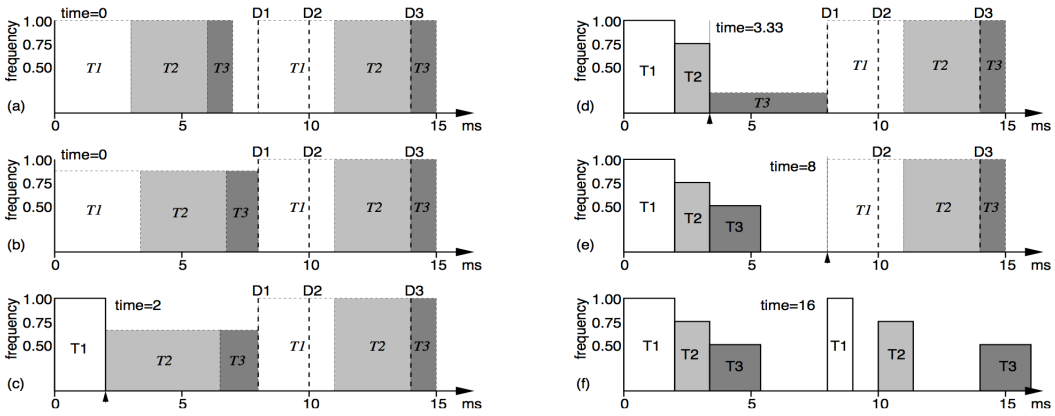


Figure 2.5: Scheduling (8)

The example in figure 2.5 (letters in parentheses corresponds to the letters in the figure) shows:

- (a) a scheduling of tasks and assignment of frequency made by static analysis. It can bee seen that there is an idle period just after the completion of τ_3, this is based on WCET. CCRM is different compared with LPP, since it tries to distribute the extra available computation

time not only to one (if enough slack is available), but all remaining tasks which are released and ready to run.

- (b) despite the fact that, the computed frequency to be run by the tasks is a decimal between 0.75 and 1.0 (b), CCRM is a discrete DVS algorithm. Moreover, it makes a pessimistic approach and chooses the available frequency level which is just above the computed frequency.

- (c) the actual execution is in progress. τ_1's frequency has been set to the maximum (since the exact computed frequency is not available in the set of operating modes). The WCET is 3 ms, but during execution the task has already finished executing after 2 ms. The online algorithm computes a new frequency, which lies between 0.5 and 0.75.

- (d) τ_2 has been set to the closest available "round-up" frequency level, 0.75, and has finished just after 1.33 ms. Hence, τ_3 can be prolonged to execute till the release of τ_1. But the available operating modes limit the possibility of taking full advantage of the total available slack.

- (e) τ_3 is assigned the lowest available frequency, 0.5. The unused slack is shown and τ_1 is released at 8 ms. However, the strategy of CCRM limits the algorithm further to not look beyond the next deadline in the system, so τ_1 is executed at highest possible frequency.

- (f) Shows the trace of actual executions and frequency levels of all tasks for 16 ms.

Beside a delay queue and a run/ready queue, CCRM makes use of multiple counters. Each task is equipped with two counters. One counter, c_left_i, is initialized to the remaining time, which initially would be equal to it's WCET. The other counter, d_i, indicates allocation and is initialized to zero. Whenever there is slack available the algorithm distributes this slack among tasks, which are in the run queue. More specifically, the slack added to a task's allocation is constrained by $d_i \leq c\_left_i$, and if more slack is available then the slack goes on to the next task in the run queue. This constrain ensures schedulability and the computation of the frequency is shown in equation 2.7, although the algorithm makes a slightly different approach, since it "rounds up" to the closest available operating frequency. The complete pseudo code can be found in appendix B.

$$f = \frac{d_1 + ... + dn}{max\_cycles\_until\_next\_deadline} \tag{2.7}$$

2.1.5 Dynamic Voltage Scaling in immobile devices

Up until now, DVS has been mentioned in mobile embedded devices. Even though the immediate advantages speak of DVS being most relevant in these devices, this is not entirely true. DVS has potential in immobile devices as well, even though these devices have unlimited access to resources and are not bound by energy constraints.

Governments and non-governmental organizations all over the world is fighting a moral war to reduce the power consumption. Indirectly, a reduction in power consumption can lead to reduction in carbon dioxide emissions and greenhouse gasses. Private and public companies spend a huge amount of money in buying and maintaining cooling systems, to reduce the temperature in various environments where embedded devices work. Even though this topic is not going to be followed further, both mentioned issues can be eliminated by introducing DVS in these immobile embedded devices. DVS causes potential reduction in power consumption and accomplishes this by reducing the operating frequency of the CPU. The released heat from the CPU is directly proportional to the operating frequency.

2.2 System model

The architecture of the relevant real-time system is described as a uniprocessor. This is only because this thesis isolates the CPU from the bigger real-time system. The bigger real-time system is taken from (5) and consists of a set $\mathcal{N}$ heterogeneous processors and interconnected by a communication channel. The relevant real-time system is then a subsystem and can be seen as one of the processors. The communication channel is a non-preemptive fixed-priority bus, but since the bus is only used whenever a task is transferred between the processors and the relevant real-time system only can take actions on the present processor the bus and energy consumption from the bus is neglected.

2.2.1 Operating mode

It is assumed that all processors supports DVS and a single processor, N_i, has a set of available operating modes. An operating mode is modeled in equation 2.8.

$$\Lambda_j^{N_i} = (f_j^{N_i}, v_j^{N_i}, p_j^{N_i}) \tag{2.8}$$

Where $f_j^{N_i}$ indicates the current operating frequency for the processor N_i in the operating mode j. The frequency is measured in Hertz (Hz), correspondingly, the supply voltage, $v_j^{N_i}$, is measured in Volts (V) and the power,

$p_j^{N_i}$, is measured in Watts (W). The normalized frequency and normalized voltage are computed in equation 2.9 and equation 2.10, respectively.

$$F_j^{N_i} = \frac{f_j^{N_i}}{f_{max}^{N_i}} \tag{2.9}$$

$$V_j^{N_i} = \frac{v_j^{N_i}}{v_{max}^{N_i}} \tag{2.10}$$

Switching between operating modes

Any dynamic switch in the operating modes of the CPU creates an overhead in both energy and time. This overhead should somehow be addressed and (5) suggests two matrices, X and Y. In matrix X an element, X_{jl} $(j \neq l)$, represents the time overhead required to switch from mode j to mode l. In matrix Y, an element, Y_{jl} $(j \neq l)$, represents the energy overhead required to switch from mode j to mode l. The representation make it possible for the overhead to be symmetrical, where the overhead is identical whether switching from mode j to mode l or vice versa. In addition, the matrix could be asymmetrical and have different overhead whether switching from mode j to mode l or from mode l to mode j.

2.3 Application model

The application consists of a set of periodic tasks, Γ. Every task, τ_i, in Γ is assigned a unique priority and it's deadline is smaller or equal to it's period, i.e., $D_i \leq T_i$. Although, the mapping of τ_i on to a specific N_j is not done at this level of the system. As earlier stated, the present real-time system model is a subsystem and running on a uniprocessor. Hence, the optimal mapping and assignment of frequency level is already done in a preprocessing module. This preprocessing module is a permanent part of the bigger real-time system and make use of the offline algorithm mentioned in section 2.1.3 and further explained in (5). So the tasks of one application could be distributed out to different CPUs, but it is not a requirement for the online scheduler at this system level to have an overview of the set of tasks.

2.3.1 Criticality

The application can indicate criticality on any given task, such that some tasks are critical and others are non-critical, for the application to successfully complete. To enforce this criticality and tolerate (transient) faults there are different solutions, two of which are: *replication* and *re-execution*.

Replication

Replication is when there is made extra copies of the exact same critical task. The application designer has to specify a desired *redundancy level*, k_i. The redundancy level will indicate the exact amount of replicated tasks of task τ_i will be introduced to the system. This solution increases the amount of tasks to be scheduled in the system. An assumption is that during execution whenever a critical task has completed successfully the remaining replicated tasks will be suspended, which will correspond to idle period or slack. Replication is already covered by (5) and an integrated part of the offline scheduler. That being so, the decisions regarding creating replicated tasks, assigning an optimal individual frequency level and the distribution of the replicated tasks to CPUs are made by the preprocessing module.

Re-execution

Instead of replicating critical tasks to ensure successful completion of an application, another approach is to re-execute any failed critical task. Obviously, re-execution solely makes sense in online scheduling, since it is only during execution it can be known that a critical task has failed and needs to be re-executed. Therefore, the responsibility of re-execution cannot be placed in the preprocessing module, but has to be placed at this level of the system.

2.4 Power model

A core and necessary feature of the system is to keep track of the energy consumed. A power model was proposed by (5), shown in equation 2.12, to compute the total energy consumption of all tasks in the custom set, Γ', within the hyperperiod, $T_{\Gamma'}$. Every task τ_i in Γ' does not need to be trunked from the same job, but what they have in common is that the schedulability analysis has chosen these tasks to be run on the same CPU, N_j, and in the same operating mode, l. The WCET of τ_i on N_j is $C_i^{N_j}$, assuming the processor will run in it's highest operating mode (maximum frequency) constantly. This will most likely not be the case since the whole purpose with the system is to take advantage of dynamically switching operating modes, so the WCET needs to be adjusted to the operating mode decided by the schedulability analysis.

The adjusted new WCET, c_i, is computed by using equation 2.11 and $F_l^{N_j}$ is the corresponding frequency to the decided operating mode, l, on the CPU.

$$c_i = \frac{C_i^{N_j}}{F_l^{N_j}} \tag{2.11}$$

$$E_S = \sum_{\tau_i \in \Gamma'} \left\lceil \frac{T_{\Gamma'}}{T_i} \right\rceil \cdot p_l^{N_j} \cdot c_i + O \tag{2.12}$$

The explained power model is very applicable in an offline schedulability analysis and the values derived from it can be characterized as an upper-bound for the energy consumption in the system. However, the very fact that the calculated energy consumption is an upper-bound makes room for further energy savings. The power model can be modified so it can be used in an online schedulability analysis, such that potential energy savings are exploited, using the gained knowledge of the actual execution time after each task has completed it's execution.

2.4.1 Online Power model

Instead of computing the expected energy consumption in advance the online power model computes the energy consumption ad hoc after each task has completed it's execution. At this point the exact execution time and frequency-level are known and the computation of energy consumption, shown in equation 2.13 [doublecheck notions], becomes fairly trivial and precise.

$$E_S = \sum_{\tau_i \in \Gamma''} p_l \cdot c_{\tau_i} + O \tag{2.13}$$

The main differences are: firstly, there is no reference to any processor due to this system works on a uniprocessor. Secondly, the execution time does need to be adjusted, since the exact execution time is known and the assigned frequency level, l, is obtained from the online schedulability analysis. Thirdly, the custom task set, Γ'', does not longer only contains all tasks with the same assigned frequency level, but the entire set of tasks which has run to on the processor.

2.5 Task Execution model

A simulation saves the application designer time, resources and the final evaluation gives a better understanding of how each individual task fits the real-time system. Since the tasks are not actually executed on a processor, the complexity of how to simulate the execution online emerges. This complexity is solved by introducing a task execution model, which predicts the actual execution based on a probabilistic distribution and parameters. The task execution model used in (11) is based on a *random gaussian distribution*, although, there is also presented an extra model which is based on a

negative exponential distribution [Explain short why this introduction of an extra model/distribution].

2.5.1 Random Gaussian distribution

The random gaussian distribution (normal distribution) is used widely in both analyzing data and in experiments. Often data from the real world is normal distributed and this claim is supported by the *central limit theorem*, which basically states the distribution of the mean will be distributed normally as the number of random independent observations gets sufficiently large.

The actual executions of the tasks can be modeled as behaving like normally distributed. Where a large group of tasks finishes their execution around an estimated mean, while tasks which either are finished almost instantly or near their WCET are seldom. The popularity of the gaussian distribution is also caused by being fairly straightforward to understand, implement and sample from. As it can be observed in 2.14, the distribution can be made unambiguously by a mean, μ, and a variance, σ^2, leading to the notation: $\mathcal{N}(\mu, \sigma^2)$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.14}$$

The characteristics about the gaussian distribution are that it tries to center around the mean and is symmetrical having no skewing. However, in experiments the practicality of the distribution depends highly on the two estimated parameters, mean and variance.

Estimation of parameters

The authors from (11) have not observed actual execution times of tasks for applications run on processors, so no statistics of actual execution time is available for them. Instead they define mean with a variable, Best-Case Execution Time (BCET), which can be seen in equation 2.15. BCET is the time a task in best-case completes, so saying a task can never finish earlier than it's BCET. During their experiments they vary BCET, and thereby the sampled actual execution time, to analyze the significance of increased or decreased actual execution times in accordance with energy consumption.

$$\mu = \frac{BCET + WCET}{2} \tag{2.15}$$

It is known from probability theory that the *standard deviation* of a probability distribution is the square root of it's variance. The standard deviation,

which determines the spread around the mean, can also explain how many observations (actual executions) are guaranteed to exist within a certain area of a distribution.

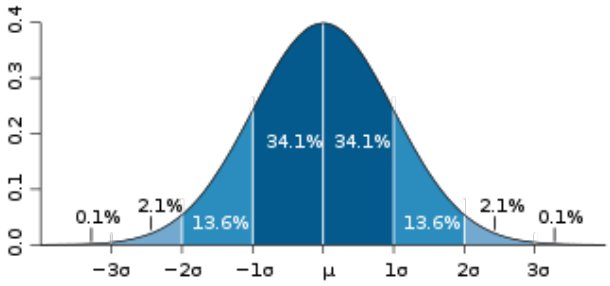


Figure 2.6: Gaussian distribution with standard deviation [source: wikipedia.org]

Figure 2.6 illustrates a gaussian distribution, where the standard deviations are marked. The percentages indicate the probability, that a random task takes on a value (actual execution time) in that particular interval. Reasoning is used by (11) to determine the standard deviation and the variance, respectively, σ and σ^2. They argue that no task can execute longer than WCET and there is a probability of 99.69 %, that any random task has an actual execution time which lies within the interval $[\mu - 3\sigma; \mu + 3\sigma]$.

$$\begin{aligned} \mu + 3\sigma &= WCET \\ \frac{BCET + WCET}{2} + 3\sigma &= WCET \\ \sigma &= \frac{WCET - BCET}{6} \end{aligned} \tag{2.16}$$

During online execution the individual task's actual execution is then sampled from a gaussian distribution, $\mathcal{N}(\frac{BCET+WCET}{2}, (\frac{WCET-BCET}{6})^2)$.

2.5.2 Negative Exponential distribution

An extra task execution model is proposed in this thesis, the negative exponential distribution. This exponential distribution deviates from the gaussian distribution by having a completely different shape.

In practice this distribution is often used to model the time interval between random events. The distribution can be seen in equation 2.17.

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases} \tag{2.17}$$

However, the sampling from the negative exponential distribution is a bit

more tricky. The sampling method described here make use of the cumulative distribution function (CDF), shown in equation 2.18 and initially computes a maximal CDF value, $maxCDF$, by solving x with WCET, shown in equation 2.19.

$$F(x;\lambda) = \begin{cases} 1 - e^{-\lambda x}, & x \geq 0, \\ 0, & x < 0. \end{cases} \tag{2.18}$$

$$maxCDF = 1 - e^{-\lambda \cdot WCET} \tag{2.19}$$

Equation 2.19 describes the probability that a task will finish execute either *before or at latest* at WCET and is noted $maxCDF$. Hereafter, a uniform distribution is used to sample a random number, r, from zero to $maxCDF$. This r is then used in the inverted CDF to compute the actual execution time, x, seen in equation 2.20.

$$x = \frac{-ln(1-r)}{\lambda} \tag{2.20}$$

Estimation of parameters

The core parameter here is λ. The parameter can be estimated according to the application, since there is no doubt that the pattern of an actual execution time of a task is application dependent. In the present context, it is believed that the tasks execute close to their WCET. A very subjective estimation of λ is seen in equation 2.21.

$$\lambda = WCET \cdot 3/4 \tag{2.21}$$

$$\mu = \lambda^{-1} \Rightarrow \mu = (WCET \cdot 3/4)^{-1} \tag{2.22}$$

2.5.3 Uniform distribution

The last task execution model, which is introduced in this thesis is making use of an uniform distribution. In practice, a hypothesis or observations from actual task executions would be used to estimate the parameters in the above-mentioned two task execution models. In this model there is no extra parameters than the interval within the sampling should take place. It could be discussed, whether the model is realistic, but the purpose of this model is to get a view of how the strategies work on tasks which has equal probabilities of execution time between BCET to WCET. So the model could rather be applied to test various strategies. The sampling is relatively straightforward and the interval is defined as $x \in [BCET; WCET]$.

All three task execution models make use of a probability distribution to sample the online actual execution time for each task. However, there is a subtle difference between the models. The model based on the gaussian distribution tend to output actual execution times centering around the middle/mean, while the model based on the negative exponential distribution tend to output them close to the task's WCET. The uniform distribution do not require any parameters to bias the distribution in any direction or skewness.

2.6 Software model

The software models are explained in a conceptual architectural manner, and both entities and relations are illustrated in figure 2.7. The simulation framework encapsulates five units and those are *specification, scheduler, simulation engine and output*. One of the main goals is to make the framework flexible, such that all units can be replaced with a custom one, except the simulation engine. This flexibility is enforced, so it becomes easy for a developer or researcher to develop and prototype various scenarios and strategies. This section could also be characterized as the structural requirements for the simulation framework.

2.6.1 Specifications

The specifications should be seen as input to the simulation framework and is split into task specification and machine specification. While the task specification is static input, the machine specifications are dynamic input. The simulator engine will dynamically interact with the machine specifications throughout the simulation.

Task specification

The task specification contains the set of tasks, which are to be scheduled and evaluated through the simulator. The specification describes each task in detail and contains the parameters shown and explained in table 2.3. The reason to make task specifications interchangeable is to provide the developer the possibility of creating task sets with certain characteristics. These task sets can hereafter be evaluated and give an impression how well the algorithm has performed. The set of tasks, $\Gamma = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$, is stored in a data-file, which is then parsed into the simulation engine as input.

Parameter	Description
Taskname	Identifier of τ_i
WCET	Worst Case Execution Time of τ_i
BCET	Best Case Execution Time of τ_i
P_i	Period of τ_i
D_i	Deadline of τ_i
ρ_i	Priority of τ_i
δ_i	The arrival offset of τ_i
f_i	Frequency of τ_i

Table 2.3: Task specification for the task, τ_i

Machine specification

The machine specification describes the hardware model, namely, the processor architecture. Basically, the machine specification is an operationalization of the system model described in section 2.2. It contains a set of processors, N, where each processor has a set of operating modes, Λ^{N_i}, and a single operating mode is described in the system model and equation 2.8. Like the task specification, the purpose of this specification is to give the developer more control and flexibility. Especially, it could be interesting to analyze how well an algorithm performs on various different processors with different parameters such as number of operating modes and their energy consumption.

Parameter	Description
Processor name	Identifier for N_i
f_{max}	Maximum frequency of N_i
scalingPoints	Set of operating modes, Λ^{N_i}, of N_i
scalingFactor	Scaling factor of N_i

Table 2.4: Machine specification for a single processor, N_i

2.6.2 Energy consumption

This unit of the simulation framework operationalizes the power model from section 2.4. In the literature researchers have various ways of estimating the consumption of power and energy. It is essential not to make this unit a static part of the simulation framework, since then it is possible to adapt foreign power models and the consequences can be observed directly.

2.6.3 Task execution

The task execution part could be one of the three different task execution models, which were introduced in section 2.5. Each of them do exactly the same, announce the *actual* execution time of a task, but differ in the way of computing it.

2.6.4 Scheduler

The primary objective for a scheduler is to react upon requests from the simulator engine. It is implemented based on an online DVS algorithm. Multiple schedulers could be developed and the motivation is to make the developer rapidly prototype ideas and strategies and compare the schedulers.

2.6.5 Simulator engine

The simulator engine should not only take actions based on inputs, but also act like an ”umpire” overseeing all impacting actions on the scheduling of the tasks. Hence, the engine needs overall overview and should enforce the virtual laws, e.g. take action whenever a deadline miss occur. In contrast to all the other units of the simulation framework there exist only *one* simulator engine.

2.6.6 Output

The simulation engine has the possibility of passing raw information it collects during execution to an output unit. This output unit then can process the information and show it in a graphical presentation for the developer.

2.7 Design

The analysis of the software model has so far illustrated the conceptual architecture. The purpose of making a conceptual architecture is to explain the description of the system in a more abstract way and highlight the the ideas and concepts of the simulation framework. Nevertheless, the conceptual architecture cannot stand alone in an analysis. To fully cover the usability of the simulation framework the analysis has to be supplemented with requirements and use cases.

2.7.1 Requirements

The requirements explained here are functional requirements and non-functional requirements. These requirements are defined to ensure the needs of developing an online DVS algorithm and that it is possible to evaluate in the simulation framework.

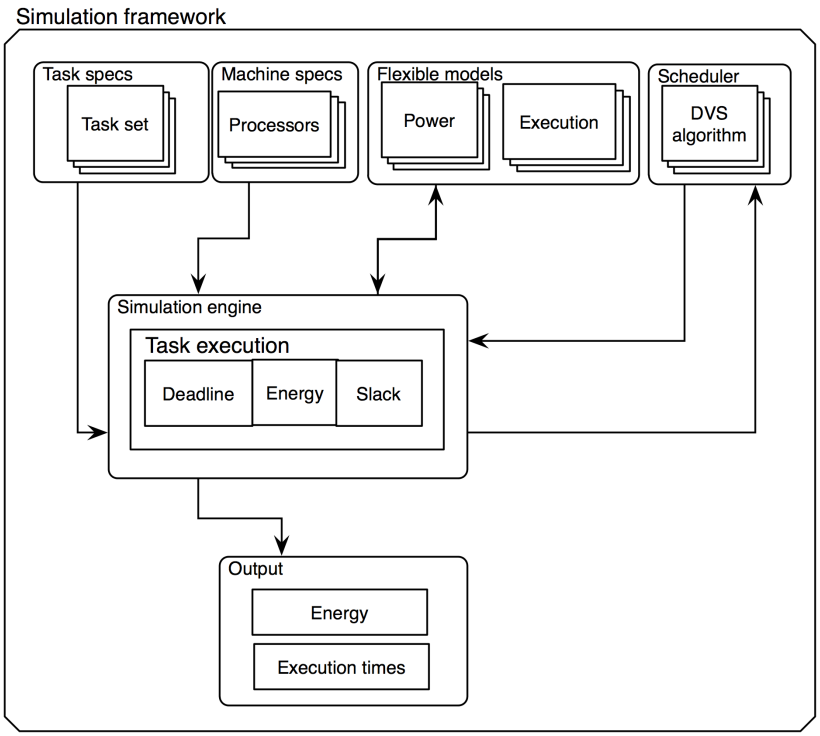


Figure 2.7: Conceptual software architecture of the simulation framework

Functional requirements

The functional requirements demand which specific functionalities are to be expected in the simulation framework and what it actually can do. They are shown in table 2.5.

Non-functional requirements

The non-functional requirements concerns the performance of the simulation framework and constraints on the system. They are shown in table 2.6.

2.7.2 Use cases

It is a good practice to develop use cases before an implementation of the software model. In the process of defining a use case, multiple scenarios are identified, which then are tied together by a common user goal (4). Although, having identified important scenarios does not directly lead to a use case, since there is no format in creating the optimal use case or collaborating use case diagrams (2). Which has resulted in many variants of use cases. A suggested use case template made by (2), explained in table 2.7, is used as the format for creating use cases here.

Name	Description
Task set	Task set can be defined by developer and used as input
Machine specs	System architecture can be defined by developer and be used as input
Models	Power- execution distribution models can be chosen (build-in) or implemented by the developer
Scheduler	A scheduler can be loaded into the framework or build-in schedulers can be chosen
Multiple schedulers	It should be possible to choose multiple schedulers to be run
Visualization	The results should be visualized inside the framework

Table 2.5: Functional requirements of the simulation framework

Name	Description
Look and feel	The look and feel should be intuitive and simple
Multi-platform	The framework should be able to run on cross platform and the GUI should not vary
Waiting time	There should not be unnecessary waiting time
Text	The messages inside the framework should be understandable

Table 2.6: Non-functional requirements of the simulation framework

Use Case	*Use case identifier and reference number and modification history*
Description	*Goal to be achieved by use case and sources for requirement*
Actors	*List of actors involved in use case*
Assumptions	*Conditions that must be true for use case to terminate successfully*
Steps	*Interactions between actors and system that are necessary to achieve goal*
Variations *(Optional)*	*Any variations in the steps of a use case*
Non-functional *(Optional)*	*List of non-functional requirements that the use case must meet*
Issues	*List of issues that remain to be resolved*

Table 2.7: Use case template

Use Case	Evaluation of two online algorithms (LPP and CCRM)
Description	An evaluation of those two algorithms together with a baseline scheduler (RMS)
Actors	Developer, researcher, student
Assumptions	The algorithms should have been implemented or given by the actor
Steps	The actor should first decide which models should be used in the evaluation, the next step is to actually run the simulation
Issues	The task set is not schedulable, since the pre-processing phase has failed

Table 2.8: Evaluation of two online algorithms

Chapter 3

Implementation

The implementation of the simulation framework is made in Java and described in the upcoming sections. Emphasis in the description of the framework has not been in-depth technical details and line-by-line specifics. It is rather design patterns and significant implementation concepts, which are brought forward.

3.1 Simulation engine

The simulation engine is actually a group of the basic objects:

- `BasicTask:` This is the basic task model having the minimum level of parameters, which all schedulers can use. However, a scheduler can derive from this task, e.g. create an `ExtendedTask`, if it needs further parameters to support the developing algorithm.
- `ScalingPoint:` The scaling point is a single operating mode of a DVS processor.
- `Processor:` The processor, which has a list of all available scaling points.
- `Execution:` This object contains the specific timings, when a task starts executing and has finished or preempted.
- `Job:` Containing the primary task and the list of executions.
- `Simulator:` The essential object within the engine-package. Holds all necessary information to act as an neutral umpire. All simulation runs are executed by the `run` function in this object.

3.2 Schedulers

The three schedulers which have been implemented are: `EventCCRMScheduler`, `EventLPPcheduler` and `EventRMScheduler`. They have been earlier described in-detail by going through their individual algorithm in section 2.1.4. Moreover, CCRM and LPP have also been the given the opportunity to be run in a continuous frequency-level. Instead of going further into the implementation-details, the more important part is how to implement a scheduler.

3.2.1 Interface

The solution of making the schedulers replaceable is based on *interfaces*. The interface for a scheduler is shown in listing 3.1. So when developing a scheduler, the scheduler has to implement the interface and all the methods listed in the interface. Since the engine has list of all tasks and knows in advance when they are released and when each task has it's deadline it will during execution tell the scheduler the current time and which task has been released. Then it is the scheduler's responsibility to give a reply back with the intended task to be run. The engine will react upon deadline-misses. If any should arise the scheduler is notified and the run is terminated. The actual replacement is done by runtime class loading.

Listing 3.1: Interface for a scheduler

```
1  public Task taskReleased(Task releasedTask, Task executedTask, ...
       double currentTime, double deltaTime);
2  public void invokeDeadlineMiss(HashSet<Task> deadlineMisses);
3  public void invokeInvalidAssignedExecutionTime(Task ...
       executedTask);
4
5  public Task taskFailed(Task failedTask, double currentTime, ...
       double deltaTime);
6  public Task taskFinished(Task finishedTask, double currentTime,...
        double deltaTime);
7
8  public void isDiscrete(boolean isDiscrete);
9  public boolean isDiscrete();
10
11 public Task timedPreemption(Task executedTask, double ...
       currentTime, double deltaTime);
12
13 public void setLayout(IGui gui);
14 public void setExecutionDistribution(IExecutionDistribution ...
       executionDistribution);
15 public void addHandler(Handler handler);
16 public CubicSpline getPowerModel();
17 public void run();
```

3.3 Replaceable design pattern

The same approach of making scheduler's replaceable has been used when handling execution distributions models and power models. If the developer wants to modify or create a completely new model, the interfaces must be implemented and followed.

3.4 Making use of external libraries

In the process of developing the simulation framework multiple third party libraries have been used. Worth to mention is the `jgoodies`-library, which has provided the look and feel and animation-functionality. The `gson`-library has been used to develop the parser of the input specifications, while the *CubicSpline* functionality is brought in by `flanagan`-library.

3.5 Parser

The inputs, both task- and machine specification, are read into the simulation framework through a parser. This parser uses the `gson`-library and avoids several middle steps, compared to a traditional parser, by reading from a JSON-format file and directly using a class. An example is shown in 3.2, where a list of processors is created by iterating over a machine specification file.

Listing 3.2: Parsing a machine specification directly into a list of objects

```
1 while(scanner.hasNext()) {
2         processors.add(gson.fromJson(scannner.nextLine(),
3                                                 Processor.class));
4 }
```

3.6 Interpolation

The `flanagan`-library has been of great help, when interpolating between the scaling points (operating modes). During the continuous frequency level, the precise power consumption cannot be read from the machine specification. Thus, there has been made an interpolation of all the operating modes with a curve fitting called CubicSpline. The library provides a very straightforward way of ”sampling” an arbitrary power consumption given a set of finite operating modes.

3.7 Graphical User Interface

The Graphical User Interface (GUI) has been developed using the `jgoodies`-library, mainly because of the look and feel, which ensures the same representation across platforms. Additionally, it enables parallel animations through a build-in thread-based mini-framework. After each simulation run, the engine feeds the library with the details of a run, after all simulation runs have finished the simulation initializes a visualization of a run from each scheduler. An example can be seen in figure 4.6 and 4.7.

3.8 Step-by-Step-based vs. Event-based

Initially, the simulation framework was build upon a step-by-step design. So a clock was build and it incremented ”time” with a time unit of one. The engine would then ”wake-up” every time-unit and orientate itself (which tasks have been released, any deadlines missed, etc.) and ask for decisions from the scheduler. However, the way it was implemented required that all executions should be discrete and that was an assumption, which was too drastic. Hence, another approach was tried, namely, the event-based. The main difference is that the engine can ”sleep” till an event occurs, and this event can occur in the continuous range of time. Roughly compared with SOA-architecture, it could be said that the framework went from using ”pull”-mechanism (fixed delay to wake up) to ”push”-mechanism (arbitrary delay to wake up).

Chapter 4

Experiments

The simulation framework is a product of this thesis, even so, the practicality of the product is difficult to determine. To overcome this, the simulation framework has been taken into use in this chapter by running experiments and obtaining results. This chapter will first introduce how the experiments will be run and then visualize the results.

4.1 Scenario

The scenario which should be imagined is the two online DVS algorithms mentioned in section 2.1.4 has been understood and implemented by following the interfaces from section 3.2. Intuitively, and by looking at the pseudo-code, it seems like CCRM is slightly more advanced, which should most likely have an impact on the energy-savings during execution and evaluation. To have a baseline, there has also been implemented a simple static Rate-Monotonic Scheduling algorithm (RMS), which does no dynamic changes regarding the operating frequency. RMS will schedule the task according to the priority and in case a task finishes before it's WCET, the scheduler will execute the next available task, if none then it goes into an idle period. Since the power model used do not consume energy during idle period, this will correspond to a power down mode.

4.1.1 Models and parameters

Operating mode	Frequency (Mhz)	Voltage (V)	Power (W)
1	1000.0	1.6	25.0
2	666.0	1.4	12.0
3	334.0	1.2	4.0

Table 4.1: Specification of DVS processor

The parameters have been chosen to be simple and clear. The following models should be defined before the experiment can begin:

- **Task set:** As task set the previous set, shown when explaining the LPP algorithm, is used and can be found in table 2.1.
- **Machine specification:** The machine specification is a DVS processor with operating modes specified in table 4.1.

- **Power model:** The online power model is mentioned in section 2.4.1 and has been implemented as a standard model in the simulation framework.

- **Execution distribution:** The execution distribution used in this scenario is the negative exponential distribution, explained in section 2.5.2.

- **Scheduler:** The three schedulers, which have been implemented and are going to be tested, are: RMS, LPP and CCRM.

4.1.2 Execution

To evaluate the schedulers, multiple simulations are run for each scheduler. 250 runs have been observed and the average power consumption has been computed for each scheduler. For practical reasons, the execution time has been sat to totally 300, and even though the time unit do not need to be specified, then to get a sensible unit from the power model, the time unit is defined as milliseconds (ms). It is worthwhile mentioning that idle intervals (NOP) in this system do not consume any energy, while in practice this has been measured to be approximately 20 % of a typical instruction (1). Hence, the comparison would be very conservative, especially RMS which will have the most idle periods during execution, since the scheduler is not taking advantage of extra slack and idle periods. This will be further discussed in chapter 5.

4.1.3 Results

The output from the simulation framework has been processed further in `SAS JMP`, since it has strong statistical analysis functionalities. Only the mean value has been taken into this thesis. The output is shown from figure 4.1 to 4.5.

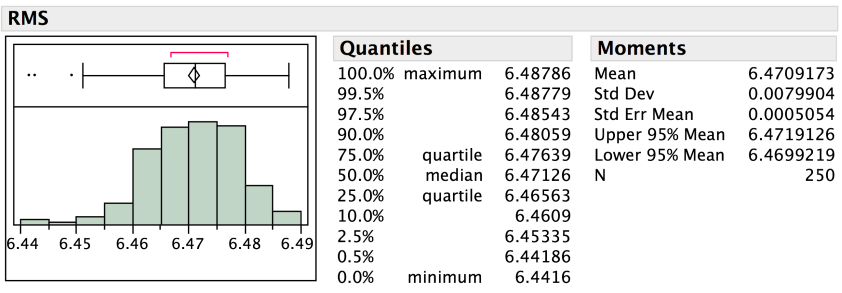
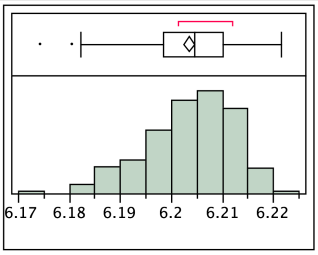


Figure 4.1: The result of RMS

LPP discrete

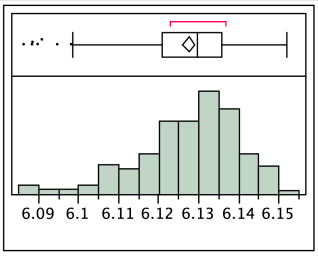


Quantiles		
100.0%	maximum	6.22153
99.5%		6.22111
97.5%		6.2167
90.0%		6.21377
75.0%	quartile	6.21003
50.0%	median	6.20464
25.0%	quartile	6.19842
10.0%		6.19192
2.5%		6.18491
0.5%		6.17593
0.0%	minimum	6.17434

Moments	
Mean	6.203579
Std Dev	0.0084463
Std Err Mean	0.0005342
Upper 95% Mean	6.2046311
Lower 95% Mean	6.2025269
N	250

Figure 4.2: The result of LPP discrete

LPP continuous

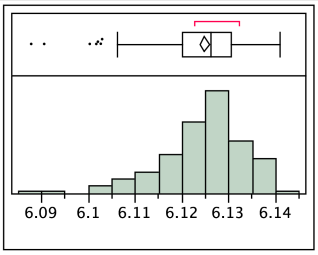


Quantiles		
100.0%	maximum	6.15187
99.5%		6.15116
97.5%		6.14663
90.0%		6.14177
75.0%	quartile	6.13588
50.0%	median	6.12961
25.0%	quartile	6.1209
10.0%		6.10944
2.5%		6.09578
0.5%		6.08697
0.0%	minimum	6.08643

Moments	
Mean	6.1275303
Std Dev	0.0122394
Std Err Mean	0.0007741
Upper 95% Mean	6.1290549
Lower 95% Mean	6.1260057
N	250

Figure 4.3: The result of LPP continuous

CCRM discrete

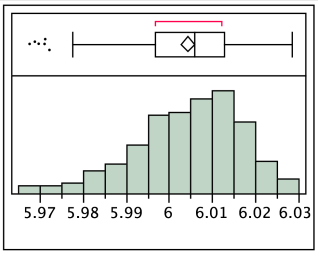


Quantiles		
100.0%	maximum	6.1408
99.5%		6.14042
97.5%		6.13788
90.0%		6.13518
75.0%	quartile	6.13053
50.0%	median	6.1263
25.0%	quartile	6.12011
10.0%		6.11357
2.5%		6.10281
0.5%		6.08856
0.0%	minimum	6.08785

Moments	
Mean	6.1248184
Std Dev	0.0086367
Std Err Mean	0.0005462
Upper 95% Mean	6.1258943
Lower 95% Mean	6.1237426
N	250

Figure 4.4: The result of CCRM discrete

CCRM continuous



Quantiles		
100.0%	maximum	6.02846
99.5%		6.02816
97.5%		6.02513
90.0%		6.01917
75.0%	quartile	6.01287
50.0%	median	6.00589
25.0%	quartile	5.99683
10.0%		5.98875
2.5%		5.97373
0.5%		5.96803
0.0%	minimum	5.96772

Moments	
Mean	6.0043453
Std Dev	0.0121505
Std Err Mean	0.0007685
Upper 95% Mean	6.0058588
Lower 95% Mean	6.0028317
N	250

Figure 4.5: The result of CCRM continuous

4.1.4 Comparison

The mean values obtained from the `SAS JMP` output and the relative deviation from the baseline (RMS) are shown in table 4.2. The relative deviation is computed by using equation 4.1.

$$baseline\_dev\% = \frac{non\_baseline\_mean\_value - baseline\_mean\_value}{baseline\_mean\_value} \cdot 100\% \tag{4.1}$$

Scheduler	Mean value (J)	Baseline deviation
RMS (baseline)	6.47	-
LPP (discrete)	6.20	-4.17 %
LPP (continuous)	6.13	-5.26 %
CCRM (discrete)	6.12	-5.41 %
CCRM (continuous)	6.00	-7.26 %

Table 4.2: Comparison of scheduling output

4.1.5 Simulation framework output

In figure 4.6 a single run is visualized in the simulation framework. This figure only illustrates the discrete version of CCRM and LPP and a regular version of RMS, while figure 4.7 illustrates the continuous version of CCRM and LPP along with regular version of RMS. The listing of schedulers are mentioned in the caption of each figure.

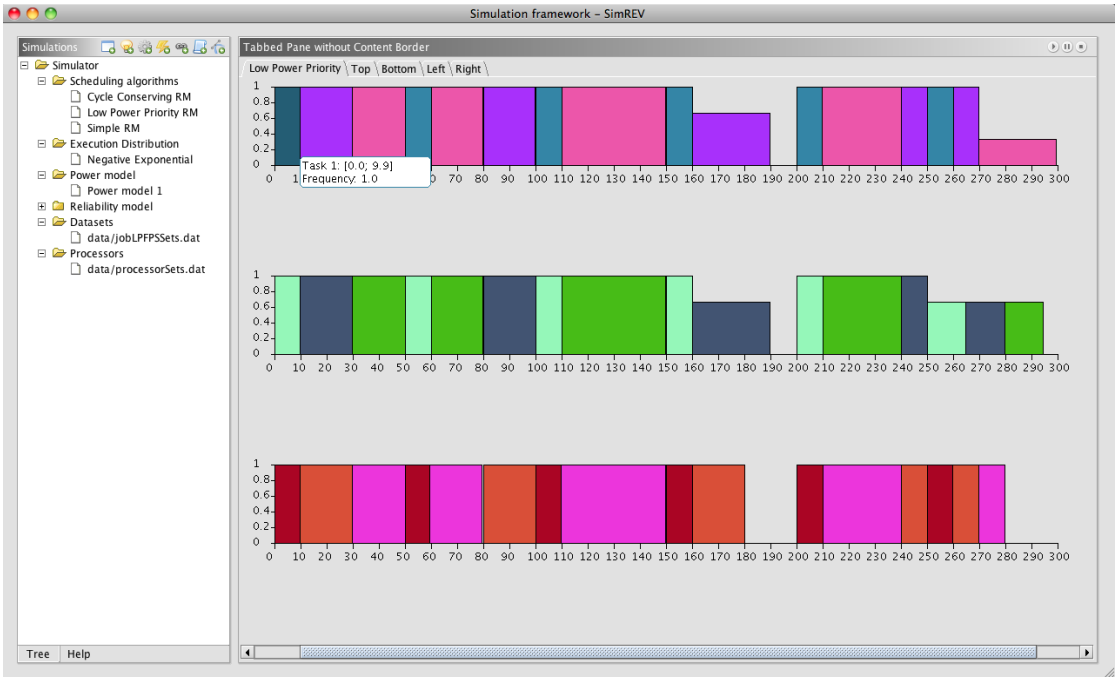


Figure 4.6: Single run visualized in the simulation framework - order is top: LPP (discrete); middle: CCRM (discrete); bottom: RMS

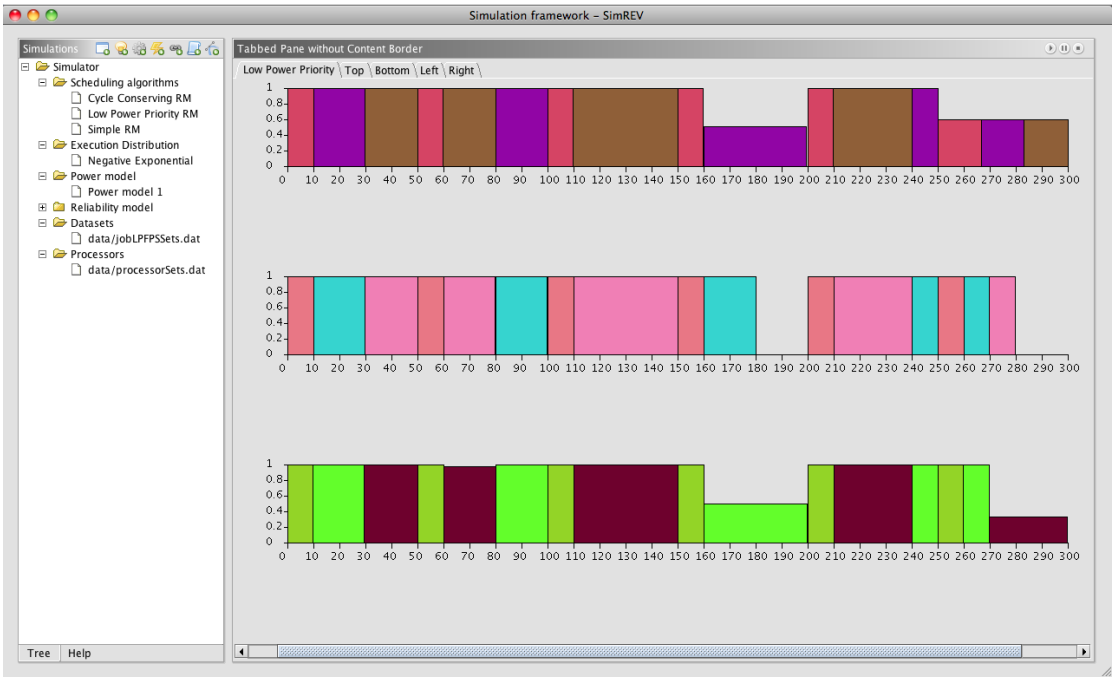


Figure 4.7: Single run visualized in the simulation framework - order is top: CCRM (continuous); middle: RMS; bottom: LPP (continuous)

Chapter 5

Discussion

This chapter will not only discuss the results obtained in the previous chapter, but also discuss the importance of a unified simulation framework. Short-term and long-term future works are pointed out.

5.1 Evaluation

Based on the output and the comparison made in table 4.2 it shows that DVS is absolutely a worthwhile investment instead of implementing a regular RMS. As indicated in the analysis, CCRM is a slightly more advanced algorithm than LPP, but spending time to build a more advanced algorithm can lead to further energy savings. CCRM makes a further reduction in energy than LPP. Another important observation is that the schedulers using continuous frequency levels actually make a significant energy saving compared to the discrete frequency levels, this yields for processors with more frequency levels or an actual processor with 100 % support for continuous operating modes.

Viewing the visualization provided by the simulation framework, it is noted how CCRM, LPP and RMS distinguish from each other and the algorithms individual characteristics are further clarified. E.g. CCRM in figure 4.6 at time 250, where CCRM distributes the slack multiple tasks, while LPP only distributes it when their is one (active) task left - RMS does nothing intelligent at all. This kind of visual analysis is very valuable, when working with more advanced algorithms. In figure 4.7 the idle periods from 4.6 do not exist, since they are absorbed by the continuous algorithms. The scope of this thesis is not to simulate multiple scenarios and definitively conclude whether LPP or CCRM is the most efficient algorithm, it is to provide the tool to make it possible to make the final evaluation. This has been demonstrated in a small scale and by using the simulation framework more, the evaluation can be extensive.

5.2 Simulation framework

There has only been simulated one single scenario, although, the practicality of the framework has been proven. The result produced by the simulator can be part of a statistical analysis (confidence intervals), by using it e.g. in `SAS JMP`. Especially, the visualization can make it easier for the developer to comprehend various strategies designed in the algorithm. Deciding to

go from a step-by-step-based to an event-based framework has definitely shown to be an advantage. Assuming all tasks to execute in discrete time would have been too unrealistic and less energy savings would have been demonstrated.

5.3 Importance

It is of great importance to be able to re-produce other researchers results, since the academic world relies on this principle. Multiple articles claim their algorithms have the best performance, but in many cases their results are enclosed in simulators, which are not accessible. Even if they are, it is very difficult to extend the simulator to compare their algorithm with another one, since the simulator has been designed and ”hardcoded” to their specific purpose and algorithm. This simulation framework breaks with that tradition and focus on being open, extensible and flexible.

5.4 Future work

Future work tend to be a long of list to-do’s, where ”nice-to-have”-features appears. Here, concrete suggestions are presented, which has been split into short-term and long-term goals.

5.4.1 Two-level frequency

An extended version of the LPP and CCRM scheduler was implemented supporting what could be characterized as ”dithering”, so a task switches between two-levels of available frequencies to simulate the optimal computed frequency. However, these schedulers were successfully simulating in the framework, the validity of the algorithms have not been proven and therefore not completed. To validate and verify the correctness of these extended algorithms would be a short-term future work. To support two-level frequency the engine has been extended with what would be characterized as an *interruption*. So a scheduler can plan an interruption during execution, which highly reflects real-world application.

5.4.2 Reliability

As mentioned in section 2.1.3 the reliability of a real-time system has become a very important subject, since embedded systems also needs to be fault-tolerant. The foundation to extend LPP or CCRM with this reliability constraint and evaluate the trade-offs in the framework has been established. This extension could be a short term goal.

5.4.3 Preprocessing/offline-module

A long term goal would be to extend the simulation framework to enable preprocessing of tasks with regard to the offline algorithm mentioned in 2.1.3 (TABU search-based algorithm). Where the mapping of tasks to processor and operating mode is done before the actual online simulation run, but within the framework.

Chapter 6

Conclusion

In this thesis an extensible simulation framework has been developed, which purpose is to evaluate online DVS algorithms. The framework is both extensible, but also flexible. The algorithm-developer can setup an environment based on the actual application, system model and assumptions, such as power and actual execution. The framework provides output in both numeric form and in a visual form (histograms). As default the framework offers two build-in online DVS algorithms (LPP and CRRM), a power model and three execution distribution models.

There has been made evaluation of the above-mentioned online DVS algorithms together with a baseline (RMS). This evaluation has shown that the total energy consumption deviates relatively 4.17 % (LPP) and 5.41 % (CCRM) from the baseline, when only using (discrete) the available operating modes on the DVS processor. If a utopian CPU is introduced, which can operate at arbitrary frequency levels the relative deviation can go up to 5.26 % (LPP) and 7.26 & (CCRM). It should be noted that the scenario simulated only was with three tasks and a short simulation period of 300 ms, although, it is not the scope of this thesis to do a comprehensive evaluation of the two online DVS algorithms with multiple scenarios - or ”play around” with the various parameters, which the models allow. It is rather to show that the evaluation can be done and conclusions can be drawn from the results.

Before developing the framework, an analysis of the algorithms had to be made to understand which requirements should be created to satisfy an extensible and flexible simulation framework. For which reason, the architecture and requirements are defined in chapter 2.

To take the simulator to the next level, there had been developed extended algorithms and initial thoughts about introducing a new constraint into the system, the reliability. The extended algorithms can be run, but the results indicate they have not implemented the interruption-feature of the engine correctly. Due to time constraints a solution to the extended algorithms was not found, and the reliability constraint was neither investigated further.

It is believed the framework is highly applicable, when developing more complex algorithms, where it is essential to comprehend the execution and analyze the numeric results. The framework can be used to accept, reject or modify strategies. Hence, it is a powerful tool for researchers, developers and educational purposes.

Bibliography

[1] T. Burd and R. Brodersen. Processor design for portable systems. *Journal of VLSI Signal Processing*, 1996.

[2] Derek Coleman. A use case template: draft for discussion. http://www.bredemeyer.com/pdf_files/use_case.pdf, June 1998.

[3] Jim Cooling. *Software Engineering for Real-Time Systems*. Pearson Education, 2002.

[4] Martin Fowler. *UML Distilled*. Pearson Education, 2000.

[5] Junhe Gan, Flavius Gruian, Paul Pop, and Jan Madsen. Energy/Reliability Trade-offs in Fault-Tolerant Event-Triggered Distributed Embedded Systems. *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, 2011.

[6] IEEE. Embedded Systems and the Year 2000 Problem: Guidance Notes., IEEE Technical Guidelines. *IEEE*, 1997.

[7] Jane W.S. Liu. *Real-Time Systems*. Prentice-Hall Inc., 2000.

[8] Padmanabhan Pillai and Kang G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *SOSP '01 Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[9] Dongkun Shin, Woonseok Kim, Jaekwon Jeon, Jihong Kim, and Sang Lyul Min. SimDVS: An Integrated Simulation Environment for Performance Evaluation of Dynamic Voltage Scaling Algorithms. *Springer-Verlag Berlin Heidelberg*, 2003.

[10] Youngsoo Shin and Kiyoung Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. *DAC '99, ACM*, 1999.

[11] Youngsoo Shin, Kiyoung Choi, and Takayasu Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, pages 365 –368, 2000.

Appendix A

Pseudo code of the LPP scheduler

```
1  if current_frequency < maximum_frequency then
2          increase the clock frequency and the supply voltage to ...
               the maximum value;
3          exit;
4  end if
5
6  while delay_queue.head.release_time ≤ current_time do
7          move delay_queue.head to the run_queue;
8  end do
9
10 if run_queue.head.priority > active_task.priority then
11         set the active_task.executed_time;
12         context switch;
13 end if
14
15 if run_queue is empty then
16         if active_task is null then
17                 set timer to (delay_queue.head.release_time - ...
                       wakeup_delay);
18                 enter power down mode;
19         else
20                 speed_ratio = Compute_speed_ratio();
21                 find a minimum allowable clock frequency ≥ ...
                       speed_ratio * max_frequency;
22                 adjust the clock frequency along with the ...
                       supply voltage;
23         end if
24 end if
```

Appendix B

Pseudo code of the CCRM scheduler

```
1  assume f_j is frequency set by static scaling algorithm
2
3  select frequency():
4          set s_m = max_cycles_until_next_deadline();
5          use lowest freq. f_i ∈ {f_1,...,f_m|f_1 < ... < f_m}
6          such that (d_1 + ... + d_n)/s_m ≤ f_i/f_m
7
8  upon task_release(T_i):
9          set c_left_i = C_i;
10         set s_m = max_cycles_until_next_deadline();
11         set s_j = s_m · f_j/f_m;
12         allocate_cycles(s_j);
13         select_frequency();
14
15 upon task_completion(T_i):
16         set c_left_i = 0;
17         set d_i = 0;
18         select_frequency();
19
20 during task_execution(T_i):
21         decrement c_lefti and d_i;
22
23 allocate_cycles(k):
24         for i = 1 to n, T_i ∈ {T_1,...,T_n|P_1 ≤ ... ≤ P_n}
25         /* tasks sorted by period */
26
27                 if(c_left_i < k)
28                         set d_i = c_left_i;
29                         set k = k− c_left_i;
30                 else
31                         set d_i = k;
32                         set k = 0;
```

www.imm.dtu.dk

DTU Informatics
Department of Informatics and Mathematical Modelling (IMM)
Technical University of Denmark
Richard Petersens Plads
Building 321
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 33 51
Fax: (+45) 45 88 26 73
EAN: 5798000430204
Email: reception@imm.dtu.dk

IMM-M.Sc.-2012-xx