Performance of Real-Time Scheduling on Sensor Nodes
Comparing scheduling algorithms, resource policies and energy conservation methods in AmbientRT

T. Bijlsma

University of Twente
Department of EEMCS
Distributed and Embedded Systems research group (DIES)
P.O. Box 217, 7500 AE Enschede
The Netherlands
bijlsma@cs.utwente.nl

July 6, 2006

Graduation committee
ir. P.G. Jansen
ir. T.J. Hofmeijer
dr. S.O. Dulman

Samenvatting

Wireless sensor networks worden gebruikt om de omgeving in de gaten te houden. In het nieuwe tijdperk van ubiquitous computing worden zulke netwerken op veel locaties ingezet om functionaliteit en ondersteuning aan de gebruikers te bieden. Deze netwerken zijn opgebouwd uit kleine sensorknooppunten, uitgerust met een snelle microcontroller en sensoren en worden meestal gevoed door een batterij. Om voorspelbaar gedrag te krijgen, zijn de sensorknooppunten voorzien van een real-time besturingssysteem en resource beheer. Een resource is functionaliteit die mogelijk beschikbaar is voor taken op het sensorknooppunt, bijvoorbeeld de radio, een sensor of specifieke data. Daarnaast kunnen er nog energiebesparende algoritmes toegepast worden om de vrije tijd te benutten van de sensorknooppunten, om zo de levens duur van de batterij te verlengen. Dit rapport vergelijkt de prestaties van drie real-time scheduling algoritmes op sensorknooppunten, namelijk het Earliest Deadline First (EDF), het Deadline Monotonic (DM) en het Rate Monotonic (RM) algoritme. Het rapport vergelijkt ook de transaction en Nested Critical Section (NCS) resource beheermethodes, waarmee de onderzochte schedulers uitgebreid kunnen worden om de resources in het systeem te beheren. Als laatste worden twee nieuwe methodes om energie te besparen in combinatie met real-time scheduling geïntroduceerd en onderzocht. Het Earliest Deadline First with Inheritance and Scaling (EDFIS) beleid verlaagt de frequentie van de microcontroller, terwijl het Temporal Shutdown Scheduling (TSS) beleid de processor uitschakelt als er niets te doen is. De testen tonen aan dat het gedrag van het EDF algoritme het beste is. De transaction resource beheermethode is de beste oplossing als er veel gedeelde resources in het systeem zijn. Als er een gemiddeld aantal resources zijn, presteert het NCS resource beheeralgoritme beter. Als de taken een klein deel van de processortijd beslaan, wordt de meeste energie bespaard met het TSS beleid. Wanneer de taken een groot deel van de processor gebruiken, bespaart het EDFIS beleid de meeste energie.

Abstract

Wireless sensor networks are used to monitor the environment. In the new era of ubiquitous computing such networks are employed at different locations, to provide functionality and support to the users. These networks are composed out of small sensor nodes, which contain a fast microcontroller and sensors and are typically fed by a battery. To achieve predictable behavior the sensor nodes are equipped with a real-time operating system and a resource policy. A resource is a functionality that is possibly available to the tasks on the sensor node, for example the radio, a sensor or specific data. Furthermore energy conserving policies can be applied to utilize the idle time of the sensor node, to extend its life time. This report compares the performance of three real-time scheduling algorithms on a sensor node, namely the Earliest Deadline First (EDF), the Deadline Monotonic (DM) and the Rate Monotonic (RM) algorithms. The report also compares the transaction and the Nested Critical Section (NCS) resource policies, that can extend the examined scheduling protocols to manage shared resources. Two new policies to conserve energy in combination with real-time scheduling are introduced and examined. The Earliest Deadline First with Inheritance and Scaling (EDFIS) policy lowers the frequency of the microcontroller, while the Temporal Shutdown Scheduling (TSS) policy disables it when there is idle time. Performed tests show that the behavior of the EDF algorithm is most desirable. The transaction resource policy is the best solution when there are a lot of shared resources in the system. When there is an average amount of resources in the system, the NCS resource policy performs best. When a task set has a low utilization, most energy is conserved with the TSS policy. Above an average utilization the EDFIS policy shows the best power savings.

Contents

Preface

The results of the research performed form my master thesis are presented in this report. The research has been performed at the Distributed and Embedded Systems (DIES) research group of the University of Twente. It lasted from September 2005 till Juli 2006.

I would like to thank Pierre Jansen and Tjerk Hofmeijer for providing new ideas and helping me with encountered problems. Furthermore I would like to thank Stefan Dulman for helping me with improving the readability of this report. I also would like to thank my graduation colleagues for their help on numerous problems. Last but not least I would like to thank my family and Dieuwke for their support.

Tjerk Bijlsma
Enschede, July 6, 2006

Chapter 1

Introduction

With the growing popularity of Wireless Sensor Networks (WSNs), the demand increases to perform time critical operations within such networks. A WSN should be ubiquitous and independent. The sensor nodes in such a network are more than simple sensors. In the current WSN a sensor node contains a radio, ports for multiple sensors and mostly a multiple mega hertz microcontroller. These sensor nodes have to provide a wide range of functionality as long as possible, while they use their scarce energy from a battery.

Most of the tasks a sensor node in a WSN has to perform, are periodic tasks. Examples of such tasks are the monitoring of the temperature each 3 minutes, periodically displaying new (status) information on a Liquid Crystal Display (LCD) or communicating with nodes in the neighborhood. In some cases these tasks have to be performed in real-time, which means that they have to be started and finished at a given time. This could be the case when nodes have to synchronize periodically, with the nodes in the neighborhood. Full-time listening would consume a lot more energy compared to the case in which all nodes would synchronize and send, one after another, their status information, every hour. To achieve this, a task in the Operating System (OS) should turn the receiver on at the right moment. Therefore the scheduler of the kernel should give *real-time guarantees* on task execution.

Giving guarantees on task execution comes with a price. First, *priorities* should be given to tasks. With priorities for tasks an ordering in the execution of the tasks is made. When a task is released or a task finishes the OS can decide, given the priorities, which of the available tasks should be executed next. Beside that, it is not possible to just add tasks to the task list in a real-time OS. The OS should check or know the feasibility of the extended task list, before the new task is accepted.

As in regular operating systems a real-time operating system should also manage the *access to its resources*. Examples of resources that can be available in an OS, are: the radio, the serial port or a sensor. Since tasks share resources a kind of lock should be provided for mutual exclusive access to resources, when they are needed by a task. With the priorities of the tasks, as assigned by the real-time OS, mutual exclusion can be granted in multiple ways.

Even when the sensor node has nothing to do, the real-time OS can do something useful. Since energy is scarce for sensor nodes, the sensor node can turn unused devices off in idle time to *save energy*. To use the available energy and idle time optimal, the real-time kernel can shutdown the CPU or lower its clock frequency.

A real-time OS spends spends time on deciding which task should be scheduled next and, in some cases, determining if new tasks can be added. The OS should also provide support for exclusive access to its resources. It would be even better if the OS could decide how to save power by using the available idle time. The OS needs additional processing power to make such decisions, while processing power and energy are typically scarce for a sensor node. For that reason it is interesting to determine what kind of scheduling method and resource management systems performs best under which circumstances and which method is best to use the idle time.

1.1 Research goals

In this report research is presented on lightweight real-time scheduling in the real-time operating system *AmbientRT*. The research is conducted by three main goals:

- Compare the performance of the Deadline Monotonic with Inheritance, the Earliest Deadline First with Inheritance and the Rate Monotonic with Inheritance real-time scheduling algorithms in AmbientRT.
- Compare the performance of the nested critical section resource policy with the performance of the transaction resource policy in AmbientRT.
- Explore extensions for the scheduler in AmbientRT to achieve energy efficient scheduling, by using the idle time.

1.2 Organization of the Report

This report starts with the state of the art in chapter 2, where the current state of the technology is discussed. First the idea of a WSN is discussed, followed by lightweight operating systems. In the last section of this chapter energy aware scheduling is discussed. In chapter 3 real-time scheduling is explained. The chapter starts with the general scheduling theory and resource policies. At the end of the chapter the rate monotonic, deadline monotonic and earliest deadline first scheduling algorithms are highlighted. In chapter 4, earliest deadline first with inheritance and scaling is introduced. This chapter explains the theory that supports this offline frequency scaling policy. Chapter 5 discusses the wireless sensor node. First the hardware platform is discussed, followed by a section discussing the real-time operating system called AmbientRT. The implementation of the scheduler extensions is discussed in chapter 6. The first thing highlighted are the extensions of the AmbientRT scheduler. Next the feasibility analysis and the implementation of the two energy conserving scheduler extensions are discussed. Chapter 7 discusses the performed test to compare the alternatives. First three scheduling algorithms are compared on their processor load, followed by the comparison of two resource policies. The last part of this chapter discusses the comparison of two energy conserving policies. The conclusion of the research is given in chapter 8, followed by the recommendations for future work in chapter 9.

Chapter 2

State of the art

This chapter discusses the state of the art. The new computer era of *ubiquitous computing* will be discussed, including the WSNs. Special attention is given to how *WSNs* work and what their purpose is. These kind of networks require independent devices running *lightweight operating systems*, these operating systems are discussed in the next section. Since these devices should be energy efficient, this chapter concludes with a section about *energy aware scheduling*.

2.1 Ubiquitous computing

The main idea of ubiquitous computing is funded by Mark Weiser. After the era in which everybody got his personal computer, the era of *ubiquitous computing* has already started. In this era the computer should not demand the focus and attention of the user, rather it should fade to the background and give the focus to the problem at hand. When people learn to use a ubiquitous system sufficiently well, they cease to be aware of it. Nowadays there are systems equipped with microcontrollers which already activate the world around us, for example the stereo, the oven, light switches or the thermostat. Weiser expected the future to provide an interconnection of these devices in a ubiquitous network. This kind of network interacts with the user by sensing what the user wants and communicating it to other parts of the network, therefore location is important in such a network. A wireless sensor network is an example of such a network.

2.2 Wireless sensor networks

In the introduction WSNs are mentioned, but not properly explained. Therefore this section will address what the purpose of a WSN is and how it can function, followed by the description of a typical sensor node.

2.2.1 Network types

At the moment of writing the development of WSN is still busy, but the technology is becoming mature. The first products equipped with sensor nodes become available to the market. Still a lot of research is performed on a lot of different WSNs, which increases the application area. Culler et al. [9] provides the following three categories:

- WSN monitoring spaces: Habitat monitoring, climate control, surveillance and intelligent alarms;
- WSN monitoring things: Structural monitoring, condition based equipment monitoring and urban terrain mapping;

- WSN monitoring interactions of things with each other and the encompassing space: Complex interactions, wild live habitats, ubiquitous computing environment and health care;

This is a high level categorization based on the complexity of the service performed by a WSN. In all three the categories, the WSN consists of small devices that are monitoring variables in the environment. These devices, called sensor nodes, communicate the monitored values to another sensor node. In the first two categories the network has a typical tree structure, where all the values are send to the root sensor node. The root node is connected to a more powerful network or processes the variables itself. The last category, including ubiquitous computing, interacts with the user, which demands more processing power of the WSN. This type of WSNs requires communication between nodes, to enable the reaction of one node when another senses specified behavior.

The deployment of a WSN should be inexpensive. In general the nodes in a WSN should be cheap and replaceable. To avoid reconfiguration when replacing sensor nodes, the network should be self organizing. This means that sensor nodes are able to discover their neighbors. Using a distributed routing algorithm, the nodes can send packets for their applications. Packets may pass multiple nodes before they reach their destination, also called multihop routing. Note that mobility or physical placement of a node can limit its connectivity. The transmission range can be limited by walls or the direction of the antenna.

2.2.2 Sensor node

A typical sensor node consists of a microcontroller, a transceiver and an array of sensors, powered by a battery. This makes the sensor node a *small stand alone device* that can gather information, perform algorithms and transfer information to neighbors. The microcontroller of a node typically performs computations at multiple megahertz and has 2 to 10 kB of RAM. Compared to a modern PC, a sensor node has a very limited computational power and storage. Sensor nodes should be inexpensive and replaceable, therefore low cost hardware is used. This causes that the used sensors are not completely reliable and accurate and not always available. Therefore a WSN tries to combine multiple sensor nodes to get reliable results on demand.

The power consumption of sensor nodes is typically in the range of one to five milliwatt. Sensor nodes that are battery powered, have their life time limited by the capacity of the battery. A 1.5 volt alkaline battery can deliver 2,600 milliamp-hour [26]. A sensor node equipped with two alkaline batteries, using 1 milliamp at 3 volt, would have a life time of 2,600 hours, when it would be full time operational. Alternatives for battery power are solar power and mechanical generated power. Solar cells can deliver about 10 mW/cm^2 outdoor and 10 to 100 μW/cm^2 when used indoors. Mechanical power can be generated by for example the movement of windows or the vibrations of an air duct and delivers approximately 100 μW.

2.3 Lightweight Operating Systems

Since the usage of WSN is increasing, there is an increasing demand for lightweight OSs on sensor nodes. Multiple OSs are available, this section will address two of them. First the commercially available Salvo RTOS will be discussed. The open-source TinyOS will be discussed as second lightweight OS.

2.3.1 Salvo RTOS

The commercially available Salvo Real-Time Operating System(RTOS) is developed by Pumpkin, inc [18]. The OS is available for a wide range of micro-controllers, which includes the TI MSP430. Pumpkin offers the possibility to enable features in the OS, which causes the used memory to increase. Still the OS requires a small amount of RAM and no general purpose stack.

Salvo RTOS provides binary and counting semaphores to lock resources or data. The inter process communication is done by messages. It is possible to use message queues for multiple messages. Timers and events can be used to trigger task release.

The scheduling algorithm used in this RTOS is cooperative scheduling. Salvo RTOS provides 15 priorities, where a priority can be shared among multiple tasks. Cooperative scheduling is based on the idea that tasks need to cooperate with preemption. This means that a running task is only preempted when it allows the higher priority task to. The only forced preemption is performed by interrupts. When a task is preempted, its state is saved to the "hardware stack".

The advantage of Salvo RTOS is that it is using a small *amount of resources*. The *context switch time* is typically very low, because only the OS can force a preemption.

A disadvantage of Salvo RTOS is that the usage of semaphores introduces the risk of *deadlocks or errors*. Pumpkin suggests to use timeouts when using semaphores, in case a timeout occurs an error is generated and the deadlock is prevented. Another disadvantage is that semaphores introduce the possibility of *priority inversion*. The usage of cooperative scheduling introduces a possible *worst-case response time*, since preemption is only allowed by the OS.

2.3.2 TinyOS

The lightweight OS TinyOS for sensor nodes is developed by the University of California at Berkeley [22] and the open-source community. At the moment of writing version 2.0 of TinyOS is made available. TinyOS is one of the first OSs developed for a resource scarce and inexpensive hardware platform. The OS is divided in three levels: a Hardware Presentation Layer (HPL), a Hardware Abstraction Layer (HAL) and a Hardware Independent Layer (HIL). This layered structure make the OS easy adaptable to new hardware platforms. Among the supported sensor boards is the Eyes node, this node is equipped with a MSP430 microcontroller and has a lot in common with the μnode v2.0.

TinyOS is a component based OS, where a component is a software block implementing a specific function. The components are ordered in a kind of graph. The lowest components in the graph are the ones implementing the HPL functionality, the components implementing functionality from the HIL are in the top of the graph. Each component has a command handler, an event handler, a fixed-size frame and one or more tasks to perform the function of the component. A higher component can issue a command to the component, which is immediately executed, so it behaves like a function call. Events from the lower components can propagate to the components on top of it and are scheduled in a queue.

The scheduler used in TinyOS is a non-preemptive FIFO scheduler. In the scheduler every task has its own slot, so each task can be scheduled only once. A slot in the scheduler contains a variable which can be set to reschedule the task after completion. This slot provides the possibility to queue additional calls to the task. The scheduler used by TinyOS is not real-time, but can be replaced by a real-time scheduler. The scheduler is not allowed to preempt, because it would violate the static concurrency analysis.

Version 2.0 of TinyOS provides extended support for timers. The OS provides the components a 32 kHz timer and optionally one or two independent high precision timers, with a millisecond granularity. Resources are introduced in version 2.0 of TinyOS, to provide an easy way to use shared buses and other shared devices.

An advantage of TinyOS is that it is *layered*, which makes the OS easy adaptable for other hardware platforms. The development by the *open-source community*, makes that there is a large user group and that the OS is frequently updated. Furthermore it is also a *lightweight* OS, so it can work with limited memory, energy and processing power.

TinyOS has a few drawbacks. The FIFO scheduler causes the OS to *react slow to hardware events*. Replacing the scheduler by a real-time scheduler could improve the lateness of these tasks, but the non-preemption constraint limits the effectiveness. It is *not possible to determine the load* of the independent tasks on the operating system. Without this information it is not possible to determine whether the task set for the OS is feasible. Since the applications are *configured at compile time* the task set cannot be adapted dynamically.

2.4 Energy aware scheduling

The recent development of mobile computing in scarce resource environments, requires efficient allocation of available resources. The energy consumption (P) of a processor is found with the equation $P = V^2 \cdot f \cdot C$. The energy consumption depends on the square of the voltage (V), the frequency (f) and the average switched capacity of the transistors (C). In general a processor is not full-time used. Since the largest energy savings can be achieved by lowering the voltage, most algorithms try to scale this variable. When scaling the voltage the delay in the circuit increases. With the increasing delay, the frequency should be reduced to compensate for the increased latency in the circuit. An obvious alternative, to voltage and frequency scaling, is to disable the CPU when it is idle.

In this section a few methods to apply voltage and frequency scaling and shutdown scheduling are discussed. First, four online task set scaling algorithms are discussed, which are quite similar. Next subsection 2.4.2 discusses a combination between online and offline task set scaling. In subsection 2.4.3 the Pinwheel model is discussed, where the task set can be altered to ease the application of scaling. Shutdown scheduling is discussed in subsection 2.4.4. Subsection 2.4.5 discusses elastic scheduling, which shows a kind of behavior that can be applied in power aware environment.

2.4.1 Online task set scaling

In literature multiple frequency and voltage scaling algorithms are proposed. Multiple articles, propose an online scheduler extension to utilize *slack time*. In the articles [20], [25] and [11] an extension for the Earliest Deadline First (EDF) scheduling is proposed. Sinha [27] even proposes an extension for a Rate Monotonic (RM) and EDF scheduler.

The articles use a simplified scheduling model for EDF and RM, where the deadlines are equal to the periods and no resources are considered. This makes it possible to use the simple EDF feasibility test, which states that the utilization (U) is smaller or equal to 1, this will be presented in section 3.8.2. The scaling of the tasks is performed online, which means that the scheduler determines the frequency of a task when it starts the task. Tasks are annotated with a maximum computation time C, which is in general higher then the actual computation time. The time a task has left between the time it finishes and when it should be finished, is called the *slack time*. The mentioned extensions, propose to record the slack time of the previous task. Since the slack time is unused time, the next task can be slowed down to use it. Because the scaling for this task is performed with its computation time (C), it probably finishes earlier, enabling the next task to be scaled.

2.4.2 Offline and online task set scaling

In [19], a combination of online and offline scaling is proposed. The proposed algorithm is an extension of RM scheduling, with a simplified scheduling model without resources. The offline part of the algorithm scales the utilization of the task set to the least upper bound of the RM schedule (see equation 3.15). The online scaling is performed using the slack time of the tasks, as explained in the previous subsection.

2.4.3 Pinwheel model

In [14] the Pinwheel model is presented. This model can be used to rearrange the periods of a task set, scheduled with the RM algorithm and without resources. The periods of the task are rearranged and decreased in such a way that the periods become *harmonic*. When a smaller period is assigned to a task it is called more frequent, therefore the scheduler used with this model should transforms the additional calls to idle time. Using this, the task set stays feasible and the real-time constraints of the original task set stay valid. The harmonic periods cause less scheduler calls, even though the periods are shorter.

In addition, [13] uses the offline Pinwheel model to provide information for predictable online scheduling.

An online Linear Programming (LP) algorithm is used to calculate the optimal frequencies for the tasks in the altered task set. Additional energy savings are gained by using the actual computation times of the tasks in the LP algorithm, although the real-time constraints of the task set are not guaranteed in this case.

2.4.4 Shutdown scheduling

Jejurikar [16] proposes a combination of *offline task set scaling* and *shutting down* the CPU when it is not needed. These methods can be applied on task sets scheduled with EDF, without resources and with tasks having deadlines equal to their periods. A critical frequency for the CPU is found in this article, below which the leakage of the CMOS exceeds the savings achieved by applying a lower voltage. Therefore it is proposed to scale the task set to a frequency with the optimal combination of leakage and power per cycle.

The article addresses a few draw backs that can be encountered, when shutting down the CPU for small periods. The overhead that can be encountered are that the registers need to be stored and that the caches and the transition look aside buffer are lost. Therefore it is proposed to create large gaps in the schedule, with a procrastination algorithm, in which the CPU can be disabled. The article claims that the combination of shutting down the CPU in idle time and scaling the task set to the optimal frequency delivers optimal power savings.

2.4.5 Elastic scheduling

Buttazzo [6] proposes elastic scheduling for flexible workload management. Although it is not the main purpose to save energy, the described algorithm could be applied to enable flexible power consumption. The main idea is that periods of tasks can be treated as springs. This makes the utilization of the tasks flexible. The scheduling algorithm used is EDF and can be extended with the stack resource policy. The scheduling model assumes that the deadline is equal to the end of a period for a task. Each task is annotated with a minimal and maximal period, to denote the elasticity. When the period of the task increases, the service delivered by the task decreases and the utilization of the task set drops. The periods of the tasks in the task set can be increased to lower the utilization, which makes it possible to insert a new task. The periods of tasks can also be increased or decreased to adapt the quality of the system during run time. In combination with shutdown scheduling, elastic scheduling can provide a certain quality of service and shutdown scheduling can save energy during idle time.

Chapter 3

Real-time scheduling

Real-time systems can be divided in two classes, *hard* and *soft* real-time systems. When a task fails in a hard real-time system, correct system behavior cannot be guaranteed. To cover the worst-case scenario, the feasibility of a schedule for such a system should be verified. If a task fails in a soft real-time system, the behavior of the system stays correct, only the quality of the delivered services will decrease.

In the first section of this chapter an introduction to real-time scheduling is given. Following, section 3.2 presents the general scheduling assumptions. The next section discusses the general scheduler model. In section 3.4 resources in general and two possible resource policies are discussed. Admission and removal of tasks from task sets during run time, is discussed in section 3.5. This chapter concludes with the discussion of the RMI, DMI and EDFI real-time scheduling algorithms. The discussion will start with a section about the simple and fast Rate Monotonic scheduling, followed by a section about its extension, called Deadline Monotonic scheduling. The Earliest Deadline First algorithm differs from the other two and is discussed last. These three sections have a similar structure, with an introduction of the scheduling algorithm, its definition, the blocking conditions and the feasibility analyses.

In this chapter the definitions and theory behind the scheduling problem will be addressed. Although some readers will be familiar with it, it can be read to clarify definitions or methods in the report.

3.1 Basic definitions

In the discussion of real-time scheduling a few basic definitions are used. In most cases the notation of Jansen [15] and Buttazzo [7] are used.

The first to discuss is *periodic tasks*. A periodic task is performed on a given frequency. The difference between *concrete* and *non-concrete* periodic tasks [17], is that concrete periodic tasks are non-concrete periodic tasks extended with a phase. The phase of task i is denoted with Φ_i and represents the time after time $t = 0$, when the periodic task is released.

A *periodic task* is described with τ_i, i $\in$ (1,..,n). Each periodic task τ_i has a *relative deadline* D_i, a *minimum period* T_i and a *maximum computation time* C_i. An invocation of a task is called a *job*. The kth invocation of task i is denoted with τ_i^k, k$\in$(1,2,...). Each invocation of τ_i has its own *absolute release time* r_i^k and *absolute deadline* d_i^k. Constraints for the absolute values of periodic tasks are: $d_i^k = r_i^k + D_i$, $d_i^k \leq r_i^{k+1}$ and $r_i^{k+1} - r_i^k \geq T_i$ for any i $\geq$ 1, k $\geq$ 0. Beside the absolute release time and deadline, job τ_i^j has an *absolute starting time* s_i^j and *absolute finishing time* f_i^j. Using the absolute finishing time and the absolute deadline, the time a task has left till its deadline can be calculated, called the lateness L_i^j. The *lateness* of a job τ_i^j is given by $f_i^j - d_i^j = L_i^j$, where in general the lateness is negative. Figure 3.1 shows an example, which contains the discussed variables.

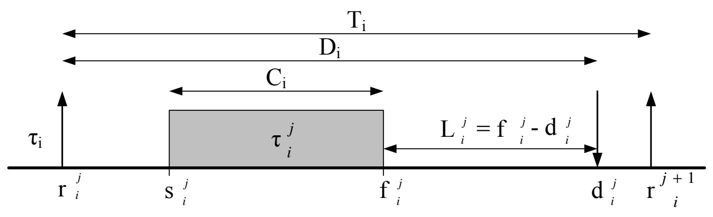


Figure 3.1: Single periodic non-concrete task with its parameters

Not every task is periodic. There are also tasks that are released at a random moment. Such tasks are called *sporadic tasks*. These tasks have the same relative and absolute variables as periodic task. The only different constraint is the one used for the release time. For a sporadic task the (k+1)th invocation happens at $r_i^{k+1} \geq r_i^k + T_i$.

Tasks that have to be executed are grouped in a task set. A *non-concrete periodic task set* is denoted with Γ and consists of *n* periodic tasks. Task sets containing concrete periodic tasks are denoted with Ω.

Beside the definitions for tasks and jobs there are also a few functions expressing the behavior of the tasks in the task set. Bellow the utilization, the workload, the processor demand function and the Baruah point will be defined.

Using the basic definitions of tasks, it is possible to calculate the utilization of a task set, as in definition 3.1.1. Note that when $U > 1$, the schedule is not feasible. There is more load than time to resolve it.

Definition 3.1.1. *(Utilization)*
The utilization *U, is the fraction of processor time spent executing the task set. It is defined as:*

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.1}$$

The *workload* W(t) of a task set is introduced by Audsley [3]. The equation for W(t) as defined below is used in the feasibility analysis of the EDFI scheduling algorithm.

Definition 3.1.2. *(Workload)*
The workload offered to the processor by all n tasks, between time 0 and t, is defined by:

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{3.2}$$

This function determines the times a task has been released till time t and multiplies it with the computation time of the task, the work load is the summation of these values for all the tasks. The feasibility algorithms for RMI and DMI scheduling use of an altered version of the workload function. This function calculates the workload only for the j higher priority tasks in the task set. The equation for this function is:

$$W^j(t) = \sum_{i=1}^{j} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{3.3}$$

The processor demand represents the amount of computation time requested by all jobs that are performed in the period between time 0 and time t. This can be calculated by multiplying the number of deadlines of each task in the period, with the computation time of this task and summing the resulting values for the whole task set. With this equation the feasibility of task sets, scheduled with the EDFI algorithm, can be determined.

Definition 3.1.3. *(Processor demand)*

The load that should be resolved by the processor, between t = 0 and t = n, is defined by:

$$H(t) = \sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor C_i \tag{3.4}$$

Baruah [28] derived an upper bound for the processor demand function. Using this upper bound a point can be found where this line crosses the processor capacity line, the t = L_B line. After this point, the processor demand will never be larger than the processor can manage.

Definition 3.1.4. *(Baruah point)*
The time L_B limits the period after which the processor demand will always be smaller than the processor capacity:

$$L_B = \frac{\sum_{i=1}^{n} (1 - \frac{D_i}{T_i}) \cdot C_i}{1 - U} \tag{3.5}$$

The point L_B can be derived from the H(t) function by first defining the upper bound of H(t), also called the Baruah line:

$$H(t) = \sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor C_i < \sum_{i=1}^{n} \left(\frac{t - D_i + T_i}{T_i} \right) \cdot C_i = B_L(t) \tag{3.6}$$

The point where this upper bound is equal to t can be found by solving the following equation:

$$\begin{aligned}
t &= \sum_{i=1}^{n} \left(\frac{t - D_i + T_i}{T_i} \right) \cdot C_i \\
t &= \sum_{i=1}^{n} \frac{C_i}{T_i} \cdot t + \sum_{i=1}^{n} \left(\frac{T_i - D_i}{T_i} \right) \cdot C_i = U \cdot t + \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} \right) \cdot C_i \\
t(1 - U) &= \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} \right) \cdot C_i \\
t &= \frac{\sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i} \right) \cdot C_i}{1 - U} = L_B
\end{aligned} \tag{3.7}$$

3.2 Assumptions

For the theory in this report, the assumptions of Jansen [15] are used:

- All tasks in the task set are periodic.
- The schedulers use a highest priority first algorithm.
- The discussed algorithms are non-idling. This means that when there are tasks released and waiting for execution, the processor cannot be idle.
- Scheduling overhead is assumed to be included in the maximum computation time C$_i$ of the task.
- All tasks are independent of each other. There are no precedence constraints, i.e. there is no task that has to wait with executing till another task, referenced with the precedence constraint, has finished.
- Deadlines consist of run-ability constraints only, i.e each task must be completed before the next request for it occurs.

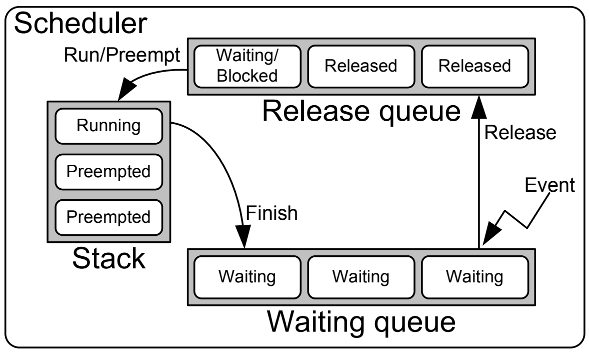


Figure 3.2: Real-time scheduler, with preemption stack, release queue and waiting queue

3.3 General model

In general the transaction system is used in real-time schedulers, see Figure 3.2. This system defines the release and waiting queue and the preemption stack. Tasks are moved from the waiting queue to the release queue when a certain event occurs, for example the event generated by a timer on the start of a new period can release a task. The tasks in the release queue are ordered according to the priority rules of the used scheduling algorithm. The scheduler decides, with the rules of the scheduling algorithm, if the task at the head of the release queue should preempt the task currently on top of the stack. This causes the stack to be arranged according to priority. When a running task finishes, it is placed in the waiting queue.

3.4 Resources

In most operating systems, tasks use resources. *Resources* are functionality available to the whole or parts of the system. Example resources are: specific data, a sensor or a radio for wireless communication. Because resources are shared, read and write access to them has to happen under *mutual exclusion*. It is assumed that multiple tasks are allowed to read a resource at the same time. Since scheduling algorithms with priorities are used, mutual exclusion can be achieved by *inheritance* of the priority from the highest priority task, that will access the resource. First the read and write floor and the inherited deadline are defined. At the end of this section two resource policies will be discussed.

Each resource has a *read floor* and *write floor*, which contain the highest priority of the tasks that read or write the resource, respectively. The read and write floor are defined by Jansen [15] and slightly modified by Maurer [24]:

Definition 3.4.1. *(Read floor D_R^r)*

$$D_R^r = min(\{\infty\} \cup \{D_i | \tau_i \in \gamma^w(R)\}) \tag{3.8}$$

where $\gamma^w(R)$ is the set of tasks that writes resource R.

Definition 3.4.2. *(Write floor D_R^w)*

$$D_R^w = min(\{\infty\} \cup \{D_i | \tau_i \in \gamma^{w+r}(R)\}) \tag{3.9}$$

where $\gamma^{w+r}(R)$ is the set of tasks that writes and/or reads resource R.

According to Jansen [15], it is possible to define the *inherited deadline* Δ_i of τ_i with the previous definitions. In this report, the deadline is used as value for the priority. In the special case of rate monotonic scheduling, the deadline should be equal to the period. When the deadline becomes smaller, the priority rises. Furthermore the set of resources read by τ_i, is denoted with ρ_i^r and the set of written resources with ρ_i^w. With these, the inherited deadline Δ_i can defined as:

Definition 3.4.3. *(Inherited deadline)*

$$\Delta_i = min(\{D_i\} \cup \{D_R^w | R \in \rho_i^w\} \cup \{D_R^r | R \in \rho_i^r\}) \tag{3.10}$$

3.4.1 Transactions

When tasks behave like *transactions*, a task only starts when it can use all the required resources without any synchronization. A transaction is run in such a way, that it is run to completion, it should not wait for resource access once it is started. It is however possible that a task is preempted by a higher priority task.

To make sure that a task is not blocked, when it acquires a resources, the deadline inheritance can be used (equation 3.10). This way, if the task can start, it is granted non-blocking use on the shared resources it uses, because it has the highest priority for that resources. Furthermore the priority inheritance can be determined statically. It is known at forehand which resources a task needs, so for each resource the read and write floor can be determined, which can be used to determine the Δ_i of the tasks.

The resources used by a task are define by ρ. Capital letters are used when write access to a resource is needed, lowercase letters for read access.

ρ	:	$\rho^w \rho \mid \rho^r \rho \mid \lambda$
ρ^r	:	$'a'..'z'$
ρ^w	:	$'A'..'Z'$

3.4.2 Nested critical sections

Tasks can also use *Nested Critical Sections* (NCS). An NCS denotes a period, during task executing that certain resources are used. The task will signal the OS when entering or leaving the NCS, this way the OS can arrange mutual exclusive access to the resources during the NCS. Note that, when tasks would be allowed to have only one NCS with the length of the task, the NCS resource policy would behave like the previous explained transaction resource policy.

The NCSs of task τ_i are denoted with $\mathcal{Z}_i$. Bellow two definitions of Jansen [15] are combined to define the NCS resource policy:

Definition 3.4.4. *(Nested Critical Sections)*
Task τ_i has m_i critical sections in $\mathcal{Z}_i$, where $\mathcal{Z}_i = \{Z_{i,0}, \dots, Z_{i,m_i}\}$.

- *$Z_{i,j}$ = ($\rho_{i,j}$, $\Delta_{i,j}$, $S_{i,j}$) where $\rho_{i,j}$ denotes the set of resources used in this critical section, $\Delta_{i,j}$ is the inherited priority for this critical section and $S_{i,j}$ denotes the maximum duration that these resources will be used.*
- *Critical sections fully overlap or are completely disjoint.*
- *$Z_{i,0} = (\emptyset, D_i, C_i)$, this describes the deadline and the computation time of the task.*

The inherited deadline as used in definition 3.4.3 concerns the resources used by a task, while $\Delta_{i,j}$ in definition 3.4.4 concerns only the resources used by one NCS. When ρ_i^r and ρ_i^w are redefined as the set of read or written resources, respectively, for an NCS, definition 3.4.3 can be used. This leaves the entering and the leaving of a critical section undefined:

Definition 3.4.5. *(Entering and leaving an NCS)*
When task τ_i enters a nested critical section $Z_{i,j}$, its Δ_i drops to $min(\Delta_i, \Delta_{i,j})$. When the NCS is left, Δ_i is restored to the value it had before entering.

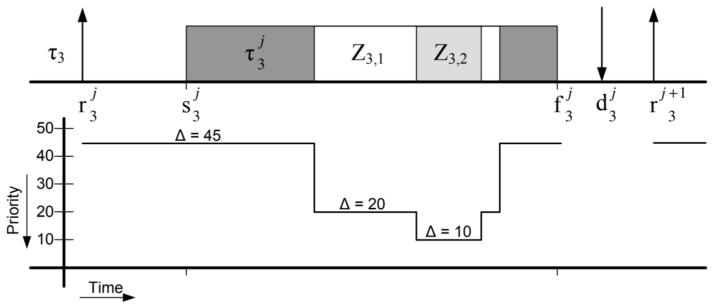


Figure 3.3: Nested Critical Section

As for transactions ρ defines the used resources for a task. Capital letters or lower case letters are used for write or read access, respectively. The time a task uses a set of resources is given by a float.

$$
\begin{array}{lcl}
\rho & \rightarrow & float\ \{\ \tilde{\rho}\ \rho\ \} \mid \lambda \\
\tilde{\rho} & \rightarrow & \rho^r \mid \rho^w \mid \rho^r\ \tilde{\rho} \mid \rho^w\ \tilde{\rho} \\
\rho^r & \rightarrow & 'a'..'z' \\
\rho^w & \rightarrow & 'A'..'Z'
\end{array}
$$

An example concerning the usage of NCSs is given in Figure 3.3. In this Figure task τ_3, from task set Γ_1 (Table 3.1), is highlighted. The task starts with its normal delta level of 45. When the task claims resource b and a for writing, its delta level drops to 20 and 10, respectively. These delta levels are equal to the deadlines of the tasks τ_2 and τ_1. The delta level of the task rises to the previous level when an NCS is left.

Table 3.1: Worst-case transaction task set Γ_1

Γ_1	D_i	T_i	C_i	ρ_i
τ_1	10	20	2	2{a}
τ_2	20	25	3	3{b}
τ_3	45	50	30	5{B 2{ A }}

3.5 Task admission and removal

When a task is added to a running task set, first the feasibility of the extended task set has to be analyzed. When the extended task set is feasible, the new task can be released. However, admitting the task may cause the inherited deadlines to change, which may jeopardize the ordering of the preemption stack. For example, when the task with lowest priority has the largest Δ, corresponding to the lowest priority, a not shared resource and is at the moment of admission at the bottom of the preemption stack. The new task has the highest priority and writes the resource of the lowest priority task. Adding this new task, would cause the Δ-value of the lowest priority task to drop beyond all the deadlines of the tasks on the stack, invalidating the stack ordering. This may cause transitive blocking.

Jansen proposed two methods to insert tasks without disturbing the stack ordering. The first solution is to *wait for idle time*. Since a new task can be added, the utilization of the task set is below 1, so there will be idle time. The second solution is to *alter the preemption condition*. This solution is outside the scope of this text, details can be found in [15].

Tasks can be instantly *removed* from a task set. Since the removal of a task can only cause the Δ-value to rise till the deadline of the task, the ordering of the stack stays valid.

3.6 Rate Monotonic with Inheritance

In 1973 Liu and Layland introduced Rate Monotonic (RM) scheduling [23]. They showed that RM is the *optimal* scheduling algorithm, among all *fixed priority assignment* scheduling algorithms.

Jansen [15] proposed the extension of Deadline Monotonic (DM) and RM scheduling with Inheritance, called DMI and RMI, respectively. The inheritance is introduced, so the tasks can use resources in a mutual exclusive way. Therefore each task τ_i is extended with a Δ_i as proposed in definition 3.4.3. The used resource policy is important for the determination of the maximum blocking time, which can be used to determine if task set are feasible.

Definition 3.6.1. *(RMI scheduling)*
The following rules define RMI scheduling:

- *The priorities of the tasks are assigned inversely proportional to the periods of the tasks.*
- $\forall\, \tau_i \,\in\, \Gamma,\; the\, D_i \,=\, T_i.$
- *If τ_i is in the ready queue, the running task τ_r will be preempted,*
  if $(T_i \,<\, T_r) \,\vee\, (T_i \,<\, \Delta_r) \,=\, T_i \,<\, \Delta_r$

3.6.1 Blocking

When RMI scheduling is used, mutual exclusive access to resources should be granted, as described in section 3.4. Therefore the inherited deadline Δ_i may contain a higher priority, than the τ_i actually has. When τ_i is running and there is a task in the ready queue with a priority between Δ_i and T_i, this task is blocked.

Definition 3.6.2. *(Waiting or blocking, with RMI scheduling)*
If τ_i is in the release queue and τ_r is on top of the stack of the processor, τ_i is waiting if:

$$T_r \leq T_i$$

τ_r blocks τ_i, if:

$$T_r > T_i \wedge \Delta_r \leq T_i = \Delta_r \leq T_i < T_r$$

In [15] it is proved that there can be at most *one blocker* when using the proposed resource policies, with DMI scheduling. Since RMI is a special case of DMI, where the deadlines are equal to the periods, the prove also applies to RMI. With RMI and DMI scheduling, the blocker is *running* or *preempted* at the run time stack. This because the run time stack is fully ordered. This implies, that there will be at most one task on the run time stack which satisfies the blocking condition. Since transitive blocking cannot occur, there will be no deadlocks.

It is possible to calculate the maximum blocking load for a task with $C_B(T_i)$, as proved in [15]. The function $C_B(L)$ can be defined by:

Definition 3.6.3. *(Blocking interval, with RMI scheduling)*
The maximum blocking experienced, when using transactions, in an interval with length L is expressed by:

$$C_B(L) = \max_{j}\{C_j | \Delta_j \leq L < T_j\} \tag{3.11}$$

The maximum blocking experienced, when using Nested Critical Sections, in an interval with length L is expressed by:

$$C_B(L) = \max_{j,k}\{S_{j,k} | \Delta_{j,k} \leq L < T_j\} \tag{3.12}$$

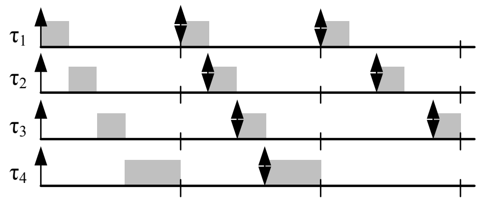


Figure 3.4: Critical instant for a lower priority task

3.6.2 Feasibility

The feasibility of a task set can be determined using a *sufficient* algorithm. This means that, when the task set is determined to be feasible it is feasible. In case it is found to be infeasible by the sufficient algorithm, it can still be determined feasible by the sufficient and necessary feasibility algorithm. If a *sufficient and necessary* feasibility algorithm determines that a task set is infeasible, it cannot be scheduled by the scheduling algorithm. In general a sufficient feasibility algorithm can be expressed with a short formula, such as the U_{lub} or the hyperbolic bound, see equation 3.15 and 3.16, respectively. In general a necessary and sufficient feasibility algorithm is a bit more complex.

First a short analysis of the tasks in the task set will be performed. Using this information, a sufficient scheduling algorithm for RM will be derived, called the hyperbolic bound. This section ends, with the discussion of the necessary and sufficient scheduling algorithm as introduced by Audsley et al.

Worst-case release times

To determine if a task set is feasible, it is best to check the worst-case situation for the tasks. For that reasons the critical instants are analyzed first. The analysis does not include inheritance, but stays valid when added. Besides that the analysis below is performed with RM scheduling. Analog to this analysis a version for DM scheduling can be derived.

Theorem 1. *(Critical instant for a task [23])*
A critical instant for any task occurs, if the task is released at the same time t as the higher priority tasks.

Proof : It is first shown that a critical instant occurs when a task τ_i is released at the same time as all higher priority tasks $\{\tau_j | \tau_j \in \Gamma; j < i; \Phi_i = \Phi_j = 0\}$. All tasks in the task set Γ are ordered according to their priority, where the tasks τ_1 and τ_n have the highest and the lowest priority, respectively.

Assume task τ_i is released later or at the same time as task τ_{i-1}, so $\Phi_i \geq \Phi_{i-1}$. According to RM $T_i > T_{i-1}$, so one job of τ_i will overlap at least two jobs of τ_{i-1}. A job of τ_{i-1} will delay a job of τ_i, when τ_{i-1} is still executing. To achieve the largest delay, the higher priority task should execute as long as possible while the job of τ_i still has to execute. As can be seen for the tasks τ_1 and τ_2 in Figure 3.4, the longest delay for τ_2 can be achieved when τ_1 and τ_2 are released simultaneously.

Repeating the previous argument for all tasks from the task set, it is shown that the worst-case response for the tasks occurs when they are released at the same time as the higher priority tasks. □

Knowing that the a critical instant occurs when all tasks in a task set are released simultaneously, the worst-case situation has not completely been defined. It is possible to maximize the utilization of the task set, by determining the worst-cast parameters for the tasks (Figure 3.4). According to Liu and Layland [23] these parameters are:

$$\begin{aligned}
&T_1 < T_n < 2T_1 \\
&C_1 = T_2 - T_1 \\
&C_2 = T_3 - T_2 \\
&\dots \\
&C_{n-1} = T_n - T_{n-1} \\
&C_n = T_1 - \sum_{i=1}^{n-1} C_i = 2T_1 - T_n
\end{aligned} \tag{3.13}$$

General feasibility condition

Jansen defined a feasibility condition for a task set scheduled with DMI, derived from the condition of Leung et al [21]. From this theorem the feasibility for a task set scheduled with RMI can be derived, since RMI can be regarded as DMI with $D_i = T_i$.

Theorem 2. *(RMI feasibility)*

A set Γ *scheduled with RMI is feasible under blocking, if:*

$$\forall i : 1 \leq i \leq n : \exists t : 0 < t \leq T_i : W^i(t) + C_B(D_i) \leq t \tag{3.14}$$

Proof : Jansen states that the proof is based on two observations. First, when all task are released at t = 0 the load is shifted to the left (theorem 1), which results in the maximal amount of load for i tasks, which can be examined with the W^i(t) function. Furthermore maximum amount of blocking for i tasks can be found with $C_B(D_i)$. The second observation is that, if for these i tasks the load can be resolved before D_i, which is equal to T_i, The worst-case situation can be handled, so Γ is feasible.□

Sufficient feasibility algorithms for RM

In [23] the least upper bound for the utilization of a task set scheduled with RM is derived. The least upper bound is defined by Liu and Layland as in equation 3.15. When *n* converges to ∞ in this equation, $\lim_{n\to\infty} U_{lub} = ln2 \approx 0.69$ can be found as least upper bound.

$$U_{lub} = n(2^{\frac{1}{n}} - 1) \tag{3.15}$$

The hyperbolic bound is derived in [4] and [5] by Bini et al. They derive a bound for the feasibility of a task set scheduled with RM, which is sharper than the least upper bound in equation 3.15. Therefore the derivation of the hyperbolic bound will be discussed next.

Theorem 3. *(Hyperbolic bound)*
A task set Γ *with n tasks, is schedulable with the RM scheduling algorithm if:*

$$\prod_{i=1}^{n}(U_i + 1) \leq 2 \tag{3.16}$$

Proof : Using theorem 1 and equation 3.13, the worst-case scenario can be described. When defining

$$R_i = \tfrac{T_{i+1}}{T_i} \; and \; U_i = \tfrac{C_i}{T_i}$$

equation 3.13 can be rewritten as follows:

$$\begin{aligned} U_1 &= R_1 - 1 \\ U_2 &= R_2 - 1 \\ &\dots \\ U_{n-1} &= R_{n-1} - 1 \\ U_n &= \frac{2T_1}{T_2} - 1 \end{aligned}$$

Now it can be seen that:

$$\prod_{i=1}^{n-1} R_i = \frac{T_2}{T_1}\frac{T_3}{T_2}\dots\frac{T_n}{T_{n-1}} = \frac{T_n}{T_1}$$

Using this, the maximum utilization of the task set can be derived. Note that the search minimizes the maximum utilization among the tasks. The found criterion for feasibility is:

$$U_n \leq \frac{2}{\prod_{i=1}^{n-1} R_i} - 1$$

Since $\forall i \in \{1, \dots, n-1\}\ R_i = U_i + 1$, it can be derived that:

$$(U_n + 1) \prod_{i=1}^{n-1} (U_i + 1) \leq 2$$

from which equation 3.16 can be derived. □

Necessary and sufficient feasibility algorithms for RMI

A *necessary and sufficient* feasibility analysis for RM is described by Audsley et al. [2]. Originally this algorithm was described for DM scheduling. The analysis is based on the examination of the longest *response time* R_i in a part of the task set. The response time is the sum of the *interference* I_i and the *computation time* C_i of task τ_i:

$$\begin{aligned} R_i &= C_i + I_i \\ I_i &= \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \end{aligned} \tag{3.17}$$

Since both sides of the equation contain the variable R_i, the smallest value that satisfies equation 3.17 should be found. A method is proposed with multiple estimations, where R_i^k denotes the kth estimate of R_i. The I_i^k is used to define the interference on τ_i in the interval [0,R_i^k]:

$$I_i^k = \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \tag{3.18}$$

R_i for task τ_i is calculated by the following three steps:

1. $R_i^0 = C_i \wedge k = 0$.
2. I_i^k for the interval [0,R_i^k] is computed by equation 3.18.
3. In case $I_i^k + C_i = R_i^k$, than $R_i^k = R_i$ is the worst-case response time.
   Else the next estimation can be obtained with $R_i^{k+1} = I_i^k + C_i$, jump to step 2 for the next iteration.

In the *main iteration* of the feasibility algorithm, the subset of the task set is extended with the highest priority task τ_i, which is not already in the subset. The response time R_i is the time in which the load of the subset with task τ_i, can be resolved. All tasks are released at the same time to create the worst-case instant, as shown in theorem 1. When the longest response time R_i, is smaller than the longest deadline D_i in the subset the subset is feasible. The analysis is extended to support inheritance and blocking. This is done by adding the possible blocking time B_i, of the remaining task set, to the response time R_i of the subset. This gives the following algorithm:

```
Algorithm RMI feasibility analysis
    schedulable = unknown;
    tasks = order_according_period($\Gamma$);
    processed = $\emptyset$;
    R = 0; // Response time
    I = 0; // Interference
    while ( tasks $\neq \emptyset \wedge$ schedulable == unknown) {
      // Examine the following task
      $\tau_i$ = get_first( tasks );
      $\tilde{R}$ = 0;
      $B_i = max_j\{C_j|\tau_j \in \Gamma; \tau_i \neq \tau_j; \Delta_j \leq T_i < T_j\}$;
      R = $C_i$ + $B_i$;
      while $((R \neq \tilde{R}) \vee (R > T_i))$ {
        $\tilde{R}$ = R;
        $I$ = 0;
        for { list = processed; list $\neq \emptyset$; $\tau_j$ = get_first(list) } {
          $I = I + \left\lceil \frac{\tilde{R}}{T_j} \right\rceil \cdot C_j$;
        }
        R = $C_i + I + B_i$;
      }
      if $(R > T_i)$ {
        schedulable = not_feasible;
      }
      append($\tau_i$, processed);
    }
    schedulable = feasible;
```

3.7 Deadline Monotonic with Inheritance

The Deadline Monotonic (DM) scheduling algorithm was introduced by Leung and Whitehead in 1982 [21]. The DM algorithm drops the constraint of RM, which states that $\forall \tau_i \in \Gamma : D_i = T_i$. This introduces the deadline parameter for each task. Leung and Whitehead also proved the *optimality* of DM, which means that DM schedules all possible task sets which other *static priority algorithms* can schedule.

As stated in the previous section, Jansen [15] proposed the extension of RM and DM scheduling with inheritance, called DMI and RMI, respectively. The definition of a task τ_i is therefore extended with the proposed Δ_i, from definition 3.4.3.

Definition 3.7.1. *(DMI scheduling)*
The following rules define the DMI scheduling:

- *The priorities of the tasks are assigned inversely proportional to the deadlines of the tasks.*
- *If τ_i is in the ready queue, the running task τ_r will be preempted, if $(D_i < D_r) \vee (D_i < \Delta_r) = D_i < \Delta_r$*

3.7.1 Blocking

The DMI scheduling algorithm is meant to schedule tasks with resources, so it is possible that blocking occurs. Since DMI scheduling is very similar to RMI scheduling, the main part of the blocking computation is similar to section 3.6.1. The major difference is the used priority variable. DMI scheduling uses the deadline as priority and RMI scheduling uses the period. To avoid confusion, the definitions will be restated for DMI, with the correct variables.

Definition 3.7.2. *(Waiting or blocking, with DMI scheduling)*
If τ_i is in the release queue and τ_r is on top of the stack of the processor, τ_i is waiting if:

$$D_r \leq D_i$$

τ_r blocks τ_i, if:

$$D_r > D_i \wedge \Delta_r \leq D_i = \Delta_r \leq D_i < D_r$$

Jansen [15] showed that there can be at most *one blocker* when using the proposed resource policies from section 3.4, with DMI scheduling. The blocker is *running* or *preempted* on the run time stack, because the run time stack is fully ordered. Since the preemption condition for DMI is used, every task on the run time stack has a smaller D_i than the Δ_j of the task below it on the stack. This implies that there will be at most one task on the run time stack which satisfies the blocking condition. As for RM, the absence of transitive blocking makes it impossible that deadlocks occur. Because it is not possible that tasks that share resources are at the run time stack at the same time, mutual exclusive access to shared resources is provided.

The maximum blocking load for a task can be found with $C_B(D_i)$. The function $C_B(L)$ can be defined by:

Definition 3.7.3. *(Blocking interval, with DMI scheduling)*
The maximum blocking experienced, when using transactions, in an interval with length L can be expressed by:

$$C_B(L) = \max_j \{C_j | \Delta_j \leq L < D_j\} \tag{3.19}$$

The maximum blocking experienced, when using NCSs, in an interval with length L can be expressed by:

$$C_B(L) = \max_{j,k} \{S_{j,k} | \Delta_{j,k} \leq L < D_j\} \tag{3.20}$$

3.7.2 Feasibility

The feasibility of a task set scheduled with DM scheduling can be determined with a sufficient and a sufficient and necessary algorithm. First the general feasibility condition for of a task set scheduled with DMI is introduced. Next a very simple sufficient feasibility algorithm is presented. This subsection concludes with the necessary and sufficient algorithm, which is an optimized version of the sufficient and necessary algorithm in section 3.6.2.

General feasibility condition

Theorem 4. *(DMI feasibility (Jansen))*
A set Γ scheduled with DMI is feasible under blocking, if:

$$\forall i : 1 \leq i \leq n : \exists t : 0 < t \leq D_i : W^i(t) + C_B(D_i) \leq t \tag{3.21}$$

Proof : The proof is as presented in theorem 2. The only difference is the priority parameter, period D_i which can be smaller than T_i, when DMI scheduling is used.□

Sufficient feasibility analysis for DM

From the feasibility test of Lui and Layland in equation 3.15 a sufficient test for DM scheduling can be derived. Since the period T_i of the tasks is increased, the utilization is decreased, which does not affect the feasibility. A task set scheduled with DM is feasible if the following equation holds:

$$\sum_{i=n}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1) \tag{3.22}$$

Necessary and sufficient feasibility analysis for DMI

In [15] an optimization of the original algorithm of Audsley et al. 3.6.2 is discussed. The algorithm of Jansen accounts every release of a task once, instead of recalculating the interference in every cycle of the main loop. An altered version of the workload function is used to determine the amount of interference. This function determines the workload for a subset $\{\tau_1 \ldots \tau_j\} \subset \Gamma$, as presented in equation 3.3. Furthermore the algorithm is extended to account for blocking. The complexity of the optimized feasibility algorithm is of *O(n)*.

A graphical representation of the algorithm is given in Figure 3.5 for task set Γ_2 in Table 3.2. The blocking is visualized with the gray dotted lines, with a cross on top of it. Note that the optimized feasibility algorithm does not start at t = 0 for each $W^j(t)$, but takes $W^{j-1}(t)$ and simply adds C_j to it and continues from the previous t.

The algorithm presented below has been slightly adjusted, but the core functionality is the same. Each iteration of the main loop extends the examined task set with the highest priority task, which is not already in the examined task set. After every execution of the inner loop, the variable W holds the value of $W^j(t)$.

Table 3.2: Example task set Γ_2, using NCS

Γ_2	D_i	T_i	C_i	$\mathcal{Z}_i$
τ_1	3	5	1	1{a}
τ_2	6	7	2	1{b}
τ_3	7	7	3	1.5{B 1{ A }}

Algorithm *DMI feasibility analysis, optimized*

```
schedulable = true;
j = 1;
W = 0;
**while** $((j \leq n) \wedge schedulable)$ {
   B = $\max_k\{C_k|\Delta_k \leq D_j < D_k\}$;
   putOrdered(inferList,(0,j));
   (t,i) = (0,j);
   **while** $(t < D_j \wedge ((W + B > t) \vee (t == 0)))$ {
      W += $C_i$;
      shift(inferList);
      putOrdered(inferList,(t+$T_i$,i));
      (t,i) = first(inferList);
   }
   schedulable = (W + B) $\leq D_j$;
   j++;
}
```

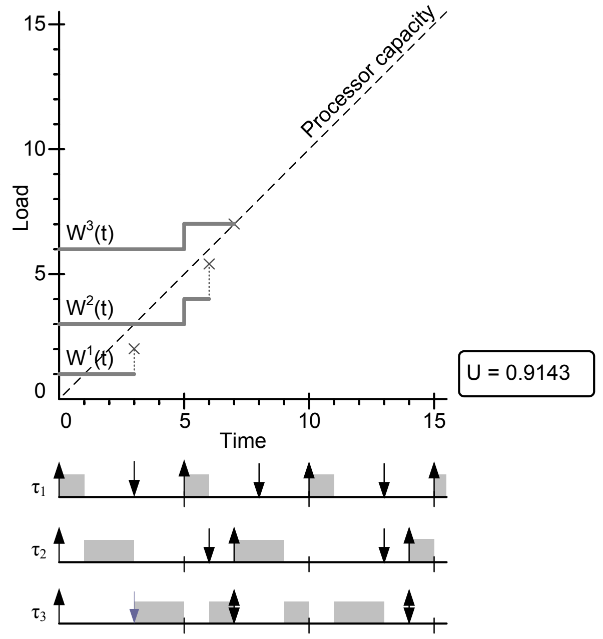


Figure 3.5: Graphical representation of DMI feasibility analyzes for task set Γ_2

3.8 Earliest Deadline First with Inheritance

With the introduction of RM scheduling Liu and Layland introduced deadline driven scheduling [23], nowadays known as Earliest Deadline First (EDF) scheduling. They introduced it as a dynamic priority scheduling algorithm, where the priorities are assigned to the task according to the deadlines of their current requests. They also prove that EDF is the *optimal* scheduling algorithm among all *dynamic priority scheduling* algorithms. This means that every schedule that is found feasible with any dynamic priority scheduling algorithm is feasible with the EDF scheduling algorithm. In [7] Buttazzo repeats the proof of Dertouzos, to show the optimality of EDF. With the proof Buttazzo also shows that EDF *minimizes the maximum lateness.*

Jansen proposed the extension of EDF with inheritance. The Earliest Deadline First with Inheritance (EDFI) algorithm can be used with the proposed resource policies in section 3.4.

Definition 3.8.1. *(EDFI Scheduling)*
The following rules define EDFI scheduling:

- *The priorities of tasks are assigned inversely proportional to the absolute deadlines of the tasks.*
- *If τ_i^j is in the ready queue, the running task τ_r^k will be preempted, if $(d_i^j < d_r^k) \wedge (D_i < \Delta_r)$.*

3.8.1 Blocking

As RMI and DMI, the EDFI scheduling algorithm is also meant to schedule tasks with resources. Therefore the tasks are extended with the proposed Δ from definition 3.4.3. This means that it is possible that blocking occurs. In the next definitions the waiting and blocking will be defined.

Definition 3.8.2. *(Waiting or blocking, with EDFI scheduling)*
If the job τ_r^s is in the release queue and job τ_i^j is on top of the processor stack, τ_r^s is waiting if:

$$d_i^j \leq d_r^s$$

τ_i^j blocks τ_r^s, if:

$$d_i^j > d_r^s \wedge \Delta_i \leq D_r$$

In [15] Jansen shows that there is at most *one blocker* when using the proposed resource policies 3.4, with EDFI scheduling. The blocker is *running* or *preempted* at the processor stack. Since the tasks are allowed to preempt the running instance according to the he EDFI scheduling as defined in 3.8.1, the processor stack is ordered. This implies that there will be at most one task at the processor stack which satisfies the blocking condition, so no transitive blocking and deadlocks will occur. Furthermore *mutual exclusive* access to shared resources is provided when using one of the proposed resources policies, because two tasks which share resources cannot be on the run time stack at the same moment.

Jansen shows that the maximum blocking load for a task can be found with $C_B(D_i)$. The reasoning behind the formula is less trivial, as thought. It can be described with the following steps:

1. The interval [t,t+L] is examined to be feasible, including the possible blocking.
2. During feasibility analysis the maximum load is calculated in the interval [0,L], which also has the length L
3. If the load and the blocking do not exceed the processor capacity, it can be concluded that this will not happen for any interval with the length L, anywhere in time.

The blocking interval function $C_B(L)$ for EDFI is equal tot the one defined for DMI in definition 3.7.3.

3.8.2 Feasibility

As with the previous scheduling algorithms, EDFI also has *sufficient* and *necessary and sufficient* feasibility analyses. In 1973 Liu and Layland found a simple feasibility condition, which is necessary and sufficient, for a task set scheduled with EDF scheduling with $D_i = T_i$. They proved that if the utilization is below or equal to one, so U $\leq$ 1, the task set is feasible. This analysis does not concern deadlines or resources.

This section starts with the introduction of the general feasibility condition for a task set scheduled with EDFI. Following, a feasibility test for EDF proposed by Devi will be introduced. At the end of this section the necessary and sufficient feasibility test, as introduced by Jansen, will be discussed.

General feasibility condition

Jansen [15] derives the general feasibility condition for EDFI from the condition as defined by Liu and Layland. The condition is:

Definition 3.8.3. *(EDFI feasibility)*
A schedule Γ is feasible when EDFI scheduling is used, if:

$$\forall L \geq 0 : H(L) + C_B(L) \leq L \tag{3.23}$$

This definition is based on the fact that the processor demand $H(L)$ and the blocking $C_B(L)$ should not exceed the processor capacity L. When for every period the processor demand and the blocking are smaller than the processor capacity, the schedule is feasible.

Sufficient feasibility for EDF and EDFI

Devi [10] proposed a sufficient feasibility analysis for EDF. The algorithm is derived from the processor demand function.

Theorem 5. *Sufficient feasibility analysis for EDF scheduling*
A schedule Γ is feasible if:

$$\forall k : 1 \leq k \leq n :: \sum_{i=1}^{k} \frac{C_i}{T_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{T_i - \min(T_i, D_i)}{T_i}\right) \cdot C_i \leq 1 \tag{3.24}$$

Proof : The algorithm can be proved to be correct, by proving that if Γ is not schedulable by EDF than the previous defined condition is false. This can be done by considering a period [t$_{-1}$,t$_d$), in which a task τ_j misses its deadline. Task $\tau_k \in \Gamma$ is the task with the longest deadline D$_k$, where $D_k \leq (t_d - t_{-1})$. The tasks in the task set are ordered according their deadlines, where task τ_1 is the task with the smallest deadline. The upper bound of the load that has to be solved by the processor, in the period [t$_{-1}$,t$_d$), can be defined by:

$$\sum_{i=1}^{k} \left(\left\lceil \frac{t_d - t_{-1} - D_i + T_i}{T_i} \right\rceil \right) \cdot C_i$$

Since task τ_j, with j < k, misses its deadline at t$_d$, the following can be written:

$$\begin{aligned} t_d - t_{-1} <& \sum_{i=1}^{k} \left\lceil \frac{t_d - t_{-1} - D_i + T_i}{T_i} \right\rceil \cdot C_i \\ \leq& \sum_{i=1}^{k} \left(\frac{t_d - t_{-1} - D_i + T_i}{T_i} \right) \cdot C_i \\ \leq& \sum_{i=1}^{k} \left(\frac{t_d - t_{-1} - min(T_i, D_i) + T_i}{T_i} \right) \cdot C_i \\ =& \sum_{i=1}^{k} \frac{C_i}{T_i} (t_d - t_{-1} - min(T_i, D_i) + T_i) \end{aligned}$$

When both sides are divided by $(t_d - t_{-1})$, the equation becomes:

$$\begin{aligned} 1 <& \sum_{i=1}^{k} \frac{C_i}{T_i} \left(1 + \frac{T_i - min(T_i, D_i)}{t_d - t_{-1}}\right) \\ =& \sum_{i=1}^{k} \frac{C_i}{T_i} + \sum_{i=1}^{k} \frac{C_i}{T_i} \left(\frac{T_i - min(T_i, D_i)}{t_d - t_{-1}}\right) \\ \leq& \sum_{i=1}^{k} \frac{C_i}{T_i} + \frac{1}{D_k} \sum_{i=1}^{k} \frac{C_i}{T_i} (T_i - min(T_i, D_i)) \end{aligned}$$

In the last equation $(t_d - t_{-1})$ can be replaced by D_k, because $D_k \leq (t_d - t_{-1})$. This is false as claimed.□

Devi also introduces a variation of the feasibility analysis where blocking is considered, under the Protocol Ceiling Policy or the Stack Resource Policy. Since these protocols show similar blocking computations as with EDFI, the equation can be rewritten to:

$$\forall k : 1 \leq k \leq n :: \frac{C_B \cdot D_k}{D_k} + \sum_{i=1}^{k} \frac{C_i}{T_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{T_i - \min(T_i, D_i)}{T_i}\right) \cdot C_i \leq 1 \tag{3.25}$$

Necessary and sufficient feasibility for EDFI

A necessary and sufficient feasibility analysis can be derived from the general feasibility condition for EDFI, as given in definition 3.8.3. This analysis starts at time t = 0 with examining the deadline and release events

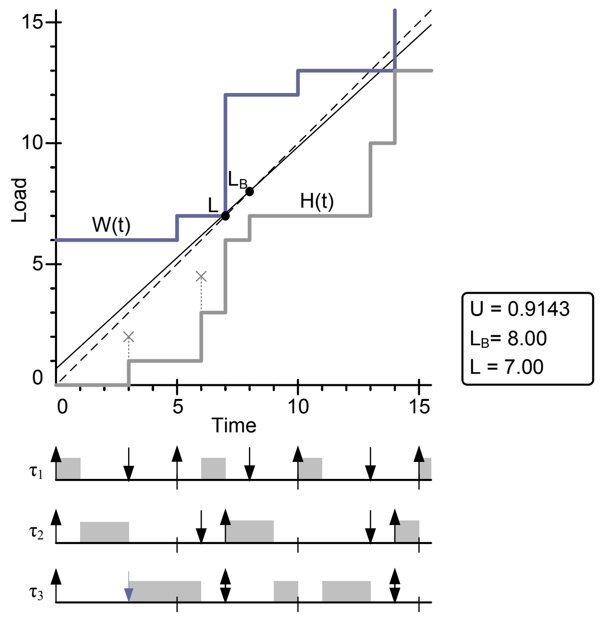


Figure 3.6: Graphical representation of EDFI feasibility analyzes for task set Γ_2

that occur. The processor demand function and the blocking are examined at every *deadline*, to verify that the feasibility condition holds. When a new task is *released* the workload is compared to the processor capacity. All tasks are released at t = 0 to examine the worst-case situation, according to subsection 3.6.2. This introduces the maximum $\frac{H(t)}{t}$ in the first part of the analysis. When the workload function is equal to the processor capacity line, point L, all load is resolved and the $\frac{H(t)}{t}$ will never be larger than the processor capacity again. Two other bounds for the time are the Baruah point L_B and the Least Common Multiple (LCM) of the periods. The Baruah bound is defined in definition 3.1.4 and shows that the upper bound of $H(t)$ is always smaller than the processor capacity after the point L_B, which means that the schedule is feasible after that point. When t is equal to the LCM of the periods of the task set, all tasks are released, because their periods start. This means that the same behavior will show up as when t = 0, so if the schedule was feasible before the LCM it will be after. Furthermore at the LCM of the periods the workload will be equal to the processor demand, which means all load is resolved, another check to determine that the task set is feasible. Figure 3.6 shows a graphical representation of the analysis for the task set Γ_2 from Table 3.2.

The algorithm below only examines the deadline and release events. These events are fetched as a tuple with the *GetNextEvent* function. The tuple contains the *time* t, a *flag* denoting if it is a deadline or a release event and the *maximum computation time* C. In case the event is of type *deadline*, a check is performed to see if the processor demand and the maximum blocking are smaller than the processor capacity. When a *release* event occurs the workload is examined. The bounds for the period that is examined are as discussed above. The necessary and sufficient feasibility algorithm for EDFI is presented below.

Algorithm *EDFI feasibility analysis*

```
schedulable = unknown;
H = 0; // processor demand
W = 0; // Workload
$L_B = \frac{\sum_{i=1}^{n}\left(1-\frac{D_i}{T_i}\right)\cdot C_i}{1-U}$;
**while** $(schedulable == unknown)$ {
  (t,flag,C) = GetNextEvent;
  **case** flag {
    deadline: {
      H = H + C;
      $C_B = \max_j\{C_j|\Delta_j \leq t < D_j\}$
      **if** (H + $C_B >$ t) { schedulable = no; }
    }
    release: {
      **if** $((t > 0) \wedge (W \leq t))$ { schedulable = yes; }
      W = W + C;
    }
  }
  **if** $((t > 0) \wedge (t \geq L_B))$ { schedulable = yes; }
}
```

Chapter 4

Earliest Deadline First with Inheritance and Scaling

When a task set Γ is scaled offline, it should still be feasible. The *offline scaling* of a task set means lowering the processor frequency f before the task set is running, this results in increasing computation times for the tasks. So the main objective of Earliest Deadline First with Inheritance and Scaling (EDFIS) is to determine the maximum amount of scaling of the tasks for which the task set is still feasible.

Scaling can be performed in two ways. The *first method* is to determine the maximum scaling for each independent task. This introduces an exhaustive search for the optimal frequency distribution. The *second method* is to determine the maximum scaling for the whole task set. Because a higher frequency corresponds to an exponentially higher power consumption, it is best to keep the frequencies of the independent tasks as close as possible to the average frequency. Therefore it would be optimal to find the lowest possible frequency for the whole task set.

This section examines the offline scaling of a task set, scheduled with EDFI. First an alternative to scaling all functions used in the feasibility analysis is proposed. The boundaries of the search space for the scaling factor are the point of discussion in section 4.2. This chapter concludes with a method to determine the optimal scaling factor for a task set.

4.1 Scaling the y-axes

When scaling the frequency with a factor $k < 1$, the C_i of all tasks in a task set are scaled with a factor $\frac{1}{k} > 1$. When applying the scaled $\tilde{C_i} = C_i \cdot \frac{1}{k}$ to the workload function 3.2, the processor demand function 3.4 and the Baruah line function 3.6, the scaling factor can be separated. This is shown in equation 4.1, 4.2 and 4.3. The processor capacity $p(t)$, with slope one (dotted line in Figure 3.6), stays the same. This because the time does not scale, only the load that can be resolved in one unit of time.

$$\sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil \cdot \frac{1}{k} \cdot C_i = \frac{1}{k} \cdot \left(\sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \right) = \frac{1}{k} \cdot W(t) \tag{4.1}$$

$$\sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor \cdot \frac{1}{k} \cdot C_i = \frac{1}{k} \cdot \left(\sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor \cdot C_i \right) = \frac{1}{k} \cdot H(t) \tag{4.2}$$

$$\sum_{i=1}^{n} \left(\frac{C_i}{T_i} \right) \cdot \frac{1}{k} \cdot t + \sum_{i=1}^{n} \left(\frac{T_i - D_i}{T_i} \right) \cdot \frac{1}{k} \cdot C_i = \frac{1}{k} \cdot \left(U \cdot t + \sum_{i=1}^{n} \left(\frac{T_i - D_i}{T_i} \right) \cdot C_i \right) = \frac{1}{k} \cdot B_L(t) \tag{4.3}$$

Since the factor $\frac{1}{k}$ can be extracted from all the functions, it is possible to adjust the scale of the y-axis instead. This results in all the functions staying the same, only the processor capacity line, with $p(t) = t$, has to be redrawn. Instead of altering the values of the y-axis they can be redefined by stating that they are multiplied with $\frac{1}{k}$. With this redefined scale the processor capacity line becomes $p(t) = k \cdot t$. Using the redefined y-axis, the H(t), W(t) and $B_L(t)$ stay the same, they don't have to be recalculated or redrawn for a new scaling factor.

4.2 Search boundaries

To determine the scaling factor $\frac{1}{k}$, the maximal possible utilization of the task set has to be found. When determining the maximal utilization two cases can be distinguished, the deadlines are equal to the periods, or there is at least one deadline smaller than the corresponding period.

If all tasks in the task set have $D_i = T_i$ and no resources are used the proposition of Liu and Layland can be used, as mentioned in subsection 3.8.2. They proof that the maximum utilization of the task set is 1. In this case the optimal scaling factor for the task set is $\frac{1}{k} \cdot U = 1$, so $\frac{1}{k} = \frac{1}{U}$.

However if the task set has a task with $D_i < T_i$, the analysis is less trivial. This is caused by the load that has to be resolved at the deadline of the task, which is earlier than the end of the period. This may causes the $\frac{H(t)}{t}$ to be larger than the utilization U. To determine the maximum $\frac{H(t)}{t}$ the behavior of the task set has to be examined till a certain time t. This can be done with the functions mentioned in equations 4.1 and 4.2.

To find the boundaries in which the maximum $\frac{H(t)}{t}$ can be found, the upper bounds and lower bounds for the processor demand and workload function are examined. In equation 4.4 the derivation of the lower bound for the processor demand function is shown. The upper bound of the processor demand function, also called the Baruah line function, is already derived in equation 3.6. Equation 4.5 and equation 4.6 shows the derivation of the lower and upper bound for the workload function W(t). Figure 4.1 shows a schematic drawing of these bounds. Note that the *slope* of all the bounds is U. It can be seen that the overlap of these areas grows when the difference between D_i and T_i increases. If the deadlines are equal to the periods, $\sum_{i=1}^{n}\left(1 - \frac{D_i}{T_i}\right) \cdot C_i$ is zero, so the upper bound of H(t) is the lower bound of W(t). Since W(t) $\geq$ H(t), the functions can only touch each other in the area or on the line.

$$\begin{aligned} H(t) &= \sum_{i=1}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor \cdot C_i \geq \sum_{i=1}^{n} \left(\frac{t - D_i + T_i}{T_i} \right) \cdot C_i - \sum_{i=1}^{n} C_i \qquad (4.4) \\ &= \sum_{i=1}^{n} \frac{C_i}{T_i} \cdot t + \sum_{i=1}^{n} (1 - \frac{D_i}{T_i}) \cdot C_i - \sum_{i=1}^{n} C_i = U \cdot t + \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i}\right) \cdot C_i - \sum_{i=1}^{n} C_i \end{aligned}$$

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \geq \sum_{i=1}^{n} \left(\frac{t}{T_i} \right) \cdot C_i = U \cdot t \qquad (4.5)$$

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil \cdot C_i \leq \sum_{i=1}^{n} \left(\frac{t}{T_i} \right) \cdot C_i + \sum_{i=1}^{n} C_i = U \cdot t + \sum_{i=1}^{n} C_i \qquad (4.6)$$

When a k is determined, it has to be verified that $\forall t : \{0 \ldots \infty\} \rightarrow$ H(t) $<$ $p(t)$. As with the EDFI feasibility algorithm from subsection 3.8.2, three bounds can limit the search space of his verification. The first bound is the point where *all load is resolved*, denoted with L. At this point the workload function W(t) touches or intersects the $p(t)$ line, meaning that all load is resolved and that $\frac{H(t)}{t}$ will never be larger than k again. The second bound is the *Least Common Multiple* (LCM) of all task periods. At this point all tasks are released simultaneously, causing the same behavior to show up as when t = 0. Furthermore the workload function touches the processor demand function when t is equal to the LCM, so it certainly touched $p(t)$. The third bound is the *Baruah point*. This is the point where the $p(t)$ function intersects the upper bound of the processor demand function $B_L(t)$. After this point $p(t)$ is always larger than $B_L(t)$.

The Baruah point, as in definition 3.1.4, can be determined using the upper bound of the processor demand

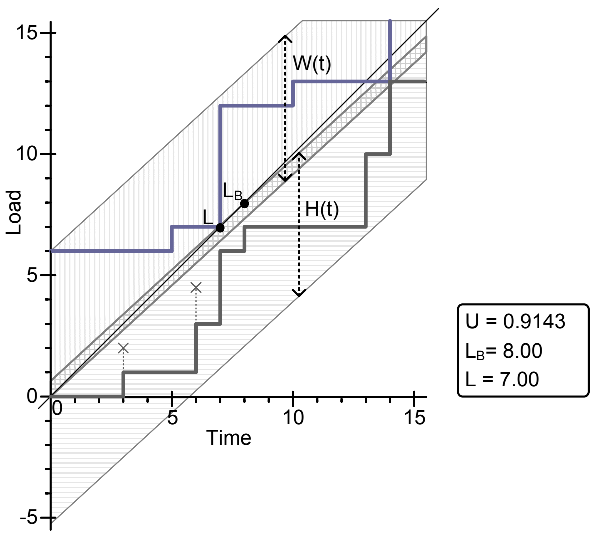


Figure 4.1: The bounds for the processor demand function H(t) and the workload function W(t)

function $B_L(t)$. When scaling occurs this function is also scaled with a factor $\frac{1}{k}$. Just as with the workload and the processor demand function this line can be kept the same when the processor capacity $p(t)$ is scaled. To find the *scaled Baruah point* $B_L(t) = k \cdot t$ should be solved, as presented in equation 4.7. The found t is the third boundary, when searching the scaling factor.

$$\begin{aligned} k \cdot t &= U \cdot t + \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i}\right) \cdot C_i \\ t \cdot (k - U) &= \sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i}\right) \cdot C_i \\ t &= \frac{\sum_{i=1}^{n} \left(1 - \frac{D_i}{T_i}\right) \cdot C_i}{k - U} \end{aligned} \tag{4.7}$$

The three bounds given on the time, try to prevent the occurrence of an exhaustive search. Nonetheless the scaling causes the utilization to rise to 1. A high utilization causes idle time to be postponed until a later point in time. This means that when k = U, the workload will only be equal to $p(t)$ at the LCM of the periods. When k = U both $p(t)$ and $B_L(t)$ have slope U, so no intersection will occur. Note that if k is close to U, the idle time and the Baruah point are placed far away. A task set can be scaled to a utilization close to U when the deadlines of the tasks get near their periods. Note that the LCM of the task periods can be a very large. For example if two of the task periods are chosen as a prime number, the LCM can be large.

4.3 The EDFIS algorithm

The factor k can be determined by using the scaled processor capacity line, the workload and the processor demand function. With these functions it is possible to determine the $\max\left(\frac{H(t)}{t}\right)$, within the presented boundaries of t. Using this maximum, the scaling is determined by $k = \max\left(\frac{H(t)}{t}\right)$.

The analysis used to find the *optimal scaling factor* for a task set is based on the necessary and sufficient feasibility algorithm for EDFI, as presented in subsection 3.8.2. The EDFIS analysis is searching for the

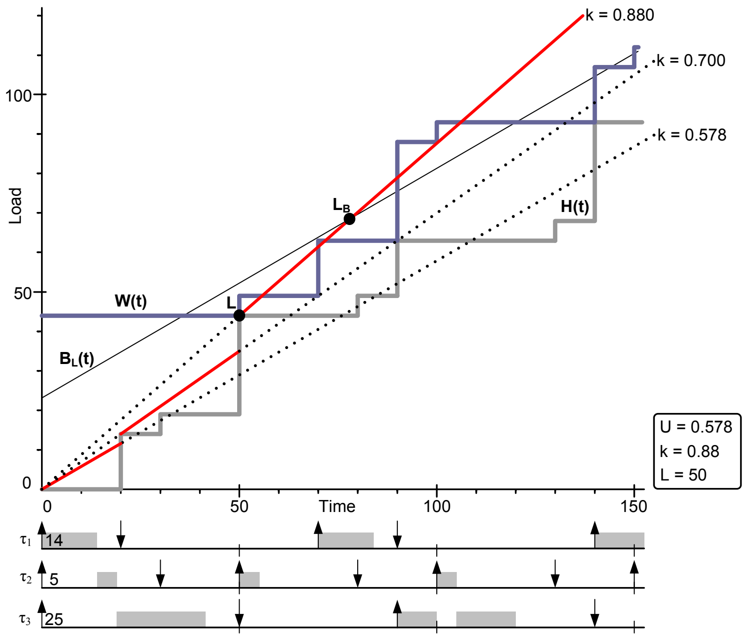


Figure 4.2: Example analysis of task set Γ_3

Table 4.1: Example task set Γ_3

Γ_3	D_i	T_i	C_i
τ_1	20	70	14
τ_2	30	50	5
τ_2	50	90	25

maximum scaling factor within the defined boundaries for the time variable t. The timestamps at which events occur are evaluated. The examined events are the releases and deadlines of tasks. The events are ordered according to their time t and their type, the event with the smallest t is processed first, if two events have the same time and different types, the deadline event is processed first. Deadline events are used to update the processor demand information and release events are used for updating the workload information.

An example of an EDFIS analysis for task set Γ_3 in Table 4.1 is shown in Figure 4.2. Initially the scaling factor k is chosen equal to U, which is 0.578 in the given case. The Baruah line $B_L(t)$ is running parallel to the initial scaling line, which means that they will never intersect. Note that the Baruah line has slope U and the offset $\sum_{i=1}^{n}\left(1-\frac{D_i}{T_i}\right)\cdot C_i = 23\frac{1}{9}$ on the y-axis. The two search boundaries left are the LCM of the periods and the first idle time. The LCM of the periods in this task set is t = 3150 and the first idle time for the scaled task set is found when the workload function is equal to the scaled processor capacity $p(t)$. At t is 20 a deadline event occurs, which introduces a new $k = \frac{H(t)}{t} = \frac{14}{20} = 0.7$. Note that at t is 20 in Figure 4.2 the old processor capacity line becomes a dotted line and the new line becomes solid. The new scaling factor gives the processor capacity a slope larger than U, so it intersects the Baruah line. The intersection will happen at t = 173.5, which can be calculated using equation 4.7. Continuing the examination of events, the deadline at t is 50 increases k to $\frac{44}{50} = 0.88$. With the new scaling factor the Baruah point L_B is located at t is 76.5. Furthermore the point L is found at t is 50 because the workload function touches the processor demand function and the scaled processor capacity. This means that idle point has passed and the optimal scaling factor has been found.

The pseudo code for the EDFIS algorithm is presented in *EDFI Scalability analysis*. In the main loop the search for the maximum scaling factor is performed by examining the events occurring for the task set. Each iteration a tuple with the event information is fetched with the function *GetNextEvent*. The tuple contains a t with the *time* at which the event occurs, a flag with the *type* of event, either deadline or release, and a C containing the *maximum computation time* of the event.

When the event is of type *deadline*, the processor demand and the blocking are checked, to see if the scaled processor capacity, $t \cdot k$, can fulfill the demand. If the processor capacity is to low, the scaling is decreased, so a larger k is chosen. When a larger scaling factor is chosen the *Baruah point* is recalculated. Initially the Baruah point $\tilde{L}_B$ of the scaled task set is chosen as infinity, because k is chosen equal to U, which makes that both lines never intersect.

The workload function is examined when a new job is *released*. When the workload function is smaller than the scaled processor capacity ($t \cdot k$), all load is resolved by the scaled processor. As with the EDFI feasibility algorithm, no larger k = $\max \frac{H(t)}{t}$ can be found after this point.

The last statement of the main loop performs a check, to see if t is larger than the Baruah point. When t is larger, the optimal scaling is found.

```
Algorithm EDFI Scalability analysis
    $H = 0$; // Processor demand
    $W = 0$; // Workload
    $k = U$; // Scaling factor
    $schedulable = unknown;$
    $\tilde{L}_B = \infty;$
    while $(schedulable == unknown)$ {
      $(t, flag, C) = GetNextEvent;$
      case $(flag)$ {
        deadline:
          $H = H + C;$
          $C_b = \max_j \{C_j | \Delta_j \leq t < D_j\};$
          if $(H + C_b > t)$
            $schedulable = no;$
          if $(\frac{H+C_b}{t} > k)$ {
            $k = \frac{H+C_b}{t};$
            if $(U < k)$
              $\tilde{L}_B = \frac{\sum_{i=1}^{n}(1-\frac{D_i}{T_i})\cdot C_i + C_b}{k-U};$
            else
              $\tilde{L}_B = \infty;$
          }
        release:
          if $(t > 0 \wedge W \leq (t \cdot k))$
            $schedulable = scalable;$
          $W = W + C;$
      }
      if $((schedulable == unknown) \wedge (t \geq \tilde{L}_B))$ $schedulable = scalable;$
  }
```

As presented in the previous section the EDFIS algorithm tries to scale the utilization to 1. If the utilization can be scaled to 1, the scaled Baruah point does not exist and there is no idle time. However the LCM of the task periods is always available and guarantees that the *algorithm is finite*. In the worst-case situation the presented algorithm has to examine all events until the LCM of the task periods. When the task periods are chosen as prime numbers the number of examined events is exponentially related to the number of tasks in the task set. Therefore in the worst-case situation the complexity of the algorithm is exponentially related to number of tasks in the task set. In general the algorithm can be performed in *pseudo polynomial time*. This because the deadlines of tasks and the blocking of resources cause the required processor capacity $p(t)$ to be larger than the utilization of the task set. Note that the algorithm increases k if more processor capacity is required. It is possible that the required processor capacity causes k to become larger than 1, increasing the frequency of the processor instead of reducing it.

Chapter 5

Wireless sensor node

A wireless sensor node is a device equipped with an OS. The hardware platform of the wireless sensor node, as used at the University of Twente, will be discussed in section 5.1. Following the AmbientRT OS will be discussed in section 5.2. This discussion includes the real-time scheduler, the data manager, the modules, the dynamic reconfiguration and the task and data definition in AmbientRT.

5.1 Hardware platform

The sensor node used in this paper is the μnode v2.0 of Ambient Systems, see Figure 5.1. At the moment of writing this is the newest available sensor node. Below the main components of the sensor node will be discussed.

At the hart of the sensor nodes is a Texas Instruments MSP430 microcontroller. This low power microcontroller has a 16-bit RISC CPU. The MSP430F1611 [30] version of the microcontroller is used. This version provides 48 kB flash memory and 10240 B RAM. Much integrated functionality is provided, including:

- 16-bit Timer A with three capture compare registers
- 16-bit Timer B with seven capture compare registers
- 16-bit Hardware multiplier, not integrated in the CPU
- Two Universal Synchronous/Asynchronous Receive/Transmit (USART) interfaces
- Six digital I/O interfaces
- Five power saving modes
- JTAG/debug interface
- Digitally Controlled Oscillator

The μnode v2.0 is equipped with an EEPROM to provide additional storage space. The used EEPROM is the ST M25P40 [29] 4 Mbit serial flash memory. The flash memory has a Synchronous Peripheral Interface (SPI). It is connected to the digital I/O interfaces of the MSP430.

Sensor nodes should communicate by radio, therefor the μnode v2.0 is equipped with a low power transceiver. This transceiver is able to communicate in the 868 and 915 MHz band. The communication ranges to typical 50 m indoors and 200 m outdoors. It can put up a connection with an effective symbol-rate of 50 kbps. As the

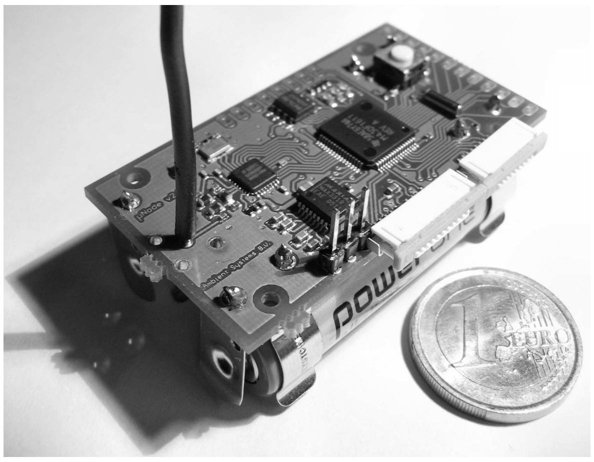


Figure 5.1: Ambient systems μnode v2.0

flash memory, this component provides a SPI interface and is connected to one of the digital I/O interfaces of the MSP430.

Beside these main components all nodes have three LEDs. On the edges of the sensor nodes additional digital I/O ports and ADCs are available, to which sensors can be connected. Some sensor nodes are equipped with a MAXIM MAX3319. This component translates the CMOS signals, from the USART interface of the MSP430, to RS232 compatible signals. This makes it possible to communicate with the node by the serial port of a PC.

5.2 AmbientRT

The lightweight OS used on the wireless sensor nodes is AmbientRT, which evolved from the Data-Centric Operating System (DCOS) [12]. AmbientRT is built as a partly platform independent OS. As Hardware Abstraction Layer(HAL), the OS uses drivers, which provide a general interface to the peripherals. User applications can perform system calls through the functions specified in the AmbientRT library. This library contains general functions like: ”malloc”, ”free” and ”printf”.

The data-centric OS has been developed with four main requirements: *real-time scheduling*, a *data-centric architecture*, *execution of modules* and the possibility for *dynamic reconfiguration*. The first two subsections explain the real-time scheduling and the data-centric architecture of AmbientRT. Subsection 5.2.3 explains how AmbientRT executes modules and how it can be reconfigured. The last subsection explains how tasks and data are defined in AmbientRT.

5.2.1 Real-time scheduling

The real-time scheduling algorithm used for AmbientRT is EDFI, as discussed in subsection 3.8. Since the OS provides resources, the transaction model is chosen as resource management protocol. The EDFI scheduling algorithm with transactions, make the OS deadlock free, as explained in subsection 3.8. Since the scheduler needs to make the scheduling decisions and needs to manage the resources, the scheduler is invoked when an interrupt occurs or when a task finishes. Furthermore the scheduler is released at a frequency of 16 Hz to examine the state of the system.

To perform the EDFI scheduling algorithm the OS needs information about the tasks that are going to be executed. Therefor the OS needs a structure *OS_Task_Specification* for each task, as shown in Figure 5.2. This structure contains the constant basic values that are needed to perform the EDFI scheduling. Because this is

```
typedef struct
{
  task_func entrypoint;     // Location of task in memory
  word      deadline;
  dword     period;
  word      cputime;        // Maximum computation time
  word      *dataspec;      // List with used data types
  byte      *resources;     // List with used resources
  byte      *dependencies;
  word      heapspace;
  void      *datasegment;
  char      *id;            // Name of the task
}OS_Task_Specification;
```

Figure 5.2: OS_Task_Specification structure (constant)

```
typedef struct OS_Task_Entry
{
  word        absolute;  // Relative time left till deadline
  word        delta;     // Inherited deadline
  ptrval      stackbase;
  word        bits;
  word        mask;      // Events to react on
  word        event;
  OS_Task_Specification *spec;
  struct OS_Task_Entry *next; // Preemption or Release queue
}OS_Task_Entry;
```

Figure 5.3: OS_Task_Entry structure (volatile)

an implementation the C_i, D_i and T_i are coupled to a 32768 Hz clock. To get the duration of these values in seconds, they should be multiplied with $\frac{1}{32768}$. Since the *deadline* and the *cputime* are 16-bit values, they can represent a maximum of 2 seconds. The *period* is represented as a 32-bit value, which enables it to represent a little more then 36 hours.

Beside the deadline, period and cputime, the structure contains the dataspec, resources and dependencies. The *dataspec* and *resources* pointers, both point to a list that contains information about the used data types or resources, respectively. Resources are represented with an 8-bit value, where the first bit indicates read or write access. The 7 remaining bits make it possible to represent 128 different resources. Data types are represented with 16-bit values where one bit is reserved to denote exclusive access, so 32768 different data types can be defined. The *dependencies* pointer, points to a list which contains pointers to tasks with whom resources or data are shared. When using transactions the scheduler uses this list for the offline calculation of the inherited deadlines.

Tasks also have a part of there description in volatile memory, in an OS_Task_Entry structure, see Figure 5.3. This structure contains information about the dynamic priority of the task and the actual location of the task in the memory.

The scheduler model used in AmbientRT is similar to the transaction system, as discussed in section 3.3. The *preemption stack* and the *release queue* are realized by the ”next” pointer in the OS_Task_Entry structure, so only the head of the list has to be stored. The EDFI algorithm causes these lists to be ordered according to deadline. The transition of tasks from the waiting queue to the release queue happens by *events*. An event is generated by publishing a data type, which will be discussed in more detail the next section.

The word *absolute* in the OS_Task_Entry represents the time a task has left till its deadline. To avoid updating these times for all tasks in the queues, only the head of the queue contains the absolute time and the other tasks in the queue store the time relative to its successors. For example when there are three items in a queue with 4, 6 and 9 ticks left till their deadline, the first item will have the absolute value 4. The second has 2 stored in absolute, so with the value of the first item 6 can be found again. The third item stores 3, with the value 2 of the second item and 4 of the first, it can calculate that it has 9 ticks left till its deadline. The advantage of this method is that the remaining time for all tasks in the preemption stack and the release queue can be decreased by only subtracting the value from the absolute value of the first item. Only when the first item has zero as absolute value, the absolute value of the second item should be decremented and the first item should be removed. The waiting queue does not use an ordered list structure, a task is made waiting by setting the waiting bit in its ”mask”.

Beside the preemption stack AmbientRT also maintains a stack with the contexts of the preempted tasks. When a new task is started, the general purpose registers of the previous task are stored on the stack. The task also stores its exit routine and its variables on the stack. The OS_Task_Entry of a task contains a ”stackbase” pointer that points to the location on the stack where a pointer to the exit routine is stored. Directly below the stackbase the registers of the previous task are stored and above the ”stackbase” the variables of the task can be stored.

A problem in AmbientRT arises with the three groups that can publish data types: *timers*, *interrupts* and *tasks*. Since a task can be released on the occurrence of an event and an event can also be generated by for example the interrupt of the radio or a button, the release of tasks is not strictly periodic. It can be said that these tasks behave like *sporadic tasks*. Another case are tasks subscribed to events published by other tasks. This creates *precedence constraints*, which are not accounted for in theory (section 3.2). Precedence constraints make the feasibility analyses, as presented in chapter 3, invalid for the scheduler in AmbientRT. In [12] a solutions is proposed, where the acyclic dependency graph of the task set, is transformed in a schedule that satisfies the EDFI assumptions. The theoretical verification and implementation of the solution is considered future work.

5.2.2 Data manager

In the data-centric architecture of AmbientRT, data and the validity of data are most important. The data manager in AmbientRT manages the data types. Tasks can *publish* a data type and optionally write data to the data type. The data manager maintains a subscriber table, which contains for each data type the task that is *subscribed* to it. In case a data type is published, the data manager puts all subscribed tasks, from the waiting queue in the release queue. The combination of the data manager and the real-time scheduler results in the *data-centric scheduler* of AmbientRT.

A data type is defined by the DMGR_TypeEntry structure, as shown in Figure 5.4. The *size* defines the number of words allocated for this data type and *name* contains a unique name. When a data type is published, the data manager searches the subscription table. This table is an array of DMGR_Subscriber structures (Figure 5.5). The *offset*, of the DMGR_TypeEntry structure points to the row in the descriptor table where the first subscription can be found, there will be *count* subscriptions for the data type. The DMGR_Subscriber structures contain an *index*, which points to the subscribed task. The *ormask* contains one bit set to 1. When publishing an event, a logic ”and” is performed between the ormask of the DMGR_TypeEntry and the *mask* of the OS_Task_Entry (Figure 5.3). At run time the 15 least significant bits in the mask can be altered. A task will respond to an event, when the logic and between the ormask an the mask is larger than zero. Note that since the 15 bits of the mask can correspond with one resource each, each task can be subscribed to 15 events.

The OS_Task_Specification descriptor, see Figure 5.2, comes with a list of data types, subscribed to or published by the task. Since tasks can share data types, mutual exclusive access to data types should be provided. This is achieved by treating data types as resources. Mutual exclusive access to resources can be achieved by temporary increasing the priority of the task, which is defined by Δ. Since it is possible to have data types that occupy 0 words of RAM and only trigger the release of tasks, interrupts also trigger tasks by data types. For example, when a task is subscribed to a timer, the timer interrupt will be translated to the publication of the timer data type. This way the data manager, manages all task releases.

5.2.3 Modules and dynamic reconfiguration

The AmbientRT OS uses modules to enable efficient reconfiguration. Efficiency can be gained by sending the node a module that contains the new tasks, instead of a module that contains the whole task set including the

```
typedef struct
{
  word offset;   // Row of first subscriber in table
  byte count;    // Number of subscribed tasks
  byte size;     // Number of words in memory
  byte name[4];  // Unique name
}DMGR_TypeEntry;
```

Figure 5.4: DMGR_TypeEntry structure

```
typedef struct
{
  word index;   // Index of subscribed task
  word ormask;
}DMGR_Subscriber;
```

Figure 5.5: DMGR_Subscriber structure

new tasks. This also makes it easier to distribute certain applications to a limited number of nodes, which offers the opportunity to create a heterogeneous network.

The transferred modules contain information about the tasks and the used data types. When a module is loaded, the current task set and data types need to be merged with the ones in the module. This discussion is outside the scope of this report, detailed information can be found in [12].

5.2.4 Tasks and data

In AmbientRT data is defined and tasks are developed separate from the OS. To combine the three of them, the tasks and the data and their relations should be defined in a Data Specification File (DSF). Offline the DSF file is used to verify the feasibility of the task set. Furthermore the tasks are written in C code. The main file should contain an *usermain* function that initializes the desired resources and inserts the desired task. Since AmbientRT is reconfigurable, tasks can be inserted or removed at run time with system calls. Additional information about creating task sets for AmbientRT can be found in Appendix A and [1].

Chapter 6

Implementation

This chapter will discuss the implementation of the scheduling algorithms and their extensions. The discussed implementations are extensions of AmbientRT, which is discussed in section 5.2. First the implementation of the scheduler with the resource policies and task admission will be discussed. Section 6.2 explains the implementation of the feasibility algorithms. In section 6.3 the EDFIS implementation issues are highlighted. The last section of this chapter, section 6.4, discusses TSS.

6.1 Scheduler design

The AmbientRT OS is already providing an EDFI scheduler. As mentioned in subsection 5.2.1 the provided scheduler performs EDFI scheduling with transactions. The scheduler is designed according to the transaction system. The scheduler maintains a list with active tasks, where the tasks provide their scheduling parameters.

To extend the scheduler with RM and DM scheduling, the available transaction system framework is used. At compile time the desired preemption condition can be chosen. As needed in the transaction system, the function to order the queues is chosen accordingly to the scheduling algorithm.

The remaining part of this section starts with a discussion about the implementation of the transaction resource policy, followed by a discussion about the implementation of the NCS resource policy. In the last subsection the implementation of task admission and removal is discussed.

6.1.1 Transactions

AmbientRT provides the possibility to insert and remove tasks from the task set at run time. To keep the task information valid, AmbientRT updates the Δ values of the tasks when a task is inserted or removed. To perform a quick update, the tasks are equipped with a dependency list, which contains the tasks with whom resources are shared, even tasks that are not running. This list is stored as constant in the flash memory. When a new task is inserted in the running task set, the scheduler checks this list for all the running tasks, to determine the inherited deadlines.

6.1.2 Nested Critical Sections

When NCSs are used, the Δ of a task can change for each resource claimed or released. In AmbientRT the new delta value is fetched from a list with resources. As in subsection 3.4, each resource has a read and a write floor. When new tasks are *inserted* in the task set, first the possible new resources are added and the read and write floors of the used resources are updated. When a task is *removed* from the task set, the read and write

```
typedef struct
{
  word resource;    // Identifier of resource or data type
  word readfloor;   // Highest priority of readers
  word writefloor;  // Highest priority of writers
}OS_Resource_Entry;
```

Figure 6.1: OS_Resource_Entry structure

floors are corrected. When these floors have a value equal to the deadline of the removed task, the running task set has to be searched to determine the new read or write floor. When the value of the floors is smaller than the deadline of the task, the read or write floor stays valid.

In AmbientRT data types and resources are separated, when delta levels have to be calculated data types are treated as resources. Therefore a data type is *transformed* to a resource in the scheduler and added to the resource list. This is done by setting one of the most significant bits of the data type identifier. Since the most significant bit is already used to denote exclusive access, 14 bits are left to identify 16384 different data types.

In the scheduler as much as possible information is kept as constant in memory, to save space in the RAM. Since the resources have read and write floors that change, they have to be stored in volatile memory. The used OS_Resource_Entry structure is shown in Figure 6.1.

Tasks may claim resources and data types, to enter a critical section. This should be done by system calls. Claiming a resource or a data type is done with the system calls:

int	*claim(HDEV handle,byte accessType);*	// Claim resource
int	*claim_data(word handle,byte accesType);*	// Claim data type

The *handle* is the name of the resource or data type. The *accessType* is READ or EXCLUSIVE, to denote read or write access to the resource. The NCS usage is listed in the DSF. To translate and validate the NCS usage pattern for applications requires resources, which are typically scarce in sensor nodes. Therefore the OS expects applications to use the resources and data types as declared in the DSF, to maintain the feasibility of the schedule. When a user wants to release a resource or data type, the most recently entered NCS is left. Note that, when the inherited deadline of the task is decreased by releasing a resource, the scheduler should be invoked. An NCS can be left by calling the function:

int *release();* // Leave last entered NCS

The structure used for an NCS is depicted in Figure 6.2. Each OS_Task_Entry, see Figure 5.3, is extended with a pointer to a linked list of OS_Scheduler_NCS structures. When an NCS is *entered* a new OS_Scheduler_NCS is made, the current Δ is stored in *old_delta*, the structure is placed at the front of the list and the current Δ is set to the inherited priority. The inherited priority is found in the resource list. The entrance of an NCS is of complexity *O(n)*, where *n* is the number of resources in the resource list of the scheduler. When an NCS is *left*, the Δ of the task is set to the old_delta of the structure and the structure is removed from the head of the linked list. The leave operation is of constant time, which makes the complexity *O(1)*.

Note that the memory usage of this *solution is minimal*, since the memory for the NCS is only assigned when needed. In the *worst-case situation* an OS_Scheduler_NCS structure has to be allocated for each resource that is only written and for each NCS in which a resource is read. This because multiple tasks can read a certain resource at the same time without inheriting a priority. As alternative solution predefined structures for the usage of NCSs could partly be stored as a constant for each task. Still a list containing the inherited deadlines for the NCSs should be stored in volatile memory, since these are determined at run time. The worst-case memory consumption of this alternative is larger compared to the implemented solution.

```
typedef struct OS_Scheduler_NCS
{
  word old_delta;
  struct OS_Scheduler_NCS *next;
}OS_Scheduler_NCS;
```

Figure 6.2: OS_Scheduler_NCS structure

6.1.3 Task admission and removal

In a real-time system tasks can be removed instantly, while keeping the run time stack and the release queue ordered. When a task is added to the running task set, while tasks are on the run time stack, the ordering of the run time stack can be disturbed, as explained in section 3.5.

In AmbientRT task removal can be performed instantly, by a system call. The admission of a task, while other tasks are in the release queue or on the run time stack, would require a verification of the new situation. Since this requires additional code and processing power, admission of tasks is only allowed when the run time stack and the release queue are empty. When the system call is performed to add a task, the id of the task is placed in the *insertion list*. When the scheduler detects idle time, it starts adding tasks from the insertion list to the running task set. To prevent the scheduler from stalling the system while adding new tasks, the scheduler can only add a few tasks each invocation. This maximum amount of tasks inserted at a scheduler invocation can be configured at compile time.

6.2 Feasibility analyses

The offline feasibility analysis and DSF parsing tool *Sparse* is extended with additional feasibility analyses. The tool implements the *necessary and sufficient* feasibility algorithms for the scheduling algorithms, as explained in sections 3.6, 3.7 and 3.8. The explained pseudo code of the feasibility algorithms is implemented, providing support for the NCS and transactions resource policy. How task sets can be offered to this tool for a feasibility check with a specific scheduling algorithm and resource policy, is explained in Appendix A.

6.3 EDFI and Scaling

The EDFIS analysis is explained in section 4. The CPU frequency scaling is performed to make the microcontroller run at a lower frequency, which saves energy. The analysis is implemented with the suggested ordered event queue. Nonetheless a few implementation issues arise when extending AmbientRT with EDFIS.

The analysis is not extremely lightweight, since scaling the performance to one also increases the time domain to be searched. Therefore the *moment* to perform the EDFIS analysis has to be chosen carefully, to prevent the system from stalling. Furthermore the algorithm is performed in slack time. When *task insertion* and *task removal* occurs, the EDFIS analysis is performed. In case of task insertion or removal, the frequency is restored to 4.6 MHz. When the task set is initialized, all tasks are inserted one by one. To prevent multiple invocations of the EDFIS analysis at the same time, a new analysis removes a running analysis from the stack and is only initialized when the insertion queue is empty. Furthermore the analysis is not immediately performed, it is postponed by the largest period in the task set.

The EDFSI analysis makes use of the maximum computation times C, of the tasks. Since the user does not always provide accurate values, the scheduler also examines the C for all tasks. After task execution the current duration is compared to the maximum computation time, which is updated if necessary. When the current duration of the task is larger then the provided C, the current duration is increase with 20 % to account for the maximum computation time of the task and the scheduler time needed for the task. When a new task is inserted in the systems, the system runs at full speed for $\max_i\{T_i|\tau_i \in \Gamma\}$, to gather the new C of each task. With the found values the analysis is performed.

To perform the EDFIS analysis with *nested critical sections*, information about the duration of the NCSs is needed. When the module is compiled with the option EDFIS and NCS, sparse includes additional information containing the NCS structure for each task, which is written in the flash memory. This way the EDFIS algorithm can calculate the blocking time for the tasks.

6.4 Temporal Shutdown Scheduling

Energy can be saved by shutting down the microcontroller when it is not needed. The Temporal Shutdown Scheduling (TSS), partly disables the microcontroller when there are no tasks left on the stack, so when it detects idle time. The MSP430 allows five different modes, with four low power modes. In low power mode 1, only the CPU gets disabled. Low power mode 2 and 3, disable the CPU and additional parts of the internal Digital Controlled Oscillator (DCO) and the internal clocks. Low power mode 4 also disables all internal functionality and the external clock. The MSP430 can wake up, in 6 μs, by an interrupt. More information about the low power modes can be found in [31].

The best energy savings, for AmbientRT, are achieved by entering low power mode 3, when the scheduler detects idle time. This disables the CPU, the DCO and the internal clocks. Since the DCO gets disabled, all devices driven by the internal high frequent clocks have to be disabled or attached to the external 32768 Hz crystal. Therefore some devices need to be reconfigured. For example the RS232 is attached to the external clock crystal when TSS is enabled, the communication for RS232 is performed at 9600 baud instead of the normal 115200 baud. The scheduler is already attached to the external timer and is executed with a frequency of at least 16 Hz.

Chapter 7

Performance comparisons

The research goals of this report include the comparison of the implemented alternatives. This chapter provides the performance comparisons for the three different subjects. The best way to verify the behavior of new scheduling algorithms, is by testing it with an exhaustive number of random task sets. Since it is not possible to perform such a test on the μnode v2.0 without modifying AmbientRT drastically, an alternative would be running the test on a simulated version of the MSP430 in Matlab. Because this research concerns algorithms that already proved themselves in such exhaustive tests, interesting task sets are selected and examined. The behavior of the algorithms in these tests is used to examine and verify the algorithms in AmbientRT. Buttazzo [8] provides a comparison of the EDF and RM algorithm, where the results are obtained with exhaustive tests.

As an introduction for the comparisons, this chapter starts with the measurement setup. We explain how the measurements are performed and which values are retrieved. Next, section 7.2 discusses the performance of the three scheduling algorithms. Four interesting situations are distinguished and examined in detail. In section 7.3 the difference between the two resource policies is highlighted for four different cases. Section 7.4 compares the EDFIS and TSS policies on energy efficiency in four different cases.

The subsections discussing the examined cases have a general structure. Most cases start with a paragraph about the *purpose* of the test, followed by one about the *expected* results. Next the subsection will contain paragraphs discussing the *results*, the behavior that is *observed* in the results and finally a *conclusion* of the performed test.

7.1 Measurement setup

The limited amount of RAM on the sensor nodes introduces the problem how to perform the measurements. When measuring the performance, timestamps and event information should be stored. In the case of the sensor node two possibilities have been examined, both with their drawbacks.

The first option for measuring is the *Joint Test Action Group (JTAG) interface* of the MSP430. The JTAG interface provides an interface to debug the microcontroller, with full control over it. The ALU can be stopped and examined by defining breakpoints in the debugger software, which allows an unlimited number of measurements. The first arising problem is that the timers are not stopped when the ALU is stopped by the debugger, via the JTAG interface. This is solved by extending the OS to store the timer values at a fixed place in the code, where the debugger is placing its break point. Immediately after this place in the code, the timers and interrupt flags are restored, to provide correct behavior when the microcontroller continues. Timer A is used to obtain timestamps, it is running at $\frac{1}{8}$ of the ALU clock to avoid saturation of the timer register. Timestamps and state information are stored in variables when the scheduler is invoked and finished. These variables are read when the JTAG interface interrupts the microcontroller. An unsolved problem is the time used for JTAG communication. Reading a measurement takes more than two seconds, which makes measuring time-consuming and simulation of radio communication impossible.

The second option is to *store the timestamps* in the remaining RAM of the MSP430. As stated in 5.1 the MSP430F1611 has 10 kB of RAM. The heap space of AmbientRT can be scaled to approximately 8 kB, in which the kernel and the user programs use space to store their variables. About 5900 bytes of the heap space can be used to store the measurements. The time that can be measured in this amount of RAM differs from the amount of values that are stored for each measurement and the speed of the defined timers. Note that the scheduler is invoked every time a timer causes an interrupt. Timestamps are obtained from timer A, which is configured it to run at $\frac{1}{8}$ of the ALU speed. The timer has to run at a fraction of the ALU speed to prevent saturation of its register. The storage of the measurements is performed by small parts of assembler code, which write the information to the reserved memory and update the memory pointer for the next measurement.

The slow reaction of the JTAG interface and the resetting of the timers make it impossible to measure real life situations. Therefore the measurements are performed by storing timestamps in the RAM. These values are read later on and are communicated via the serial port to a PC.

Beside communicating the results via the serial port, a few alternatives are available, which have not been examined. An option is to send blocks with measurements via the radio to another node, which in turn communicates it to the PC via the serial port. An other possibility could have been the available DMA controller in the MSP430, which can be configured to communicate results to the PC via the JTAG interface. When the microcontroller is running at a reduced speed, it is possible to use the DMA controller to transfer the memory of the MSP430 via the JTAG interface to the PC, while the ALU keeps working.

7.1.1 General measurement setup

As explained in section 5.2.1, after each interrupt or finished task, the scheduling algorithm is performed. The scheduler can preempt a running task or leave the stack in its current state. This way four different scheduler calls can be distinguished:

- Scheduler invocation by interrupt, with preemption
- Scheduler invocation by interrupt, without preemption
- Scheduler invocation by finished task, with preemption
- Scheduler invocation by finished task, without preemption

Internal the scheduler code is composed out of two parts. The first part is the part called by an interrupt or by the exit routine of a task. The second part is the actual scheduler, which possibly preempts the task on top of the stack. To check the influence of the measurements on the scheduler, the duration of the measurements is also measured. Since the interrupt and the exit routine have the same code to record the measurement, the measurement duration is equal. Figure 7.1 shows the durations of measurements. Since the measurements are performed with a clock at 4,608 MHz, the duration is also expressed in seconds.

The measurements performed for the NCSs also take time. When a resource is claimed or released, timestamps are stored. The timestamps for claiming and releasing are equal. Their duration is shown in Figure 7.2.

The structure used for the measurements is depicted in Figure 7.3. The *startOfSched*, *InterOfSched* and *endOfSched* are read from timer A, during measurement phase 1, 2 and 3, respectively. Note that timer A runs at 576 kHz, which is $\frac{1}{8}$ of the main clock. The measurement phases are shown in Figure 7.1 and 7.2. When measurements are performed for NCSs, the interOfSched and deadline variable are not used. Timer B is running at 32,768 kHz and captured in *startOfSchedB* when the scheduler is started. The scheduleType, task and deadline variable are fetched in measurement phase 3. The variable *scheduleType* records which of the four different scheduler invocations is performed. The word *task* contains the address of the task in memory, to distinguish between the different tasks. The *deadline* variable contains the time a task has left at the slow B

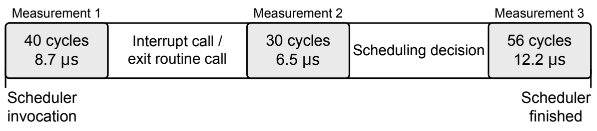


Figure 7.1: Durations of the measurements in the scheduler

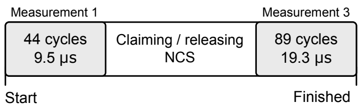


Figure 7.2: Durations of the measurements at the beginning and end of an NCS

timer when it finishes, used to calculate the lateness of the task. Typically 5900 bytes are reserved at the heap of AmbientRT to perform measurements, providing space for 421 MeasureResults structures.

7.2 DM, RM and EDF compared

To test the three different scheduling algorithms four different tests are performed. The first test in subsection 7.2.1 examines the characteristic features of the algorithms. This test should verify the behavior of the algorithms as found in the theory. The test in subsection 7.2.2 examines the behavior of the scheduling algorithms under a constant load, but with an increasing number of tasks. This should give information about the influence of the number of tasks in the task set on the scheduler. The test in subsection 7.2.3 examines the behavior of the scheduler when the utilization of the task set rises due to the addition of tasks, which classifies the relation between the used scheduler time and the utilization. In subsection 7.2.4 the best-case situation for the EDF scheduler compared to the DM and RM schedulers is examined. In addition subsection 7.2.5 examines a case in which the DM and RM schedulers have an advantage. In this test a task set is examined with short high priority tasks that are scheduled with a small lateness by DM and RM scheduling. The last subsection draws a conclusion.

Note that the tests are performed without shared resources. The used framework contains the transaction resource policy. This means that the delta levels are calculated for all tasks but are always equal to the deadline or period of the task. This way the tested algorithms behave like the DM, EDF and RM schedulers.

7.2.1 Characteristic features

The purpose of this test is to verify the proper behavior of the different scheduling algorithms. The used task set Γ_4 is shown in Table 7.1. This task set contains two tasks, chosen such that the task set is feasible with the three algorithms and has a high utilization, 0.86 in reality. In this measurement the number of preempted tasks and the lateness of the tasks is examined.

```
typedef struct{
  word  startOfSched;      // A timer @ measurement 1
  word  startOfSchedB;     // B timer @ measurement 1
  word  InterOfSched;      // A timer @ measurement 2
  word  endOfSched;        // A timer @ measurement 3
  word  scheduleType;      // Type of scheduler invocation
  word  task;
  word  deadline;          // Time left till deadline
}MeasureResults;
```

Figure 7.3: MeasureResults structure

Table 7.1: Task set Γ_4 to test characteristic features

Γ_4	D	T	C
τ_1	320	320	32
τ_2	360	360	280

It is expected that the RM and DM algorithms perform the same. Furthermore these algorithms will show a very small lateness for τ_1 and a lager lateness for task τ_2, because the higher priority task gets a low lateness by preempting τ_2 when it is released. Since the EDF algorithm minimizes the maximum lateness, it is expected that the maximum lateness of the EDF algorithm is smaller compared to the maximum lateness of DM or RM. Furthermore the smallest lateness of EDF will be larger compared to DM and RM, because task τ_1 is postponed to shorten the lateness of task τ_2. Another difference is the expected times a task is preempted. As for RM and DM, when the low priority task is running it will be preempted each time the high priority task is released. The EDF algorithm will only preempt the running task if the deadline of the task in the release queue is smaller compared to the running task. It is expected that the EDF algorithm will preempt less tasks compared to the DM and RM algorithms.

The results for this test are shown in Table 7.2. This Table shows the number of times the scheduler has preempted a task in the column *#preempted tasks*, the utilization of the scheduler in the column *U scheduler*. The times task i has been executed is given in column # τ_i, with its average lateness in column L_i and the standard deviation of L_i is in column σ_{L_i}.

As expected the DM and RM algorithms behave the same. The characteristics of the algorithms match our expectations. Task τ_1 has a very small lateness, with a small σ_{L_1}, caused by its high priority. This static high priority causes the low priority task to be preempted quite often, which is correct for the algorithm. The EDF algorithm does not preempt a task. The maximum lateness is smaller for the EDF algorithm compared to the DM and RM algorithms, but the lateness averaged over all tasks is larger.

The conclusion drawn is that the scheduling algorithms show the *expected behavior*.

Table 7.2: Test results with Γ_4

Algorithm	#preempted tasks	U scheduler	#jobs τ_1	L_1	σ_{L_1}	#jobs τ_2	L_2	σ_{L_2}
DM	85.8	0.02704	110	-296.51	1.33	98	-51.47	2.82
EDF	0	0.02721	110	-198.65	87.36	98	-70.47	5.52
RM	85.8	0.02711	110	-296.51	1.33	98	-51.74	2.71

7.2.2 Constant utilization

The purpose of this test is to examine how the scheduler behaves under a constant utilization with an increasing number of tasks. For this test three constant utilization values are used: 0.55, 0.65 and 0.80. During a test at one of the utilization values, the number of tasks in the task set is increased from 1 to 16. In the beginning of the test one task utilizes the microcontroller, according to the utilization factor of the test. At the end of the test, each task utilizes the microcontroller for $\frac{1}{16}$ of the determined utilization factor. All tasks in the task set have a period and deadline of 4096 ticks on the 32,768 kHz clock.

It is expected that the time needed by the three scheduling algorithms increases linear with the number of tasks in the task set, because the scheduler is called when a task finishes or when it is released. With the increasing number of tasks in the task set, there are more tasks calling the scheduler when they finish. A minor difference between the three algorithms is expected.

The results are shown in Figure 7.4, 7.5 and 7.6 for the utilization factors 0.55, 0.65 and 0.80, respectively. In these Figures the *a* part shows the total utilization as achieved by the number of running tasks and the scheduler. The *b* part of the Figures show the amount of time used by the schedulers.

The results partly match the expectations. The time used by the scheduler is indeed linear correlated to the number of tasks in the task set. Two things that do not match the expectations are the *fluctuation in utilization* when tasks are added to the task set and the *offset* between the utilization for the different scheduling algorithms.

The *fluctuation in utilization* is probably caused by truncation of the number of calculations that have to be performed by the tasks. The tasks contain a main loop with an inner loop. The number of times the main loop is performed, is determined by dividing a fixed number with the number of active tasks. If truncation occurs all the active tasks truncate, causing the inner loop to be skipped quite a lot of times. Despite that, the results obtained by this test are useful, because the results for the different algorithms are similar.

The *offset* between the utilization of the different schedulers is unexpected. It is probably caused by fluctuations of the temperature in the room, used for the measurements. Section 4.2.4 of the MSP430 user guide [31] mentions the temperature dependency of the DCO. The temperature could influence the measurements, because the tasks are released according to the stable external crystal and the ALU uses the internal DCO as clock. Furthermore the measurements for one scheduling algorithm are performed in series, causing the period needed for a test to be long. Since the offset is constant and the shape of the resulting graphs is equal, this does not invalidate the results.

From the measurements it can be concluded that the used *scheduling time* correlates linear with the amount of tasks. It can also be concluded that the three scheduling algorithms show *equal performances* in these tests, due to the framework of AmbientRT.

7.2.3 Constant task load

The scheduler is examined on its performance of tasks with a constant size. This test examines the behavior of the scheduler with 1 to 16 running tasks. Therefore the utilization of the task set increases with the number of added tasks. It is examined whether the increasing utilization influences the time needed by the scheduler. The deadline and period of the tasks is 4096 ticks, the maximum computation time is 200 ticks at the 32.768 kHz clock.

It is expected that the total utilization and the utilization of the scheduler rise linear. This is expected because the scheduler is called more often by the increasing number of tasks and the number of operations performed by the task set increases linear with the number of added tasks. This test shows the relation between the different scheduling algorithms and their linear rising scheduling times.

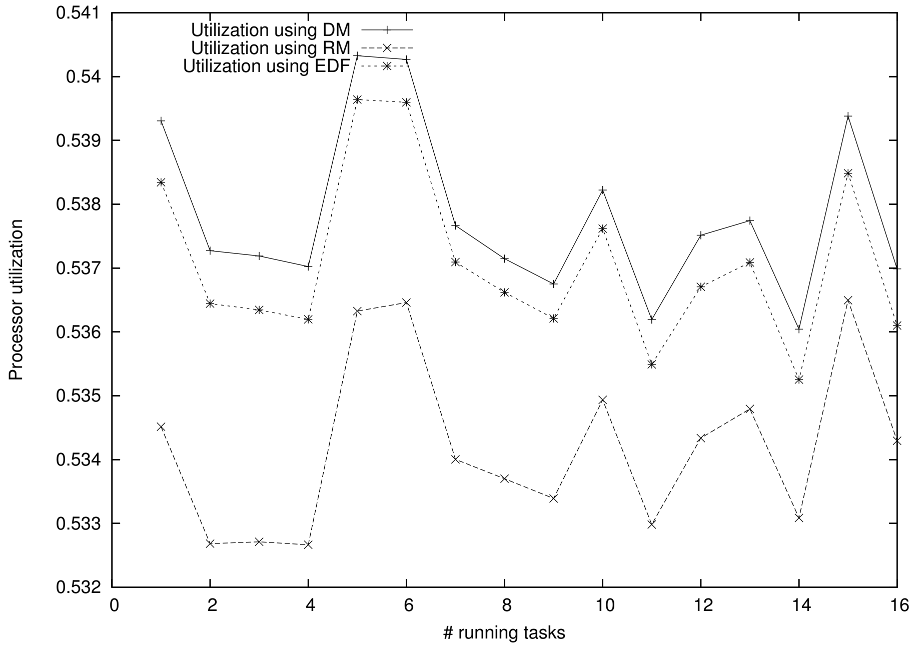
In Figure 7.7*a* the linear increasing total utilization can be seen. Figure 7.7*b* shows the linear increase in time needed for scheduling the tasks.

The results show that the total utilization and the required time for the scheduler rises linear, as expected. The time needed by the scheduler is equal as in the previous test, where the utilization was constant. Therefore the time needed for the scheduler is not related to the utilization of the task set.

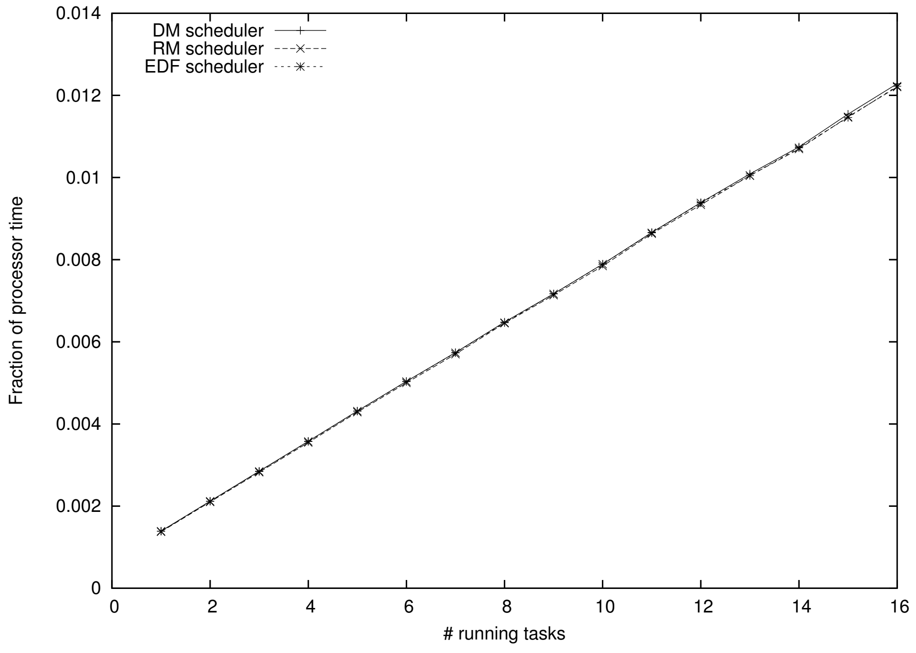
From this measurement it can be concluded that the different schedulers perform equal, as already concluded in subsection 7.2.2. In this measurement the utilization does not influence the time needed by the scheduler. Therefore the time needed by the scheduler only depends on the number of task in the task set, as found in the previous test.

7.2.4 Advantage for EDF scheduling

The purpose of this test is to examine the advantage of EDF scheduling over DM and RM scheduling. As highlighted in section 3.8, the EDF scheduler can schedule tasks with a utilization up to 1, when deadlines are equal to the periods. The RM and DM scheduler can certainly schedule task sets with U below the least upper bound ($U_{lub} \approx 0.69$). When U rises above U_{lub} the schedule has to be verified. To test the advantage of EDF above RM and DM, a schedule can be chosen in such a way that only EDF can schedule it. This kind of measurement would only show interesting results for EDF. In this case the test would not be a comparison, so mentioning that EDF performs better in this kind of situation suffices.

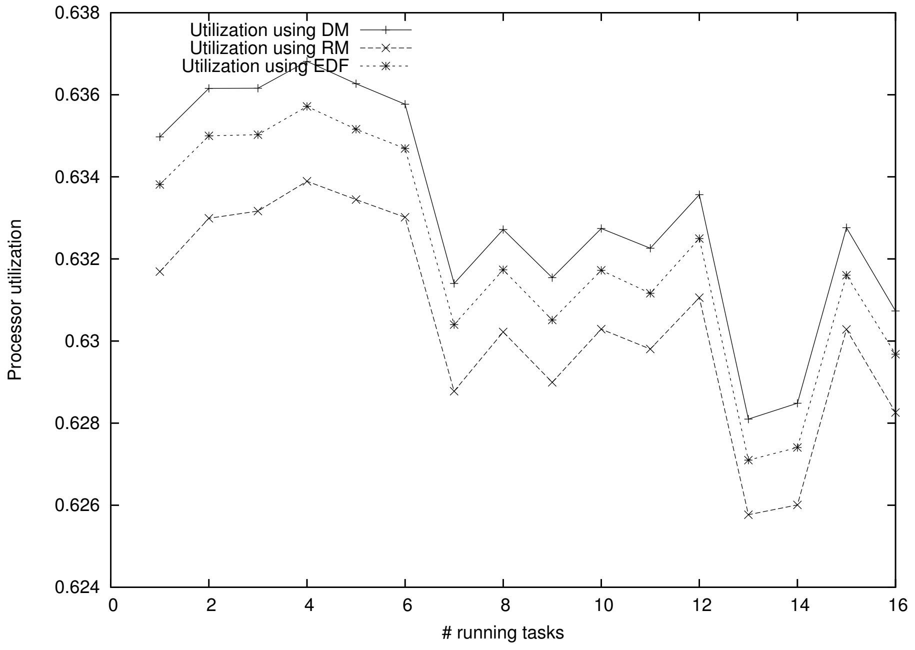


(a)

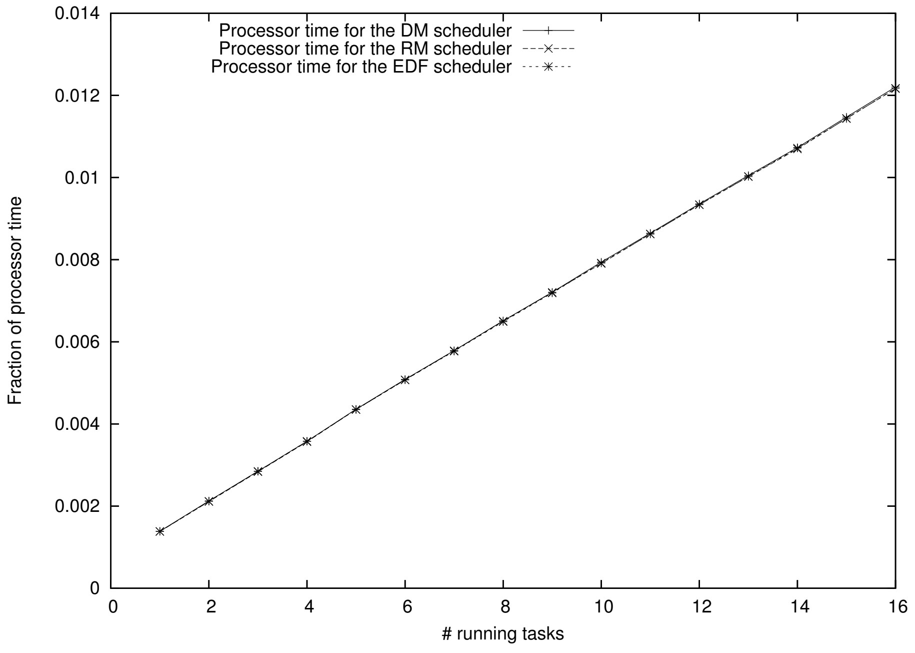


(b)

Figure 7.4: Total utilization (a) and fraction of time used by the scheduler (b), U = 0.55

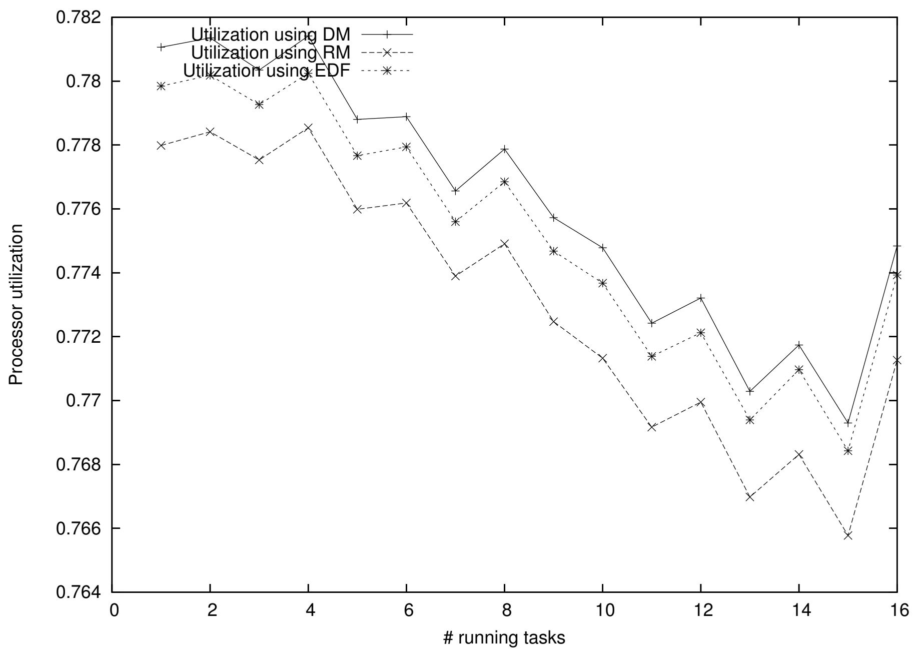


(a)

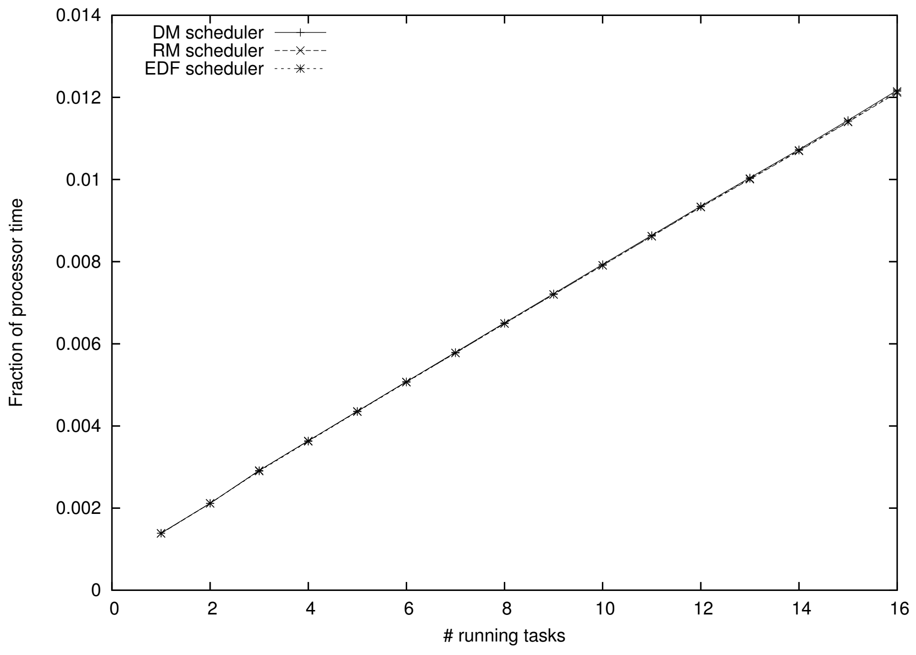


(b)

Figure 7.5: Total utilization (a) and fraction of time used by the scheduler (b), U = 0.65

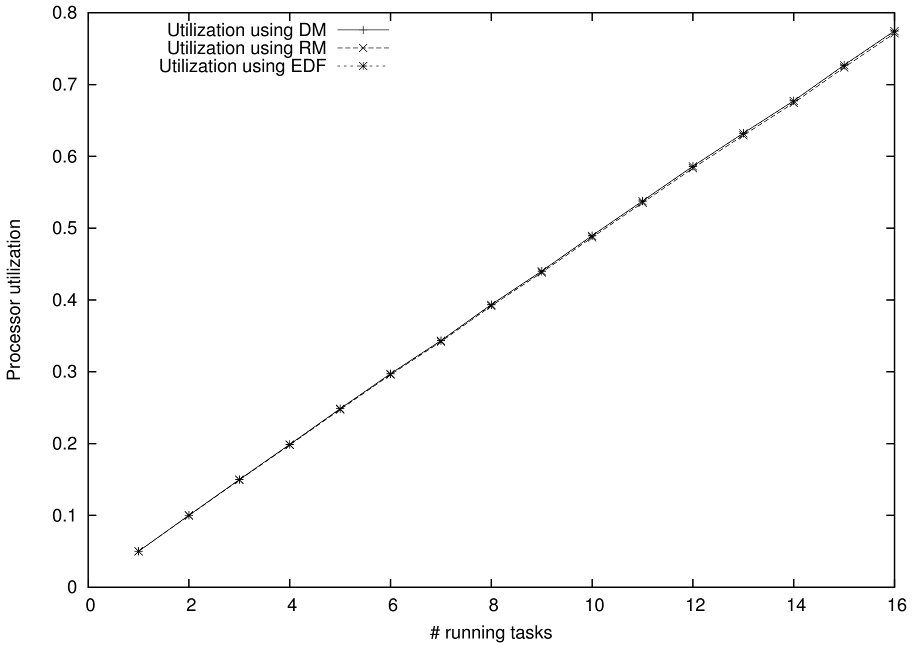


(a)

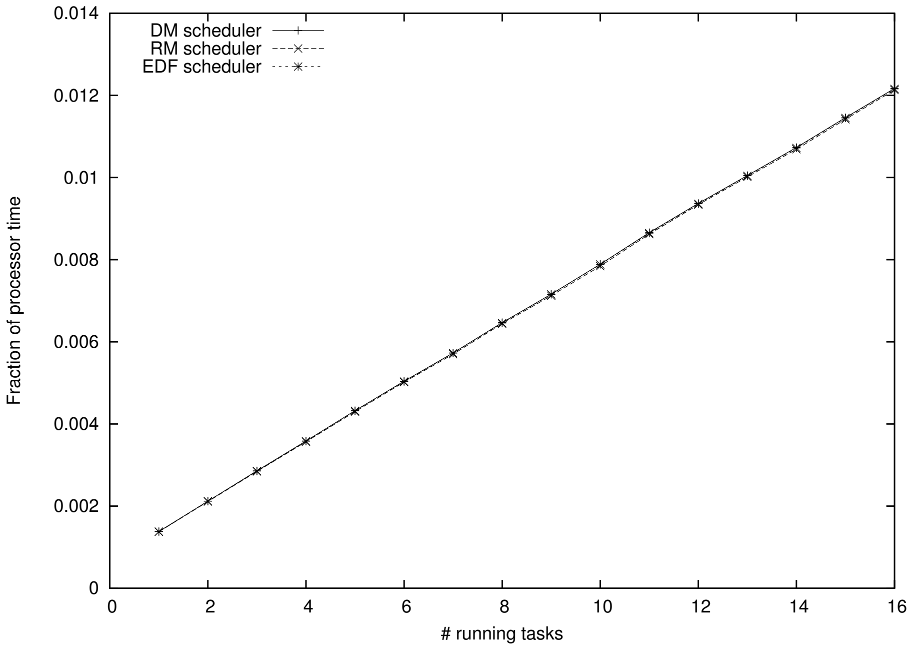


(b)

Figure 7.6: Total utilization (a) and fraction of time used by the scheduler (b), U = 0.80



(a)



(b)

Figure 7.7: Total utilization (a) and fraction of time used by the scheduler (b), U rises to 0.80

7.2.5 Advantage for RM and DM scheduling

The purpose of this test is to highlight the advantage of DM and RM scheduling above EDF scheduling. The priority rules of DM and RM scheduling make that high priority tasks are scheduled at the moment they are released. This causes the lateness L_i of high priority tasks, to be as small as possible. This is already shown in the test of section 7.2.1. To show the real latency advantage of DM and RM above EDF, a schedule is made with a high utilization. The load for the schedulers is caused by very small tasks and very long tasks. In this way a lot of tasks are preempted and queues in the scheduler are constantly filled. Note that this task set is placing an almost worst-case load on the schedulers. This load is caused by 9 tasks with short periods which are attached to the two fast timers and 7 tasks with long periods and computation times which are constantly preempted. The used task set Γ_5 is shown in Table 7.3.

Table 7.3: Task set Γ_5

Γ_5	D	T	C
τ_1	127	128	3
τ_2	128	128	3
τ_3	287	288	7
τ_4	288	288	7
τ_5	288	288	7
τ_6	2048	2048	51
τ_7	2048	2048	51
τ_8	2048	2048	51
τ_9	2048	2048	51
τ_{10}	2048	2048	51
τ_{11}	4096	4096	307
τ_{12}	4096	4096	307
τ_{13}	6144	6144	614
τ_{14}	6144	6144	614
τ_{15}	6144	6144	614
τ_{16}	6144	6144	614

The expected results are that RM and DM have a smaller lateness for their high priority tasks and a low standard deviation σ_{L_i}, compared to the EDF scheduler. It is also expected that EDF scheduling will show a larger average lateness with a larger average standard deviation of the lateness, compared to DM and RM scheduling. Furthermore the total utilization of the EDF scheduler will be lower, compared to DM and RM, because the algorithm preempts less tasks.

Table 7.4 shows the results of the test. The test results are composed of 10 measurements, which were performed with an interval of approximately one minute. The three right most columns give the results for the different scheduling algorithms. The first row presents the utilization U, the second row the number of preempted task *#preempted tasks* and the third row the fraction of time used by the scheduler U_{sched}. Furthermore, for the two high priority tasks the times task *i* has been executed is shown in row *#jobs* τ_i, the lateness in row L_i and the standard deviation of the lateness in row σ_{L_i}. The three last rows show: the total number of executed jobs *total #jobs*, the average lateness of all tasks *Average L* and the average standard deviation of the lateness for all the tasks in the task set *Average* σ_{L_i}.

The result differ from the expectation. The utilization and the number of preempted tasks with the EDF algorithm are only a little bit smaller compared to the other two algorithms. The lateness of the high priority tasks is smaller for the EDF algorithm than for the DM and RM scheduling algorithms. Both are caused by the deadlines of the low priority tasks, which are this far in to the future that EDF preempts them. This can also be the reason that EDF shows only a minor higher average L_i and σ_{L_i}. With the task set applying a heavy load on the schedulers, the fraction of time used by the schedulers is around 0.116.

Table 7.4: Results with task set Γ_5

	DM	EDF	RM
U	0.8546	0.8487	0.8636
#preempted tasks	83.4	82.1	84
U$_{sched}$	0.1146	0.1155	0.1166
#jobs τ_1	73	72.7	72.7
L_1	-120.31	-116.15	-116.34
σ_{L_1}	2.4433	2.1221	2.6143
#jobs τ_2	72.7	73.3	73
L_2	-115.15	-119.49	-118.41
σ_{L_2}	2.4176	2.2410	2.4801
Total #jobs	277	276.7	276.7
Average L all τ	-1794.10	-1817.38	-1782.62
Average σ_{L_i}	33.63	38.09	34.76

The conclusion can be drawn that the advantages for DM and RM do not show up in this test. It is possible that the chosen task set is not optimal for the two algorithms, but it is more plausible that their advantage is not that big. This is also confirmed by Buttazzo in [8], where random task sets consisting of 10 tasks are compared on their lateness. Buttazzo shows that with a utilization larger than 0.7, EDF has a small average lateness and DM and RM have a smaller lateness for their high priority tasks. The results show that the fraction of time used by the schedulers is around 0.116, leaving time to execute task sets with at most a utilization of 0.884.

7.2.6 Conclusion

In the previous tests a small subset of the possible task sets has been examined. Although no exhaustive average task set test has been performed a few conclusions can be drawn. The tests performed in subsection 7.2.3 and 7.2.2 show that the fraction of time used by the scheduler is linear correlated to the number of tasks in the task set. Furthermore, when the results of these tests are compared, it can be seen that the utilization of the scheduler is independent of the total utilization. Note that this is only valid in the situation where one timer is used. The last test examines a task set, which requires a close to worst-case amount of time for the scheduler, in this case 0.884 of the processor time is available for the task set. The results also show that the schedulers perform quite similar, when used in AmbientRT.

The last two subsections try to highlight the advantages of EDF, DM and RM scheduling. EDF scheduling is able to schedule task sets that cannot be scheduled by DM and RM. Furthermore the advantage of a small lateness for high priority tasks with the DM and RM scheduling algorithm is smaller than expected. The last test shows that the lateness and the standard deviation of the lateness are almost equal for the three scheduling algorithms. Therefore no real advantage of DM and RM over EDF can be found. When the smallest possible lateness for high priority tasks is not mandatory, EDF scheduling seems to be the best scheduling solution.

7.3 Transactions and NCS compared

In this section cases are selected to highlight the difference between the transaction and NCS resource policies. The tests are performed for multiple scheduling algorithms, but the main focus is on the performance difference between the two resource policies.

This section starts, as the previous section, with examining how the scheduler behaves when executing a task set with a constant utilization. This should show the relation between the number of resources in the task set and the time needed by the scheduler for the resource policy. In subsection 7.3.2 the time used by the scheduler is examined for tasks with fixed sizes. The relation between the utilization and the time needed for the resource policy is examined in this test. Next, subsection 7.3.3 examines a case in which the NCS resource policy has an advantage over the transactions resource policy. Followed by subsection 7.3.4, examining the time needed for removing and adding tasks. Finally a conclusion is drawn from the gathered results.

7.3.1 Constant utilization

This test examines the relation between the computation time used by the different resource policies and the number of resources in the task set. As in subsection 7.2.2, the number of tasks in the task set is increased to 16, but the utilization of the task set is kept constant. Three constant values for the utilization are selected: 0.55, 0.65 and 0.80. All tasks have 2 resources and 4 data types assigned, in total there are 5 different resources and 19 different data types in the task set. The priority of the tasks in the task set decreases, so the last task added to the task set has the lowest priority. The periods and deadlines of the tasks range from 4079 to 4094. The data types used by the tasks overlap, so most tasks get a new inherited deadline for each data type they claim. Furthermore two different cases are examined for the NCS resource policy. In the first case the resource policy releases all resources at once when the task is finished. In the second case the task releases the resources one by one, so its priority is restored after each release.

It is expected that the transaction resource policy will take the least time. This because the inherited deadlines are calculated when the tasks are inserted in the task set, so no online calculations are performed. Furthermore the time needed for the NCS resource policy is much larger, because the inherited deadlines have to be calculated when an NCS is entered. When the tasks release all resources independently, the required time is even larger due to additional time needed for the scheduler invocations after each release resource.

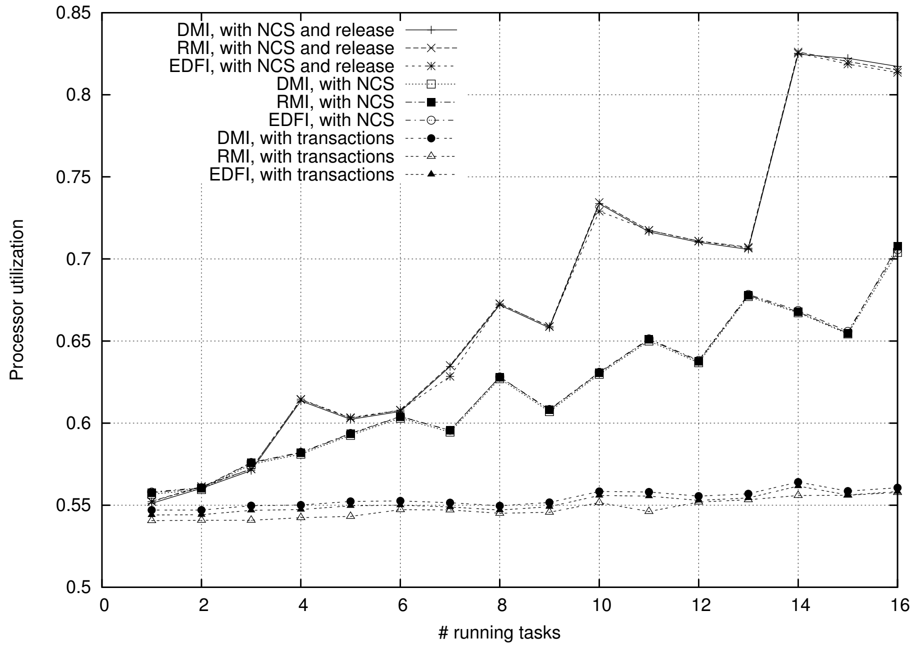
The results for this test are shown in the Figures 7.8, 7.9 and 7.10. These Figures contain the three different examined cases: *NCS with resource release*, *NCSs* and *transactions*. Furthermore the measurements are performed for the three scheduling algorithms, already compared in section 7.2. The *a* parts of the Figures show the total utilization and the *b* parts of the Figures show the fraction of time used by the scheduler.

The first thing seen in the Figures is that the three different scheduling algorithms perform equal, as in the previous section. More interesting is the fraction of time used by the scheduler when transactions are used, this is equal to the case where no resources are available in the system, see subsection 7.2.2. Therefore it can be concluded that the time used by the transaction resource policy is independent of the number of resources and data types in the system. The fraction of time used by the scheduler with the NCS resource policy is equal for the three different utilization factors, so there is no influence of the utilization. As expected, the independent releases of the resources take more time compared to releasing all resources at once, because the scheduler is invoked at each release. The number of resources and releases in the system increases linear, therefore it can be concluded that the releasing of a resource takes a constant amount of time. The NCS resource policy requires quite an amount of time, it is expected that predefining an NCS structure for each resource considerably decreases the required time, this optimization is left as future work. The fluctuation in the total utilization is caused by the limited amount of timestamps that can be recorded, this causes differences in the amount of idle time recorded for the measurements. Note that the fluctuation is equal for the three different scheduling algorithms at the three examined utilization factors.

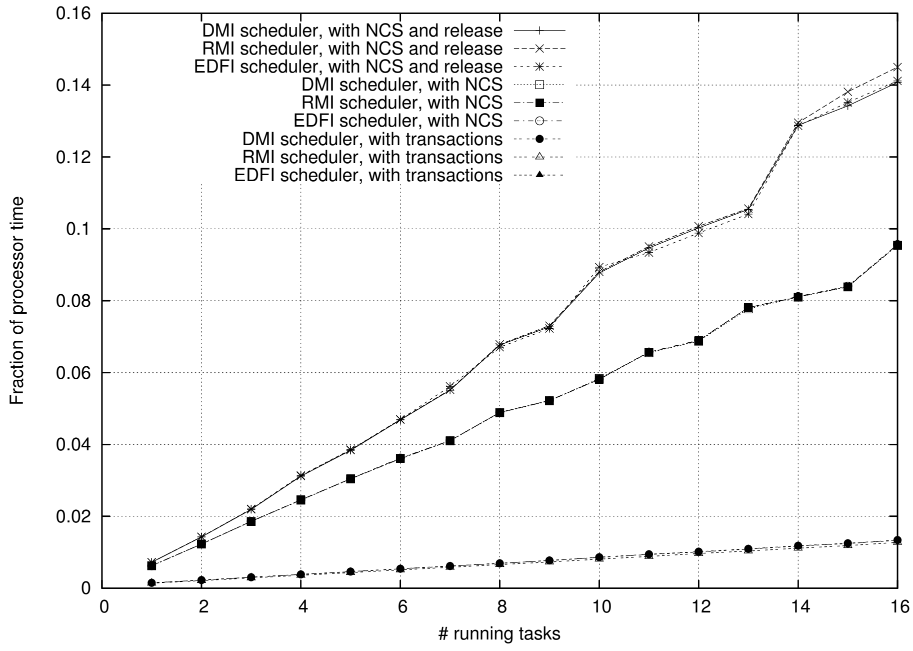
From the measurements it can be concluded that transactions take a constant amount of time, independent of the amount of resources in the system. When NCSs are used, the time to calculate the inherited deadlines increases linearly with the number of resources in the system. Furthermore, when releasing the resources, the time needed by the NCS resource policy increases even faster with an increasing number of resources in the system.

7.3.2 Constant task load

The purpose of this test is to measure the relation between the time needed by the resource policies and the utilization of the task set. As in the previous section, the test starts with one task in the task set and the number of tasks is increased to 16. Each task has 2 resources and 4 data types assigned and there are in total 5 different

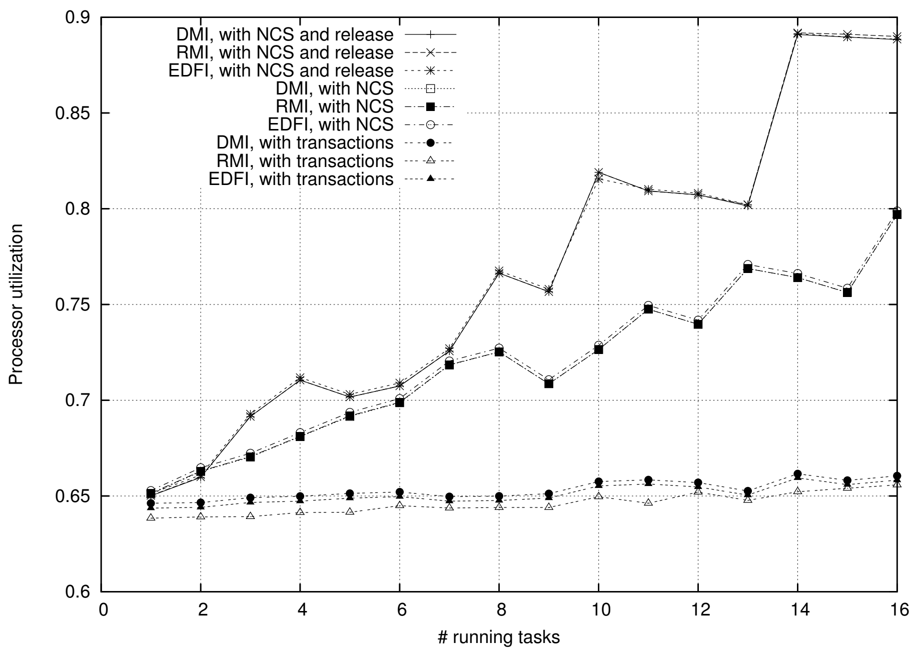


(a)

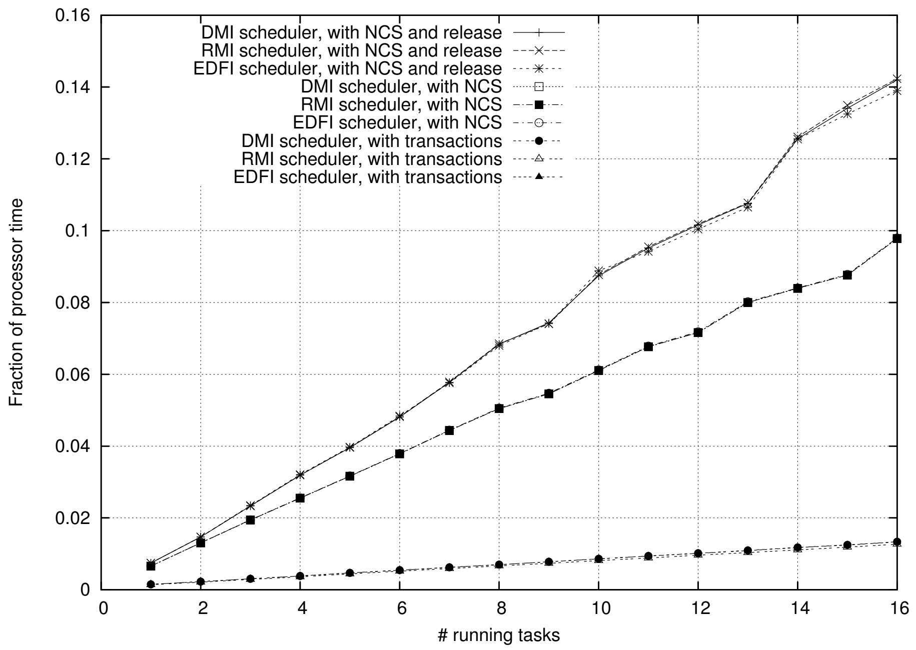


(b)

Figure 7.8: Total utilization (a) and fraction of time used by the scheduler (b), U = 0.55

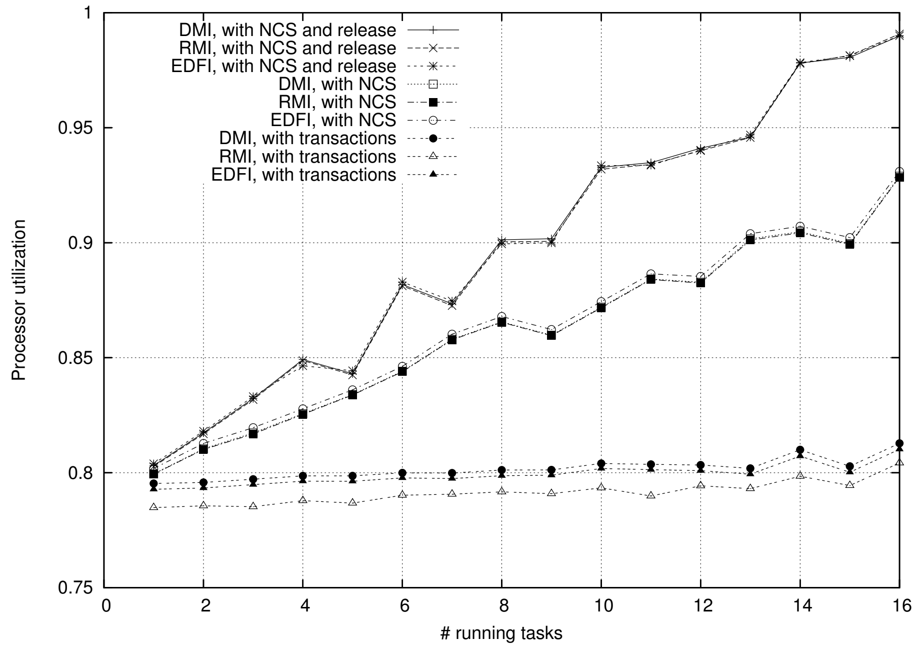


(a)

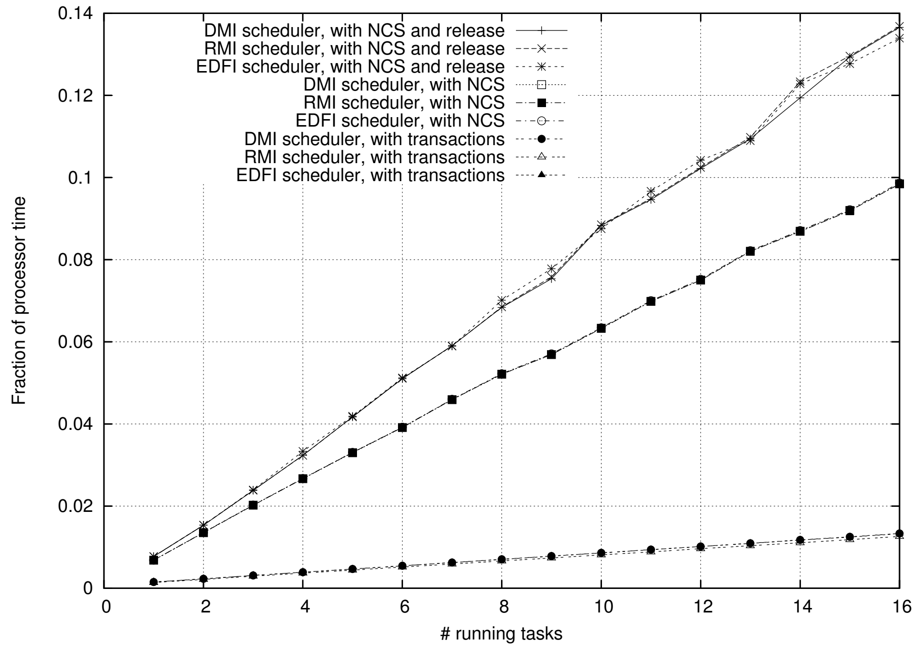


(b)

Figure 7.9: Total utilization (a) and fraction of time used by the scheduler (b), U = 0.65



(a)



(b)

Figure 7.10: Total utilization (a) and fraction of time used by the scheduler (b), U = 0.80

resources and 19 different data types. Three cases are examined, one where the schedulers are equipped with a transaction resource policy and two where the schedulers use the NCS resource policy. When using the NCS resource policy, tasks claim all their resources and data types when started. In the two cases with the NCS resource policy, the OS releases all resources at once when the task is finished, or the tasks release their resources one by one. The deadline and the priority of the tasks range from 4079 till 4094 for task 1 to task 16, respectively. As in the previous test the used data types overlap partly with the lower and higher priority tasks.

It is expected that during this test the fraction of time used by the scheduler is equal to the test in subsection 7.2.3. The scheduler with the transactions resource policy does not require additional computations, since the inherited deadlines are already calculated offline. The NCS resource policy needs additional time to find the inherited deadline. When the resources are released one by one, even more time is needed by the scheduler. It is expected that the required time by the resource policies does not depend on the increasing utilization of the task set.

The result for this test is shown in Figure 7.11. The *a* part of this Figure shows the total utilization, while the *b* part shows the fraction of time used by the scheduler and the resource policies.

From this test it can be seen that the fraction of time used by the scheduler is indeed equal to the values measured in subsection 7.3.1. This means the time needed by the scheduler is independent of the utilization. Figure 7.11 shows a high and low value in the measurement for a task set containing 12 tasks, scheduled with the DMI scheduler and the NCS resource policy. It is probable that this measurement is faulty, because the measurements with EDF and RM scheduling, in the same situation, show the correct behavior.

From this measurement it can be concluded that the time needed by a scheduler with one of the resource policies, is independent of the utilization of the task set. The transaction resource policy uses a constant amount of time, independent of the number of resources in the system. The time required by the NCS resource policy is linear correlated with the number of used resources in the task set.

7.3.3 Advantage for NCS

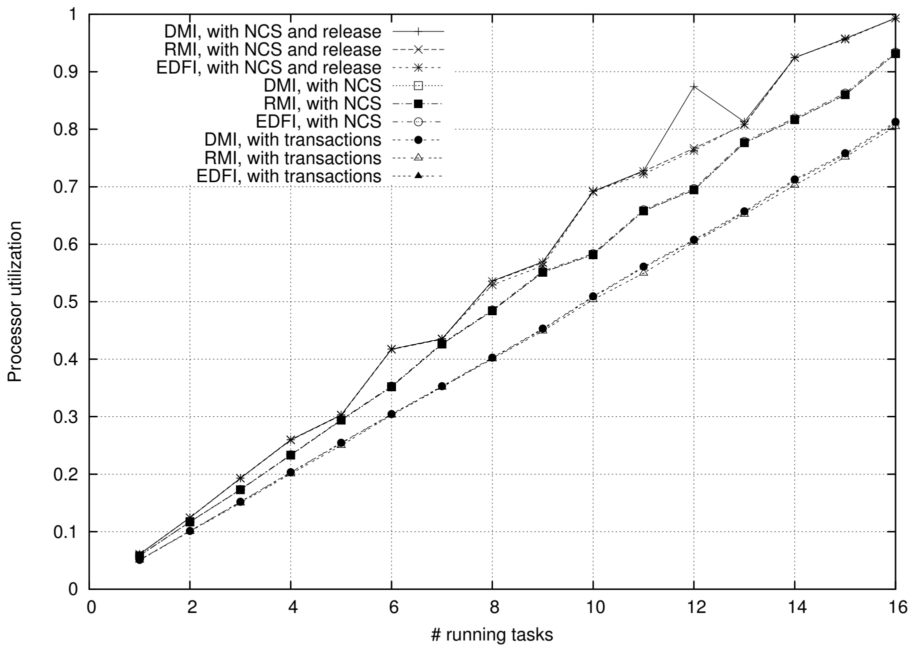
This test is performed to pinpoint the worst-case situation when using transactions. Blocking occurs when a low priority task τ_2 inherits a high priority, from task τ_1 and blocks the owner. The worst-case situation, using the EDFI scheduling algorithm, occurs when the addition of the blocking time $B(D_1)$ and run time C_1 are equal to the relative deadline D_1. In this situation τ_1 is possibly blocked by τ_2. This causes a big fluctuation in the lateness (L_1) of τ_1. The task set used in this test to simulate this behavior, is shown in Table 7.5. Task set Γ_6 is feasible with the EDFI algorithm in combination with both resource policies, although the transaction resource policy experiences a lot of blocking. The EDFI scheduling algorithm is used, because it allows more blocking compared to the RMI and DMI algorithms. It should be possible to create a similar situation for the DMI and RMI scheduling algorithms.

Table 7.5: Worst-case transaction task set Γ_6

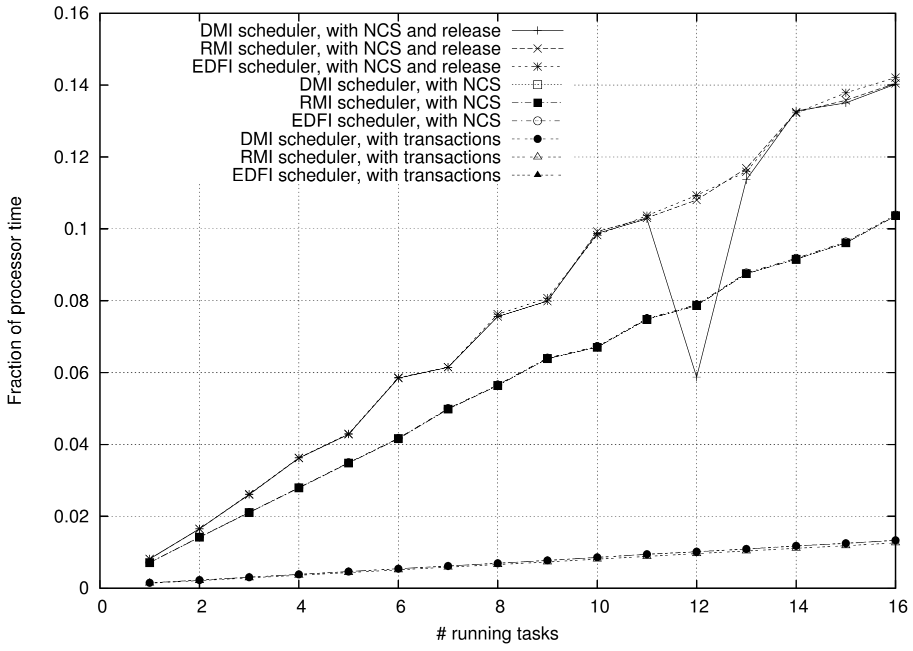
Γ_6	D	T	C	R
τ_1	160	160	40	1{*A}
τ_2	310	310	120	1{*A}

It is expected that the EDFI scheduler with the transaction resource policy shows big fluctuation in the lateness of the tasks and the number of preempted tasks will be 0. The tasks will not preempt each other, because they have the same inherited deadline. This inherited priority causes the tasks to be executed in the order they are released, causing the fluctuation in the lateness. The EDFI scheduler with the NCS resource policy will show the behavior of the underlying EDFI algorithm, since the high priority of τ_1 is only inherited for a fraction of the computation time of τ_2.

The results can be found in Table 7.6. The different resource policies can be found in the two right most columns. The rows give detail information about the measured values. The rows *#jobs* τ_1 and *#jobs* τ_2 give the number of times the two tasks are executed.



(a)



(b)

Figure 7.11: Total utilization (a) and fraction of time used by the scheduler (b), U rises to 0.80

Table 7.6: Results with task set Γ_6

	Transactions	NCS
U	0.6515	0.6889
U_{sched}	0.0439	0.0699
#preempted tasks	0	37.5
#jobs τ_1	1378	828
C_1 (mean)	38.2	38.8
L_1 (mean)	-95.29608	-113.11594
σ_{L_1}	32.47383	3.53432
#jobs τ_2	710	421
C_2 (mean)	109.6	111.1
L_2(mean)	-183.60986	-138.13777
σ_{L_2}	13.75505	9.15374

The difference between the number of executed jobs is caused by the additional timestamps needed to measure the claiming and releasing of NCSs. The results show a large standard deviation for the lateness of τ_1 when transactions are used, this is caused by τ_2 using its inherited priority. The results also show an increased σ_{L_2} and a smaller L_2 when comparing the transaction resource policy to the NCS resource policy. Due to the inherited deadline τ_2 can finish earlier with the transaction resource policy.

From this test, the conclusion can be drawn that the inherited deadline of the transaction resource policy can completely override the used priority system. The performed test shows the worst-case situation. Normal applications will not show this extreme behavior, but the underlying scheduling algorithm will not perform as well as with NCSs. By claiming the resources for a small period, the blocking gets smaller. With the smaller blocking, a scheduler in combination with the NCS resource policy can schedule task sets with a higher utilization. Furthermore the NCS resource policy shows a smaller maximum lateness compared to the transaction resource policy, when EDFI scheduling is used.

7.3.4 Admission and removal

The purpose of the test is to compare the time needed for task admission and removal with the different resource policies. Two tests are performed. The *first test* adds and removes the 16 tasks one by one. In the *second test* the tasks are added and removed in groups, starting with a group of one task and finishing with a group of 16 tasks. The task set used, is composed in such a way that the added task always has the highest priority. Since the tasks all write the same resource, this means that for every added task the inherited deadlines or the write floor has to be corrected. The periods and deadlines of the tasks range from 2001 to 2016. No maximum computation times were attached to the tasks, since only the addition and removal times of the tasks are examined.

It is expected that task admission and removal is performed faster with the NCS resource policy than with the transaction resource policy. The NCS resource policy only has to update the read and write floors of the used resources. Note that this is only needed when the deadline of the task is smaller than the current read or write floor, which is always the case in this test. When the transaction resource policy is used, the inherited deadlines have to be recalculated after each task admission. Therefore the system will compare the deadlines of all the tasks, which takes some time.

The results of this test can be found in Figure 7.12 and Figure 7.13. Three different measurements are performed for each case: tasks with NCSs, tasks with transactions and tasks with transactions, but no shared resources. Figure 7.12 shows the amount of used clock ticks when the tasks are added or removed one by one. The *a* part of the Figure shows the admission time and the *b* part shows the removal time, of the tasks. In Figure 7.13 the clock ticks needed for the admission and removal for groups of tasks is shown. As with the previous Figure, the *a* part shows the time needed for admission and the *b* part shows the removal time, for the groups of tasks. Note that in this Figure the unity of the x-axis is the size of the added or removed group.

The results in Figure 7.12a show that the *addition of a single task* with the NCS resource policy takes a constant amount of time, as expected. When tasks without resources are added, while using the transaction resource policy, the time used is increasing linear compared to number of added tasks. The reason for this linear increase in used time is that, although there are no resources, the list with tasks is processed twice. In the first iteration all task dependencies are processed and in the second processing round, the tasks get their inherited deadline set. Each time a task is added the list grows, causing the time needed to check the dependencies and write the inherited deadlines to increase. The time needed by the transaction resource policy to add tasks with resources to the task set is increasing polynomial with the number of added tasks. For each task the deadlines of the tasks in the dependency list are examined. In this test all tasks share the same resource, so each task examines the whole task set to determine its inherited deadline.

The results shown in Figure 7.13 matches the expectations. A similar explanation as in the previous paragraph can be used. An interesting thing seen in the a part of the Figure is the pattern for task admission. When the number of added tasks has three as a divider, the number of used clock cycles show a kind of a dip. This is caused by the limitation on the scheduler, which prevents the scheduler from inserting more than 3 tasks at once in the task set. This pattern is best seen in the measurements performed with the transaction resource policy, but also occurring in the measurement with the NCS resource policy. The b part of this Figure shows that the clock ticks needed for the transaction resource policy increases polynomially, compared to the linear increasing number of ticks used by the NCS resource policy to remove groups of tasks.

From this test the conclusion can be drawn that the addition and removal of a task can be done in constant time, when a scheduler with the NCS resource policy is used. The time needed to insert or remove a task with a scheduler using the transaction resource policy, is polynomially ($O(n^2)$) correlated with the number of tasks in the active task set. The protection of the scheduler, preventing it to insert more than 3 tasks at the same time, causes the need for additional time when inserting groups of tasks.

7.3.5 Conclusion

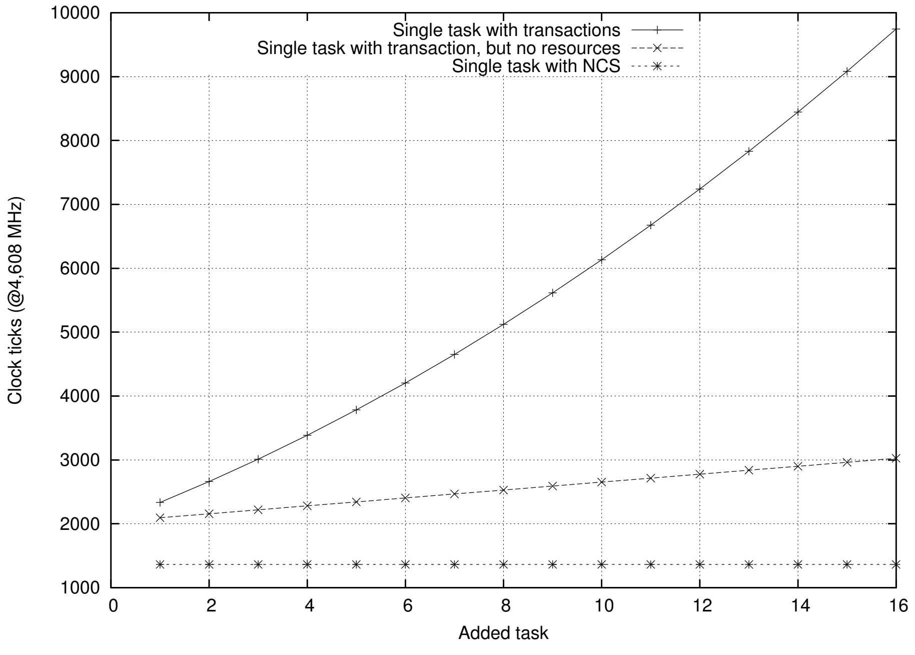
In the previous section two resource policies are compared. The tests in subsection 7.3.1 and 7.3.2 show that the time used by the resource policies is not related to the utilization of the task set. Further more the tests show that the transaction resource policy uses a constant amount of time, independent of the number of resources in the task set. The time used by the NCS resource policy is linearly correlated to the number of used resource in the task set. When considering the behavior of the two resource policies, the characteristic features of the used scheduling algorithm are better preserved when NCSs are used. This because the tasks are not using their inherited priority full time. Furthermore the NCS resource policy introduces less blocking, allowing a larger set of task sets to be scheduled. Note that transaction resource policy use less processing power, allowing task sets with a high utilization to be schedulable, while the NCS resource policy could exceed deadlines due to the additional amount of time needed.

When the admission and removal times of tasks from the task set are examined, opposing results are found. The NCS resource policy uses a constant amount of time to remove or add tasks. The transaction resource policy determines the inherited deadlines of the tasks by examining the tasks with whom resources are shared for each task. Therefore the time used for admission and removal of tasks is polynomially related to the number of tasks in the task set.

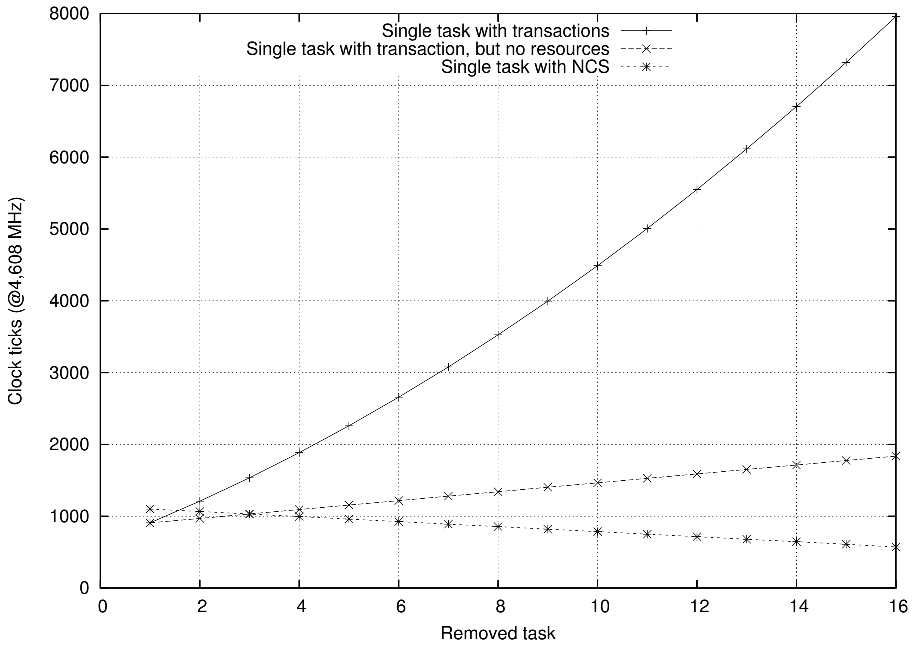
7.4 EDFIS and TSS compared

In this section the two introduced energy conserving policies are compared. This is done by performing simulations in the best and worst-case scenarios for both policies. This should explain the advantages and disadvantages of both policies. EDFIS is related to EDFI scheduling, thus EDFI scheduling is used in all tests.

Real-time measurements of the consumed power are difficult and require a complex test setup. Therefore the tests in this section measure the idle time and the time the processor is busy performing calculations. Beside

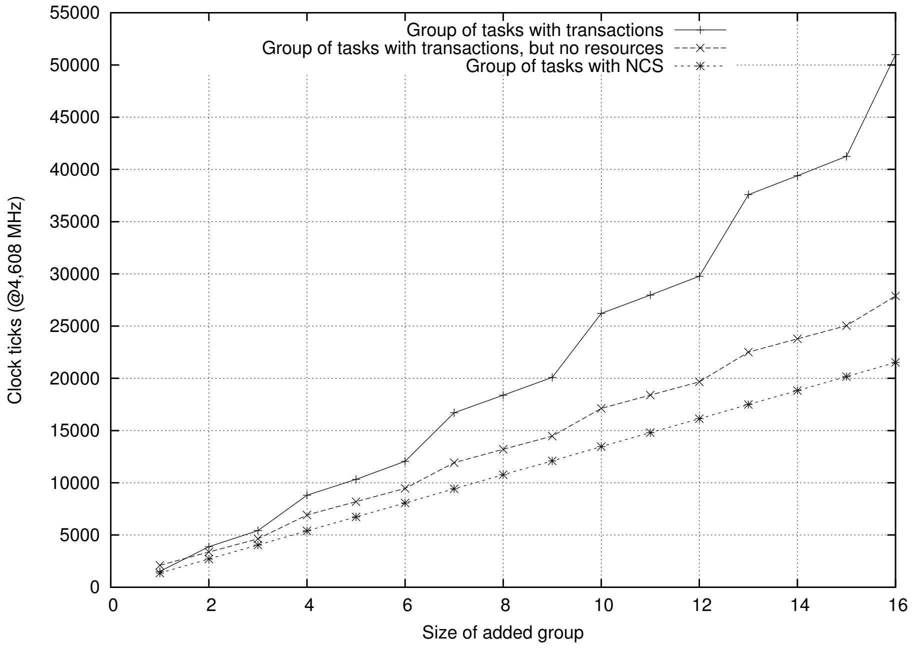


(a)

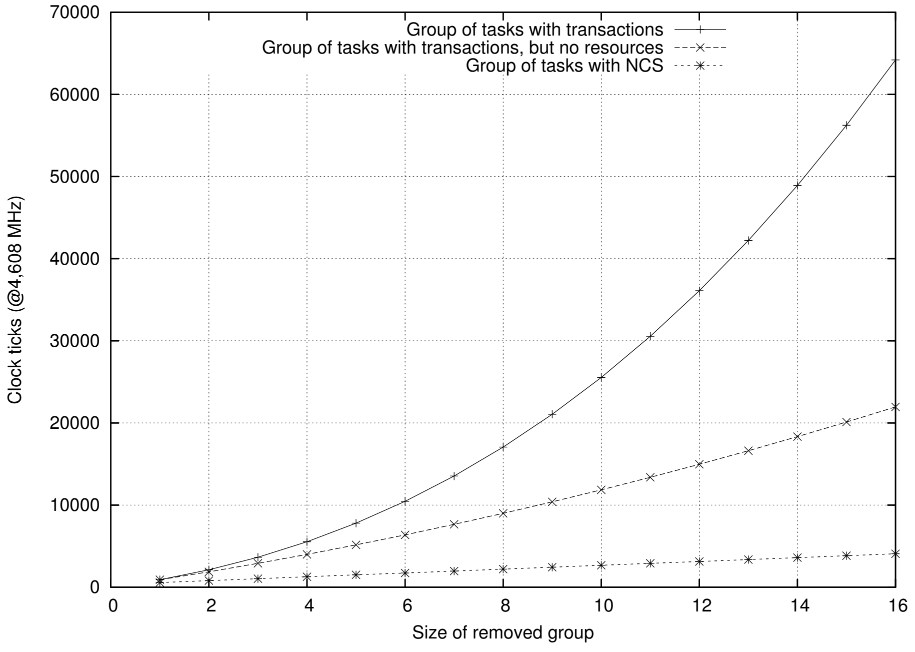


(b)

Figure 7.12: Admission (a) and removal (b) of single tasks, with different resource policies



(a)



(b)

Figure 7.13: Admission (a) and removal (b) for groups of tasks, with different resource policies

that, the determined frequency of the DCO is stored when the EDFIS policy is used. Offline test are performed to determine the power consumption of the μnode v2.0 at the different frequencies and during the different low power modes. During these measurements the peripherals of the sensor node are disabled. In this way only the power consumption of the MSP430 and the electronics at the μnode v2.0 are considered.

The measured time for TSS scheduling has to be corrected, because the 6 μs wake up time is not measured, but can be found when the power consumption is examined with an oscilloscope. Therefore the measurements of the TSS policy are corrected, by extending the computation time and decreasing the idle time, with the number of times the microcontroller woke up multiplied with 12 μs. Note that shutting down the microcontroller also takes time.

Testing with the TSS policy means that the DCO is turned off, when idle time is detected. Therefore the speed of the USART connection is lowered to 9600 baud. This introduces timing problems while sending test results to the PC, which can be solved by lowering the speed at which the information is fed to the USART connection.

The tests in this section first examine the best-case scenarios for the two energy conserving algorithms. First the best-case scenario for the EDFIS policy is examined, followed by the best-case scenario for the TSS policy in subsection 7.4.2. Next subsection 7.4.3 examines the worst-case scenario for the EDFIS policy. The last test in subsection 7.4.4 examines the worst-case scenario for TSS scheduling. This section finishes with a conclusion of the found results.

7.4.1 Advantage for EDFIS

This test is performed with the purpose of examining how both energy conserving policies behave in the best-case situation for the EDFIS policy. In the best-case situation for the EDFIS policy, the deadlines of the tasks are equal to their periods and no resources are shared, enabling the policy to scale the utilization close to 0.90. Therefore a task set containing one task, with a period and deadline of 500 is selected, without resources. Note that the task does not run at the frequency of the scheduler, causing additional scheduler calls for the TSS policy. Five different utilization factors are examined: 0.10, 0.20, 0.40, 0.60 and 0.80. The computation time of the task is 50 with a utilization factor of 0.10 and 400 with the utilization factor of 0.80.

It is expected that the EDFIS policy will have the advantage that the utilization of the task set can be scaled to something close to 0.90. Since the frequency of the microcontroller is scaled the additional scheduler calls at 16 Hz are also performed at a lower frequency, consuming less power. The TSS scheduler has to wake up, to perform the additional scheduler calls at 4.608 MHz, causing this policy to use more power.

The results for this test are shown in Figure 7.14. The *a* part shows the total utilization and the fraction of time used by the scheduler, for the different policies. Three different measurements are performed: *with the EDFIS policy*, *without any energy conserving policy* and *with the TSS policy*. The *a* part of the Figure shows a lower total utilization of the EDFIS policy, compared to the total utilization of TSS scheduling. The total utilization of TSS scheduling gets close to 1, because the fast timer used for the timestamps is turned off when idle time is detected. The experienced idle time for the TSS policy is similar to the case in which no scaling is applied. The three lines in the lower part of the Figure represent the fraction of time used for scheduling, for the different measurements. Again the fraction of time used for scheduling while using the TSS policy is increased due to the absence of recorded idle time. The *b* part of the Figure shows the estimated energy consumption at the different utilization factors. Table 7.7 provides additional results about the frequency determined by the EDFIS policy and the time the policy needed to determine the frequency. Note that the EDFIS policy is performed in slack time, while the task set is running at full speed. This time is used only once.

Observing the results it can be seen that, while using the EDFIS policy, the fraction of time used by the scheduler gets linearly larger when the frequency is reduced. The line in Figure 7.14*a* is not completely linear decreasing because EDFIS does not apply the same scaling factor to the task set for the different utilization factors. Furthermore the scaling of the EDFIS policy increases the utilization to 0.80, while 0.90 was expected. This is caused by the additional 20% of time added to the measured computation time of the tasks in the task set. The low power consumption of the TSS policy on all the different utilization factors is also unexpected. The savings of the policy in idle time, make up for the additional scheduler calls which are performed at full speed.

From the results in Figure 7.14*b* a power saving of 30% for EDFIS and even 39% for TSS can be derived[1], when the utilization of the task set is 0.10. The percentage of saved energy drops linear to 2% for both policies, when the utilization rises to 0.80.

The test results lead to the conclusion that the TSS policy is able to safe the largest amount of power with the single task in the task set. The EDFIS policy also saves a fair amount of power compared to the normal situation.

Table 7.7: Measurements for the EDFIS analysis, during the best-case EDFIS analysis

U	0.10	0.20	0.40	0.60	0.80
Frequency (Hz)	569,344	1,130,496	2,322,432	3,371,008	4,550,656
Duration EDFIS analysis (s)	0.0245	0.0158	0.0100	0.0100	0.0161

7.4.2 Advantage for TSS

Corresponding to the previous test the goal of this test is to select the scenario in which TSS scheduling performs best. The reasoning is that TSS scheduling profits from large idle times between tasks. Furthermore it if the 16 Hz scheduler calls occur simultaneously with the task releases, no wake up time is spoiled. Therefore a task set with one task is selected. The task has a period and deadline of 2048 and is attached to the same timer as the scheduler. The task set is examined for multiple utilization factors: 0.10, 0.20, 0.40, 0.60 and 0.80.

It is expected that the TSS policy performs best in this test, since the idle time is maximized. The EDFIS policy will also show a good performance, because the task set is scalable, due to the deadline that is equal to the period and the absence of blocking.

The results of this test are shown in Figure 7.15. As explained in the previous subsection, part *a* of the Figure shows the total utilization and the fraction of time used by the scheduler. The *b* part of the Figure shows the power consumption. Table 7.8 provides additional measurements for the EDFIS policy.

The results show the expected behavior. The utilization of the EDFIS and TSS policy show similar behavior as in the previous test in subsection 7.4.1. Figure 7.15*b* shows that the TSS policy indeed saves the largest amount of power. With a utilization factor of 0.10, the TSS policy uses 42% and the EDFIS policy uses 34% less power, compared to the normal case[2]. This power saving decreases almost linear for the TSS policy, to a power saving of 2% at a utilization factor of 0.80. The EDFIS policy shows a nonlinear line in the percentage of saved power. When the utilization of the task set is 0.40, the percentage of power saved by the EDFIS policy equals the savings of the TSS policy. When the utilization is above 0.40, the percentage of power saving gained with the EDFIS policy decreases even sub-linear.

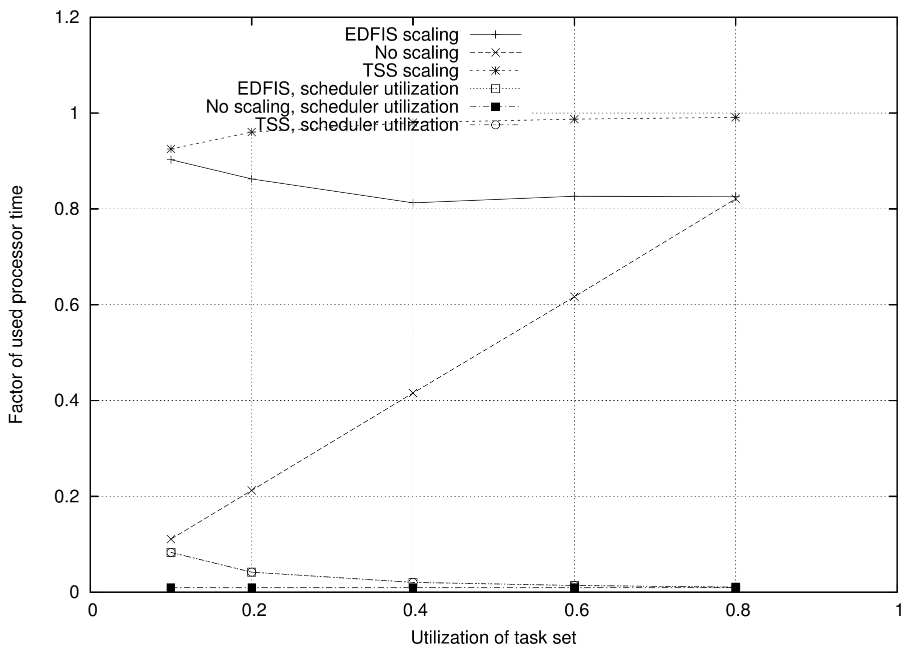
From this test the conclusion can be drawn that TSS scheduling saves the largest percentage of power, in this case. Interesting is the sub-linear relation of the saved power with the utilization, for the EDFIS policy. This because the policy saves more energy compared to TSS when the utilization increases, especially in the best-case scenario for TSS.

Table 7.8: Measurements for the EDFIS analysis, during the best-case TSS analysis

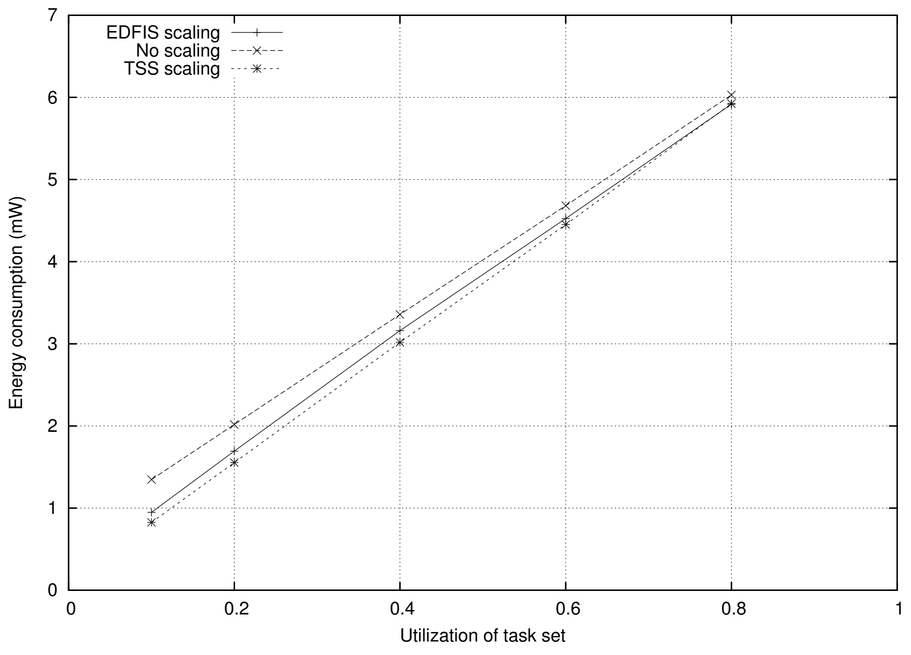
when the utilization increases U	0.10	0.20	0.40	0.60	0.80
Frequency (Hz)	540,672	1,093,632	2,191,360	3,272,704	4,378,624
Duration EDFIS analysis (s)	0.0085	0.0085	0.0086	0.0088	0.0088

[1]Applying the provided information about the batteries [26], with the TSS policy the node can be operational for 4.4 months, with the EDFIS policy for 3.8 months and in the normal case only 2.7 months.

[2]When applying the information about the batteries [26], the TSS policy extends the life time from 2.8 months to 4.8 months, while the EDFIS policy keeps the node functional for 4.2 months.

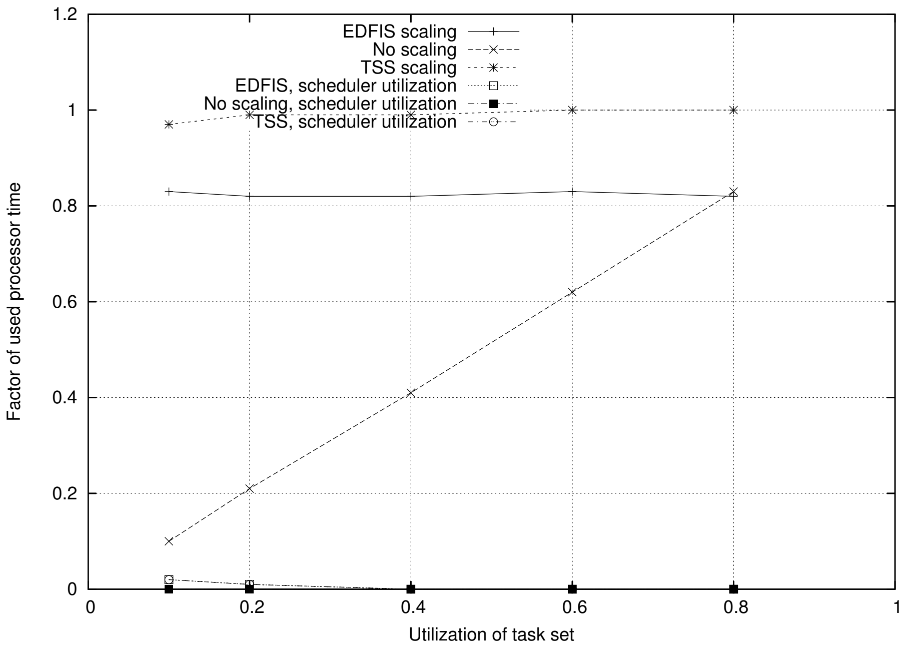


(a)

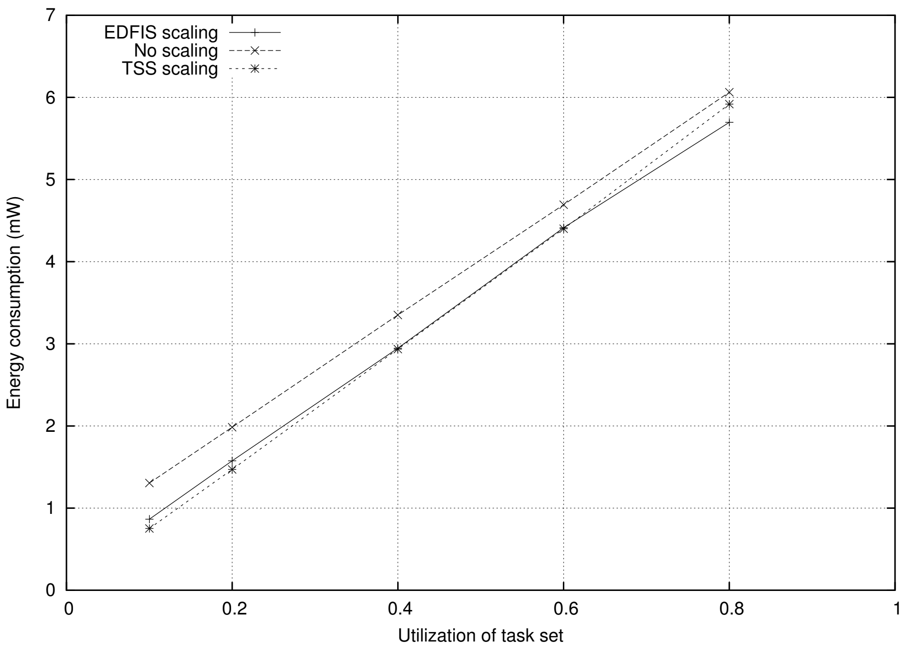


(b)

Figure 7.14: Utilization (a) and power consumption (b) during a best-case scenario for EDFIS



(a)



(b)

Figure 7.15: Utilization (a) and power consumption (b) during a best-case scenario for TSS

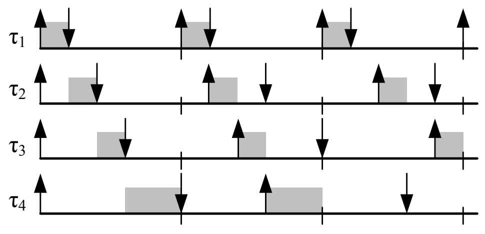


Figure 7.16: Worst-case scenario for the EDFIS policy

7.4.3 Disadvantage for EDFIS

To examine the worst-case scenario for the EDFIS policy a task set should be selected with small deadlines. In fact the scenario sketched in Figure 3.4 can be extended with small deadlines to create the scenario in Figure 7.16. Note that the tasks cannot be scaled, since that would cause the tasks to exceed their deadlines. Therefore a worst-case situation for the EDFIS policy occurs when a task set cannot be scaled. In such a case the power consumption is equal to the normal situation. The other tests show examples of how the TSS policy relates to the situation in which no scaling is performed.

7.4.4 Disadvantage for TSS

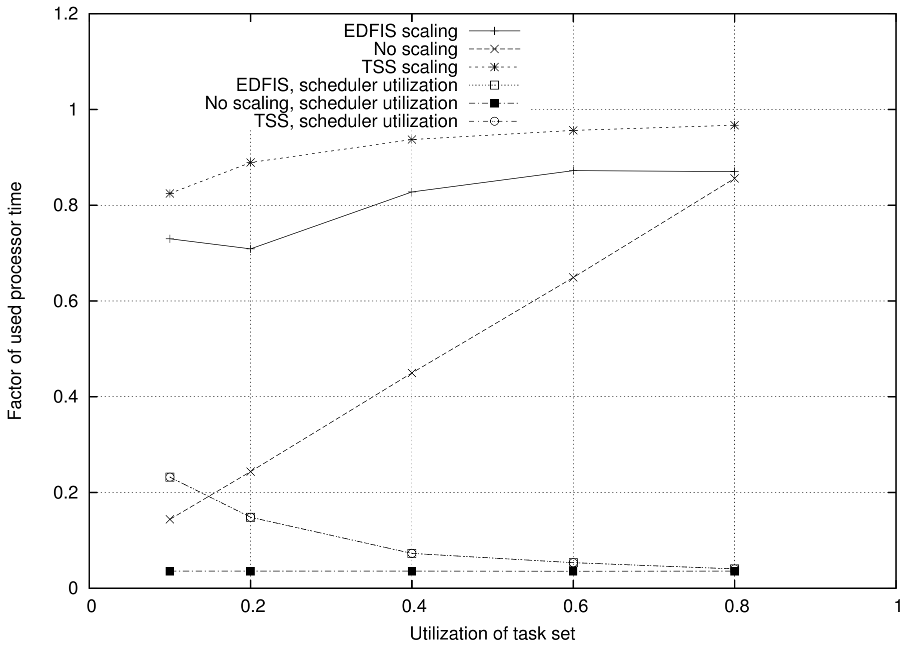
This subsection tries to select the worst-case scenario for TSS scheduling. Since TSS scheduling performs best with large idle times, a task is selected with a small period. The task has a period and deadline of 125. This means that the task is released at a frequency of 262 Hz. Furthermore the frequency is chosen such that it differs from the frequency of the scheduler. Different utilization factors are examined: 0.10, 0.20, 0.40, 0.60 and 0.80.

The expected result is that the wake up time of the TSS policy introduces a large amount of overhead. This will cause the policy to be less efficient, compared to the EDFIS policy. Since the EDFIS policy scales the utilization of the task set to 0.80, it will perform similar as in the tests in subsection 7.4.1 and 7.4.2.

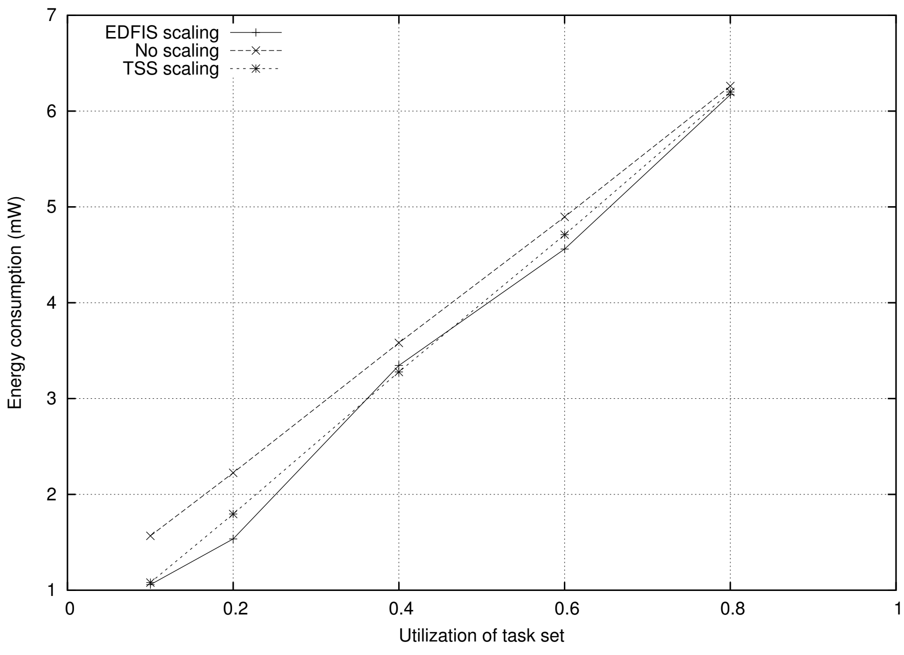
The results of this test are shown in Figure 7.17. The *a* part of the Figure shows the total utilization of the task set and the fraction of time used by the scheduler. In the *b* part of the Figure the power consumption for the different utilization factors can be found. The results for the EDFIS policy at the utilization of 0.10 and 0.20 are included, but are not valid. Due to the increase of needed scheduler time at lower frequencies, the task exceeded its deadline multiple times. Therefore the results of the EDFIS policy are not valid. The problem of *increasing the computation time* of the tasks to account for the scheduler time at lower frequencies and with fast timers, is left as future work. In Table 7.9 additional measurements for the EDFIS policy are provided.

The results match the expectations, since the lowest power saving of the TSS policy is achieved. Compared to the normal case, only 31% of power is saved, with a utilization factor of 0.10. This decreases almost linear to a power saving of 1% when the utilization factor of the task set is increased to 0.80. The results for the EDFIS policy at the utilization of 0.10 and 0.20 are not valid. Nonetheless the power savings achieved at the factors 0.60 and 0.80 are interesting. At these utilization factors, the policy saves more power, compared to the TSS policy, 7% at a utilization of 0.60 and 1.5% at a utilization of 0.80. In this test EDFIS saves even more power at a utilization of 0.60, compared to the best-case test of EDFIS in subsection 7.4.1. The reason for this power saving is the higher utilization of the task set and the small period of the task. Since the small period of the task causes a higher utilization of the scaled task set, less idle time is wasted compared to the test in subsection 7.4.1.

From this test the conclusion can be drawn that the TSS policy saves even a fair amount of power in its worst-case situation, in combination with tasks sets with a low utilization. Although the EDFIS policy is not able yet to handle the high frequent clocks, a good percentage of saved power is showed at utilization factors above the 0.60. At these utilization factors it outperforms the TSS policy.



(a)



(b)

Figure 7.17: Utilization (a) and power consumption (b) during a worst-case scenario for TSS

Table 7.9: Measurements for the EDFIS analysis, during the worst-case TSS analysis

U	0.10	0.20	0.40	0.60	0.80
Frequency (Hz)	737,280	1,179,648	2,469,888	3,354,624	4,460,544
Duration EDFIS analysis (s)	0.0293	0.0536	0.0551	0.0831	0.2111

7.4.5 Conclusion

As more devices operate in a energy scarce environment, the problem of energy conservation becomes more relevant. The tests in this section examined the behavior of the TSS and EDFIS policy, which each save energy in a different way. The TSS policy saves energy by disabling the peripherals and parts of the microcontroller, while the EDFIS policy saves energy by reducing the speed of the microcontroller.

The tests performed in this section, show that there is not a single best performing policy. The results show that in general the TSS policy saves most energy, in combination with task sets that have a low utilization, typically 0.40 or lower. Note that when the TSS policy is used, devices cannot use the fast timers. For example the USART should be connected to the external crystal. At a utilization factor above the 0.60 the EDFIS policy saves most power. These results show that the TSS policy should be applied in combination with task sets that have a low utilization and EDFIS should be applied in combination with task sets which have a high utilization. Introducing the idea that a combination of the two policies could even outperform both. The development of such an policy is left as future work.

Chapter 8

Conclusion

The context of the presented research is a Wireless Sensor Network (WSN). This network is build up by μnode v2.0 sensors using the AmbientRT Operating System (OS). In this real-time OS different solutions are compared.

The presented research focuses on three different subjects, all related to real-time scheduling. The first examined subject is the real-time scheduling algorithm. Although many research and tests are available, the behavior of the algorithms is examined for the μnode v2.0. Furthermore the performance of different resource policies is examined on the μnode v2.0. Because sensor nodes belong to the category of devices that operate in a typically energy scarce environment, two energy conservation policies are examined.

The next sections will start with a summary of the theory. Some remarks will be given about the used algorithms and the performed test are discussed. Finally a conclusion is drawn from the collected information.

8.1 Scheduling algorithms

The behavior of the Earliest Deadline First (EDF), the Deadline Monotonic (DM) and Rate Monotonic (RM) real-time scheduling algorithms is examined. The EDF scheduling algorithm assigns dynamic priorities to tasks, while DM and RM scheduling assigns static priorities.

The performed tests show similar behavior for the different scheduling algorithms of the AmbientRT framework. The time needed by the algorithms is independent of the utilization of the task set in the system. The time used by all three scheduling algorithms correlates linear with the number of tasks in the task set. A test to examine the available time for the task set shows that in the worst-case situation a fraction of 0.884 of the processor time is available for the task set. When the advantages of the different algorithms are examined, EDF scheduling is able to schedule a larger group of task sets with a high utilization, in particular when the deadlines of the tasks are equal to their periods. When a test is performed to compare the lateness of high priority tasks under DM and RM scheduling with the lateness of the same high priority tasks under EDF scheduling, only a small difference is found. In this comparison the lateness of the other tasks is smaller under EDF scheduling, compared to DM and RM scheduling.

The conclusion can be drawn that the percentage of time used by the scheduling algorithms in the AmbientRT OS is similar for the EDF, DM and RM algorithms. Therefore the characteristics determine the preferred algorithm. EDF scheduling is the best option, since it allows task sets with a utilization of at most one and minimizes the maximum lateness; the maximum lateness lower bound offers a predictable average response time of the tasks. When a short high priority task should be performed with a guaranteed small lateness, the usage of DM scheduling becomes interesting. However the lateness of the other tasks cannot be guaranteed under DM scheduling.

8.2 Resource policies

In this report two resource policies are high lighted, the transaction and Nested Critical Section (NCS) resource policy. The purpose of the resource policies is to provide *mutual exclusive access* to data and resources. Both resource policies achieve this by assigning an *inherited priority* to each task, which is possibly higher than the current priority of the task, preventing other tasks from preempting. The transaction resource policy assigns this priority for the whole length of the task. The NCS resource policy assigns the inherited priority only in the period that the task is using the resource. Therefore the NCS resource policy determines the inherited priority at run time, while the transaction resource policy assigns them offline. Note that, since the transaction resource policy assigns the higher priority for the whole duration of the task, more blocking occurs.

The expected behavior is confirmed by the tests. The transaction resource policy takes a constant amount of time, independent of the number of resources in the system, because the inherited deadlines are calculated offline. The time used by the NCS resource policy increases linear with the number of used resources in the system. Additional tests show that the transaction resource policy may hinder the behavior of the underlying scheduling algorithm, by always assigning the tasks a static inherited priority. Therefore a smaller maximum lateness can be achieved when the NCS resource policy is used in combination with EDF scheduling. The addition of a task to the task set with the NCS resource policy can be performed in constant time. When the transaction resource policy is used the time required for the addition of a task is polynomial ($O(n^2)$) in the number of tasks in the task set.

Since the transaction resource policy uses a constant amount of time, it seems the best solution in a resource scarce environment. The NCS resource policy allows the scheduling algorithm to schedule the tasks according to their priorities, but uses linearly more time on the microcontroller when the number of used resources increases. Therefore the best resource policy depends on the task set. When the task set has an average amount of used resources, the NCS resource policy is the best solution in most cases. When the task set has a high utilization or uses many resources the transaction resource policy is likely to be the best solution.

8.3 Energy conserving policies

Since resources are scarce in a WSN, two energy conserving policies have been developed. The Temporal Shutdown Scheduling (TSS) puts the microcontroller of the sensor node in a low power mode, when it detects idle time. An encountered problem when using this solution, is that devices cannot be attached to the fast internal clock. This clock gets disabled when entering the low power mode. The second energy conserving policy is Earliest Deadline First with Inheritance and Scaling (EDFIS). This policy stretches the tasks in the task set to their maximum length. The scaling possibilities of the EDFIS policy are limited by the amount of blocking and early deadlines of tasks in the task set.

When these solutions are applied to different hardware platforms, different results will show up. As stated in subsection 2.4.4 other processors might have registers, caches or transition look aside buffers that get lost when they get disabled. It is also possible that the wake up time is longer, decreasing the responsiveness of the TSS policy. At the other hand the ability to apply online frequency scaling might be missing, making it difficult to use the EDFIS policy. Another interesting addition to the hardware platform, for the EDFIS algorithm, could be the option to scale the voltage. With a small extension, the algorithm could use this option to achieve even better power savings.

To examine the energy consumption of the two policies the best and worst-case situations have been distinguished for both. The tests show that at a utilization of 0.10 the power saved by the TSS policy lies between the 31% and 42%. The difference is mainly caused by the frequency at which the scheduler is called, which is related to the period of the tasks in the task set. In the worst-case situation of the EDFIS policy, it can not scale the task set due to small deadlines or blocking. The maximum amount of power saved by the EDFIS policy is 34% at a utilization of 0.10. Another interesting result shown in the tests is that EDFIS outperforms TSS when the utilization of the tasks set is above 0.60.

Not a single solution can be provided the best power savings on the μnode v2.0. In case the utilization of the task set is low, typically below 0.40, the TSS policy can deliver power savings up to 42%. Drawback of this solution is that devices cannot use the fast timers. When the utilization is above 0.60 the EDFIS policy shows a larger amount of saved power. Restriction is that the task set should not have to much blocking or small deadlines.

Chapter 9

Future work

Searching efficient solutions for sensor nodes did not only provide results. The research also provided new points of interrest:

- **Precedence constraints and sporadic tasks:** The data-centric scheduler of AmbientRT releases tasks on signals of running tasks and on received interrupts. Both are not supported by the used real-time scheduling theory. Therefore research should be performed to integrate the precedence constraints and sporadic tasks in the theory.
- **Determination of inherited deadlines:** When tasks with shared resources are added to a task set, while the transaction resource protocol is used, a polynomial relation ($O(n^2)$) between the time needed for addition and the number of tasks can be found. It should be possible to find a solution that determines the inherited deadlines of the tasks in linear time.
- **Efficient NCS allocation:** The results in section 7.3 show a strong increase in time needed by the scheduler and the NCS resource policy when more resources need to be claimed. Further research can be performed to optimize and reduce the required time to claim and release a resource. The current implementation minimizes the memory consumption by assigning the memory for the NCS structure, when it is required. It might be faster to provide a preallocated NCS structure for each resource. An additional analysis should be performed to determine if there are multiple readers of a resource, that can be on the the stack at the same time. In this case additional NCS structures can be preallocated.
- **Determination of computation time:** When using the EDFIS algorithm, the time needed by the scheduler increases with a lower frequency or faster clocks. Research should be performed to determine the amount of time for scheduling assigned to the tasks.
- **Procrastination of tasks:** The best case situation for TSS scheduling occurs when large pieces of idle time are available. Literature [16] proposes to delay tasks in order to create larger gaps with idle time. This could increase the power savings achieved with the TSS algorithm. Note that this violates one of the main assumptions of real-time scheduling, see section 3.2. It should be verified if the definitions of real-time scheduling stay valid with such an addition.
- **Combining the TSS and EDFIS algorithm:** When the TSS and EDFIS algorithms are combined, the best of both can be used. Research should be performed to find the optimal combination in which both algorithms keep their characteristics.

Bibliography

[1] Ambient Systems B.V., Enschede, the Netherlands. *AmbientRT demo*, January 2005. v 0.4.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the eighth IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, May 1991.

[3] Neil C. Audsley. Deadline monotonic scheduling. Technical report, Department of Computer Science, University of York, September 1990.

[4] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *IEEE Proceedings of the* 13^{th} *Euromicro Conference on Real-Time Systems*, pages 59–66, Delft, The Netherlands, June 2001.

[5] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

[6] G.C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[7] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2002. ISBN: 0-7923-9994-3.

[8] Giorgio C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Systems*, 29(1):5–26, 2005.

[9] D. Culler, D. Estrin, and M. Strivastava. Overview of sensor networks. *IEEEC*, pages 41–49, August 2004.

[10] U. C. Devi. An improved schedulability test for uniprocessor periodic task systems. *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, pages 23–30, July 2003.

[11] Ajay Dudani, Frank Mueller, and Yifan Zhu. Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 213–222. ACM Press, 2002.

[12] Tjerk J. Hofmeijer. The development of system software to support a data centric real-time architecture for sensor networks. Technical report, University of Twente, The Netherlands, July 2004.

[13] Hsin hung Lin and Chih-Wen Hsueh. Power-aware real-time scheduling using pinwheel model and profiling technique. In *RTCSA'05: Proceedings of the eleventh Real-Time Computing systems and Applications*, pages 299–304. IEEE Computer Society, 2005.

[14] Shun hung Yu, Hsin hung Lin, and Chih wen Hsueh. An application using pinwheel scheduling model. In *ICPADS '04: Proceedings of the Tenth International Conference on Parallel and Distributed Systems*, pages 683–689. IEEE Computer Society, 2004.

[15] Pierre G. Jansen. A generalised scheduling theory based on real-time transactions. Technical report, University of Twente, The Netherlands, January 2003.

[16] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 275–280. ACM Press, 2004.

[17] C. U. Martel K. Jeffay, D. F. Stanat. On non-preemptive scheduling of periodic and sporadic tasks. *12^{th} Real-Time Systems Symposium*, pages 129–139, December 1991.

[18] Andrew E. Kalman. *Salvo User Manual*. San Francisco, USA, September 2003. version 3.2.2.

[19] Jian-Liang Kuo and Tien-Fu Chen. Dynamic voltage leveling scheduling for real-time embedded systems on low-power variable speed processors. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 147–155. ACM Press, 2002.

[20] Yann-Hang Lee, Yoonmee Doh, and C. M. Krishna. Edf scheduling using two-mode voltage-clock-scaling for hard real-time systems. In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 221–228. ACM Press, 2001.

[21] Joseph Y. T. Leung and Jeniffer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In *Performance Evaluation*, volume 2, pages 250–273, December 1982.

[22] Philip Levis. *TinyOS 2.0 Overview*, February 2006. http://www.tinyos.net/tinyos-2.x/doc/html/overview.html.

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

[24] J.M. Maurer. Building on the dmi and edfi foundations. Technical report, University of Twente, The Netherlands, August 2005.

[25] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102. ACM Press, 2001.

[26] Power-One. Alkaline batteries- mignong/aa - mx1500 size: Aa (lr6), 2006. www.powerone-batteries.com/en/assortments/alkaline/product-range/alkaline-product-range.php?type=4106.

[27] Amit Sinha and Anantha P. Chandrakasan. Energy efficient real-time scheduling. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 458–463. IEEE Press, 2001.

[28] L.E. Rosier S.K. Baruah and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 2:301–324, 1990.

[29] STMicroelectronics. *4 Mbit, Low Voltage, Serial Flash Memory With 50MHz SPI Bus Interface*, December 2005. Datasheet M25P40.

[30] Texas Instruments, Incorporated. *MSP430x15x, MSP430x16x, MSP430x161x:Mixed Signal Microcontroller*, March 2005. Rev D.

[31] Texas Instruments, Incorporated. *MSP430x1xx Family User's Guide*, February 2005. Rev. E [SLAU049.PDF].

Acronyms

ADC	Analog to Digital Converter
ALU	Arithmetic Logic Unit
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DCO	Digital Controlled Oscillator
DCOS	Data-Centric Operating System
DM	Deadline Monotonic
DMI	Deadline Monotonic with Inheritance
DSF	Data Specification File
EDF	Earliest Deadline First with
EDFI	Earliest Deadline First with Inheritance
EDFIS	Earliest Deadline First with Inheritance and Scaling
EEPROM	Electric Erasable Programmable Read Only Memory
FIFO	First In First Out
HAL	Hardware Abstraction Layer
HIL	Hardware Independent Layer
HPL	Hardware Presentation Layer
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LCM	Least Common Multiple
LED	Light Emitting Diode
LP	Linear Programming
NCS	Nested Critical Section
OS	Operating System
PC	Personal Computer
RAM	Random Access Memory
RM	Rate Monotonic
RMI	Rate Monotonic with Inheritance
RTOS	Real-Time Operating System
SPI	Synchronous Peripheral Interface
TSS	Temporal Shutdown Scheduling
USART	Universal Synchronous/Asynchronous Receive/Transmit
WSN	Wireless Sensor Network

Appendix A

Data Specification File

Data and tasks are defined separate from the OS in AmbientRT. A Data Specification File (DSF) is used to provide information about the tasks written in the C code and the used data. The information in the DSF file is parsed by the offline feasibility analysis and DSF parsing tool *Sparse*. This tool checks if the provided data is valid and creates a file defining the constants in C code for the task set.

This Appendix explains how tasks and data can be defined in a DSF file. Furthermore this appendix contains an extension of the DSF language to enable the usage of NCSs. In addition the extension to check the feasibility of task sets with different schedule algorithms is discussed. First the general syntax of the DSF is explained. A complete specification of the DSF language with system calls for AmbientRT, but without the extensions, is given in [1].

A data type is defined by:
data(id, size, name);
The id is the name of the variable as used within the DSF. The size defines the number of words reserved for this data type. The name, is the unique name for the data type as used in AmbientRT.

AmbientRT provides two types of timers, the fast timer working at a frequency of 32768 Hz and the slow ltimer, working at 16 Hz. There are two fast timers available for the users and up to 16 slow ltimers. A timer and ltimer are defined by:
timer(id, ticks);   # A (32768 / ticks) Hz fast timer
ltimer(id, ticks);   #A (16 / ticks) Hz slow ltimer
The *id* of the timer or the ltimer should be in the range of the available numbers, with the first timer having id 0. The *ticks* variable denotes the frequency the timer should operate on, with ticks maximal 65536 for both timers. For the fast timer the frequency is $\frac{32768}{ticks}$ and the ltimer has a frequency of $\frac{16}{ticks}$.

Tasks need to be annotated with scheduling information, so the real-time scheduler can assign priorities:
task(name, cname, deadline, period, cputime, reserved)
The *name* is the name of the task in the DSF file. The *cname* is the name of the function as defined in the C file. The *deadline*, *period* and *cputime* are as needed for EDFI scheduling. The *reserved* value is not used at the moment.

Inside the task definition three keywords are available. The *dataspec* key word is used, with a list of data types the task is subscribed to or going to publish. The *resources* keyword has a list with the resources the task is going to read or write. The descriptors in the list of dataspec and resource, can be preceded by *, this means that mutual exclusive access is desired to the resource. The list with data types and or resources is given between square brackets, where the items are separated by commas. The *subscribe* keyword contains a list with data types the task wants to subscribe for, when these data types are published the subscribed task is released. Note that timers are also treated as a data types, this means that the OS publishes the timer data types. These

```
# # # # # # #
# Example Data Specification File
# This file defines a shell, that sends the number of input
# characters to a counter. The char_count is used
# to transport the value and signal the function counter.
# The function counter writes its result to the LCD display
# and the network, by the radio it is subscribed to.
# There is also a blink_function that blinks a LED
# # # # # # #
# Data types
data(char_count,2,"ccnt");

# Timers
ltimer(0,4);

# Tasks
task(shell,shell_function,1000,1024,512,0) {
    dataspec[*char_count];
    resources[*SERIAL];
    subscribe[SERIALIN];
}

task(counter,counter_function,300,1024,50,0){
    dataspec[char_count];
    resources[*LCD, *RADIO];
    subscribe[char_count];
}

task(blink,blink_function,200,8192,50,0){
    subscribe[LTIMER0];
}
```

Figure A.1: Example DSF file for AmbientRT

data types contain no actual data.

An example DSF file is given in Figure A.1. This file defines one data type ’char_count’, one slow ltimer and three tasks. The task ’shell’ is writing and listening to the serial port, since the serial port is defined as written in resources and it is subscribed for in subscribe. The task publishes the data type ’char_count’, since it is marked as a written data type in ’dataspec’. The task ’counter’ is subscribed to the data type ’char_count’, since it is in the subscribe list and as read data type in the dataspec list. The task ’blink’ is only subscribed to the slow LTIMER0, which will cause it to blink at a frequency of 4 Hz.

A.1 NCS extension

To enable the usage of NCSs, the DSF language is extended. For every task, the used resources and duration of the sections can be stated. The *usageflow* keyword is followed by the list. The syntax for the list with sections corresponds to the syntax as discussed in section 3.4.2. The exact syntax as accepted in the DSF file is:

$$\begin{array}{lll} usage & \rightarrow & [\, \rho' \,] \\ \rho' & \rightarrow & \tilde{\rho} \mid \tilde{\rho}\ ,\ \rho' \\ \tilde{\rho} & \rightarrow & int\ \{\ exclusive\ \rho\ \} \mid int\ \{\ exclusive\ \rho\ ,\ \tilde{\rho}\ \} \\ \rho & \rightarrow & resource\_name \mid resource\_name\ ,\ \rho \\ exclusive & \rightarrow & * \mid \lambda \end{array}$$

In this syntax, *resource_name* denotes a resource or data type and *int* denotes the duration of the NCS in ticks on the 32.768 kHz clock.

An example of a task set with NCSs is given in Figure A.2. This task set contains two tasks, with the names mac_in and msgin. The task mac_in is started when the radio publishes the data type RADIOIN. During its execution it will claim radio_message and LCD sequentially. Writing access to RADIO is claimed for 150 time units and LCD is claimed 50 time units for writing. The task msgin has a critical section of 50 time units in which radio_message is read. In this critical section is an NCS of 25 time units in which counter is written.

```
# # # # # # #
# Example Data Specification File
# The task mac_in receives network traffic. This information
# is displayed on a LCD and written to the data type
# radio_message. The task msgin is subscribed to radio_message
# and writes the number of received messages to the data
# type counter.
# # # # # # #
# Data types
data(radio_message,17,"rmsg");
data(counter,2,"cntr");

# Tasks
task(mac_in,mac_in,500,1024,200,0) {
  dataspec[*radio_message];
  usageflow[150{*radio_message},50{*LCD}];
  resources[*RADIO,*LCD];
  subscribe[RADIOIN];
}

task(msgin,msgin,10000,32768,100,0) {
  dataspec[radio_message,*counter];
  usageflow[50{radio_message,25{*counter}}];
  subscribe[radio_message];
}
```

Figure A.2: Example DSF file with NCSs

A.2 Feasibility analysis

The information given in the DSF file can be used to perform a feasibility analysis. With the discussed additions to AmbientRT, as discussed in section 5, the used scheduling algorithm and resource policy should be known when performing the check. Therefore the syntax to enable a feasibility analysis is extended to:
analyze(analysis_name,schedule_algorithm,resource_policy,task_list);
The *analysis_name* is the name that is assigned to this particular analysis, since multiple analysis can be declared in one DSF. In the variable *schedule_algorithm*, the preferred schedule algorithm can be chosen, the algorithm names are 'dmi', 'edfi' and 'rmi'. Note, that since rate monotonic scheduling does not use deadlines, the deadline values of the tasks are ignored. In AmbientRT two resource policies are available, 'transactions' and 'ncs', these can be chosen in *resource_policy*. The *task_list* variable contains the tasks that should be in the task set on which the feasibility check is performed. This list contains the names of the tasks, separated by commas and between square brackets. An example DSF file is given in Figure A.3

```
# # # # # # #
# Example Data Specification File
# Three tasks are defined to be analyzed by different
# feasibility analysis. This DSF shows that multiple
# task sets can be defined in a single DSF file.
# Multiple analyzes can be performed to verify their
# feasibility.
# # # # # # #
# Timers
timer(0,500);
timer(1,700);

# Data types
data(a,2,"adat");
data(b,2,"bdat");


# Tasks
task(task1,task1_function,300,500,100,0) {
  dataspec[a];
  usageflow[5{a}];
  subscribe[TIMER0];
}

task(task2,task2_function,600,700,200,0) {
  dataspec[*b];
  usageflow[10{*b}];
  subscribe[TIMER1];
}

task(task3,task3_function,700,700,300,0) {
  dataspec[*a,b];
  usageflow[15{b,10{*a}}];
  subscribe[TIMER1];
}

# Analysis
analyze(dmi_trans,dmi,transactions,[task1,task3]);
analyze(dmi_ncs,dmi,ncs,[task1,task2,task3]);
analyze(edfi_trans,edfi,transactions,[task1,task2,task3]);
analyze(edfi_ncs,edfi,ncs,[task1,task2,task3]);
analyze(rmi_trans,rmi,transactions,[task3,task2]);
analyze(rmi_ncs,rmi,ncs,[task3]);
```

Figure A.3: Example DSF file with multiple feasibility checks