Master of Science Thesis in Computer Engineering
Department of Computer and Information Science, Linköping University
Spring 2016 | LIU-IDA/LITH-EX-A--16/048—SE

# Evaluation of EDF scheduling for Ericsson LTE system

- A comparison between EDF, FIFO and RR

**Angelica Nyberg**
**Jonas Hartman**

Supervisors: Armin Catovic and Jonas Waldeck, Ericsson AB
Examiner: Prof. Petru Ion Eles, IDA, Linköping University

LINKÖPING UNIVERSITY

Linköping University
SE-581 83 Linköping, Sweden

## Upphovsrätt



## Copyright

# ABSTRACT

Scheduling is extremely important for modern real-time systems. It enables several programs to run in parallel and succeed with their tasks. Many systems today are real-time systems, which means that good scheduling is highly needed. This thesis aims to evaluate the real-time scheduling algorithm earliest deadline first, newly introduced into the Linux kernel, and compare it to the already existing real-time scheduling algorithms first in, first out and round robin in the context of firm tasks. By creating a test program that can create pthreads and set their scheduling characteristics, the performance of earliest deadline first can be evaluated and compared to the others.

# SAMMANFATTNING

Schemaläggning är extremt viktigt för dagens realtidssystem. Det tillåter att flera program körs parallellt samtidigt som deras processer inte misslyckas med sina uppgifter. Idag är många system realtidssystem, vilket innebär att det finns ett ytterst stort behov för en bra schemaläggningsalgoritm. Målet med det här examensarbetet är att utvärdera schemaläggningsalgoritmen *earliest deadline first* som nyligen introducerats i operativsystemet Linux. Målet är även att jämföra algoritmen med två andra schemaläggningsalgoritmer (*first in, first out* och *round robin*), vilka redan är väletablerade i Linux kärnan. Det här görs med avseende på processer klassificerade som firm. Genom att skapa ett program som kan skapa pthreads med önskvärda egenskaper kan prestandan av *earliest deadline first* algoritmen utvärderas, samt jämföras med de andra algoritmerna.

# ACKNOWLEDGMENTS

This final thesis is a part of the master's programme in applied physics and electrical engineering at Linköping University. It is a master thesis of 30 credits, which has been performed in cooperation with Ericsson during the spring of 2016. We would like to thank everyone who has contributed and helped us with our project. We are grateful to the company, who made this project possible and we are also grateful to all its employees for the warm welcome and for making our time at the office very pleasant. Further, we want to thank our closest colleagues for their help and support. Finally, we would like to express sincere gratitude for the support from our families and friends.

Special thanks to:

**Armin Catovic** and **Jonas Waldeck**, *technical supervisors at Ericsson*: for guidance and help through the whole project.

**Prof. Petru Ion Eles**, *examiner*: for sharing knowledge, giving advice and providing feedback on our work.

**Evelina Hansson** and **Tobias Lind**: for reading and commenting on drafts.

**Mikael Hartman**: for mathematical support.

*Linköping, Aug 2016*
*Jonas Hartman and Angelica Nyberg*

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS AND ACRONYMS

All abbreviations and acronyms defined in this list are written in *italics* when used in this report.

| | |
|---|---|
| CBS | *Constant bandwidth server*. An algorithm that ensures that every process gets some guaranteed runtime. |
| CPU | *Central processing unit*. A unit that executes programs in a computer. |
| EDF | *Earliest deadline first*. A scheduling policy, which attempts to schedule tasks so that they meet their deadlines. |
| FIFO | *First in, first out*. Commonly used to describe that what comes first is served first. In this project, it will be used to describe a scheduling policy. |
| GEDF | *Global EDF*. A version of the *EDF* scheduling policy, which schedules processes on more than one core. |
| ID | *Identity*. A unique number used as identification of the running processes. |
| IEEE | *(The) Institute of electrical and electronics engineers*. A non-profit organization that sets standards within the fields of electrical engineering, electrics engineering and programming. |
| LTE | *Long-term evolution*. The long-term development of the 3G mobile network, sometimes called 4G. |
| NOP | *No operation.* An operation, which stalls for one clock cycle. |
| OS | *Operating system*. A program, which runs in the background on the computer and manages all other programs and processes. |
| P-GW | *Packet data network*. The link between a mobile user and the internet. |
| POSIX | *(The) Portable operating system interface*. A group of standards maintained by the *IEEE* association that deal with *OSs* to ensure portability. |
| RAM | *Random access memory.* A type of main memory accessible in terms of both reads and writes. |
| RMS | *Rate monotonic scheduling.* A way to assign static priority to a task based on the task's period. |
| RR | *Round robin*. A scheduling policy commonly used in *RT-OSs*. |
| RT | *Real-time*. Something happening in real-time is happening live. |
| RT-OS | *Real-time operating system*. See *RT* and *OS*. |
| S-GW | *Serving gateway*. The part of the mobile network that maintains data links to the *UEs*. |
| UE | *User equipment.* A piece of equipment that is the end user in a mobile network, for example a mobile phone. |
| WCAO | *Worst-case administrative overhead*. The longest time a specific task is delayed by administrative services of the *OS*. It is a part of the *WCET*. |
| WCET | *Worst-case execution time*. The longest time a specific task executes. |

# Chapter 1

# Introduction

This report starts with an introduction about the subject. First, the subject is motivated in terms of why it is interesting and important to study. Second, the background is reviewed, including a presentation of the company this thesis cooperates with and the area they want to apply this work in. Third, the purpose is presented followed by the statement of the problem. The limitations of this project is also presented in this introduction and, at last, the structure of the remaining report is described.

## 1.1 Motivation

Scheduling is extremely important for real-time systems. A real-time system is a system with special requirements regarding response time. It must process information and produce a response within a specific time. It is important that a real-time system is predictable, otherwise the user or designer cannot analyse it. One way to achieve predictability is to use priority-based algorithms to schedule tasks in the system. Many systems today are real-time systems, which means that good scheduling is highly needed.

Scheduling makes it possible to run many programs at the same time, since the processor time is shared between them. When running critical programs, one wants to guarantee that the real-time deadlines are met. This can be achieved by running the critical real-time programs on over-dimensioned computational hardware. This means that lot of the processing capacity is wasted, since most of the scheduling algorithms do not take real-time deadlines into account. If they did, the used processing capacity could be increased. There are also situations where a real-time program is constrained to specific limited computational hardware due to cost and/or power dissipation.

This project aims to evaluate a deadline based scheduling algorithm called earliest deadline first.

## 1.2 Background

Ericsson's *LTE* base stations must be capable of handling large amount of operation and maintenance as well as user equipment traffic. The base stations are real-time systems handling thousands of procedures at any given time. They could therefore benefit from a good scheduling algorithm.

Two common real-time scheduling algorithms are first in, first out (*FIFO*) and round robin (*RR*). These are already implemented in the real-time execution environment for Ericsson *LTE* application. Since Linux kernel 3.14 is the first version that supports earliest deadline first (*EDF*) scheduling and Ericsson is currently using an older kernel version, they are wondering if it is advantageous to upgrade the kernel and use *EDF* scheduling for their processes.

### 1.2.1 Ericsson

The telecom company *Telefonaktiebolaget LM Ericsson* is a company founded in Sweden in 1876 (Ericsson 2016a) by Lars Magnus Ericsson. Ericsson provides services and products related to information and communications technology, which today has started to involve many areas. For example, networks, IT, media and industries. 40 % of the world's mobile traffic goes through Ericsson networks, servicing over a billion end users and Ericsson hold around 39000 patents related to information and communications technology. (Ericsson 2016b)

Ericsson is a worldwide company with around 115 000 employees servicing costumers in 180 countries. Ericsson's global headquarters are located in Stockholm, Sweden. The company had 246.9 billion SEK in net sales in 2015 and the company is listed on NASDAQ OMX Stockholm and NASDAQ New York. (Ericsson 2016a)

### 1.2.2 The LTE standard

*LTE*, sometimes called 4G, is the Long-Term Evolution (*LTE*) of the 3G network. It is a standard that is still undergoing development in order to satisfy user's demands for latency, data rates and network coverage among other things. (Dahlman, Parkvall and Sköld 2011, 7-8)

In the *LTE* mobile network, there can be a lot of user equipment, *UEs*, connected to a single base station. Several base stations are connected to a mobility management entity, *MME*, as well as a serving gateway, *S-GW*, see Figure 1.1. The job for the *MME* is to handle the mobility of different *UEs*. It sets up different channels to the *UEs* depending on their needs, usually a normal data channel and it handles the *UEs* activity states. These states are used to keep track of if they are idle, active, handling the mobility between different base stations and locating where the user is. It also runs some verification and security applications to make sure the *UE* is allowed on the network. (Dahlman, Parkvall and Sköld 2011, 110-111)

When the *MME* establishes a data channel to a user device, the *S-GW* takes over and maintains the link. It is in turn connected to a packet data network, *P-GW*, which keeps an IP-address for a specific user and handles the internet access. The *MME* and *S-GW* are often physically located in the same place, illustrated in the dark blue box in Figure 1.1. The application used for several tests in this project is located in the *MME*. (Dahlman, Parkvall and Sköld 2011, 110-111)

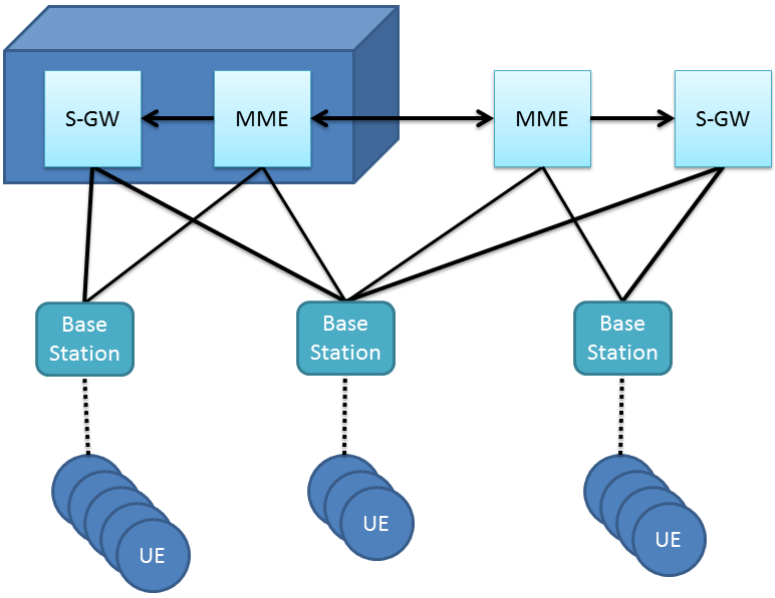


*Figure 1.1: The part of the LTE network closest to the end user*

When a new *UE* connects to the network, it searches for an available base station. The base station assists the *UE* by sending out signals in even intervals, which the *UE* can listen for. This is used for synchronization. The *UE* sends information about itself to establish a link between itself and the base station. The sent information can contain signal strength to other base stations. This information, along with the load of the current base station, can trigger the *UE* to migrate to another base station in the vicinity. The *UE* can also migrate by itself if it enters another base stations area. (Dahlman, Parkvall and Sköld 2011, 301-319)

In order to conserve battery power as well as not loading the channel, the *UE* can enter an idle state. It puts its *LTE* transmitter in a sleep mode in this state. When it receives a data packet from the network, it is important to activate the *UE* again. The *UE* is assigned a small window on a specific channel, which it periodically checks by partially waking up its receiver. If it finds the base station broadcasting in this window, the *UE* wakes up. Several *UEs* can share this window. However, the sent data is too small so the *UEs* cannot tell to whom it is directed. This means that all *UEs* assigned to that specific window wake up. At least only a subset of *UEs* are activated when one receives a message, rather than all of them. (Dahlman, Parkvall and Sköld 2011, 319-320)

## 1.3 Thesis Purpose

By creating a test program that can generate pthreads and set their scheduling characteristics, *EDF* can be compared to *FIFO* and *RR*. From these results, one will see whether *EDF* scheduling has any positive aspects compared to the other real-time scheduling algorithms. The purpose of this thesis is to evaluate the suitability of *EDF* scheduling for the Ericsson *LTE* application. This will be done in two steps. First, evaluate if the *EDF* scheduling algorithm in Linux is useful by determining if *EDF* performs better than to the other two, using the test program. Second, try out *EDF* with an application provided by Ericsson. Once this is done, the project will have fulfilled its purpose.

## 1.4 Problem Statements

Following on from the thesis purpose described above, the problems can more specifically be stated as follows (NOTE: by "current *RT* policies", *FIFO* and *RR* are implied):

Is *EDF* scheduling suitable for the Ericsson *LTE* applications?
- Is *EDF* more suitable than the current *RT* policies, regarding response time?
- Is *EDF* more suitable than the current *RT* policies, regarding utilization?
- Is *EDF* more suitable than the current *RT* policies, regarding overhead?
- Is *EDF* more suitable than the current *RT* policies, regarding met deadlines?

If not currently suitable, what could make it suitable in the future?

## 1.5 Limitations

When evaluating different scheduling algorithms, it is common to implement one's own algorithm for comparative purposes. However, this was beyond the scope of this thesis – the focus has been on evaluating scheduling algorithms already implemented and in this case, in the Linux kernel. This means that only SCHED_FIFO, SCHED_RR and SCHED_DEADLINE are evaluated.

There is a huge amount of different computational hardware on the market to choose from when deciding on the evaluation platform. Since it is unreasonable to test on all platforms, only two were chosen – both were provided by Ericsson. To use a homogeneous multi-processor system is another factor when considering evaluation platforms, however the chosen test platforms both satisfied this homogeneity. It is also important that the Linux version is more recent than or equal to 3.14, since it is the first version supporting *EDF* scheduling, i.e. SCHED_DEALDINE.

The number of test cases is also an important factor. As with the computational hardware, it is not reasonable to test all possible combinations of settings and number of running threads. Therefore, a subset of what to be believed to be the most relevant sets of test cases and parameters were chosen.

## 1.6 Report Structure

This report has started with an overview of what this thesis aims to accomplish. Then follows a chapter presenting the theory needed to understand the problem and its solution. After this, the methodology is described, followed by a chapter presenting the results. In the next chapter, these results are discussed and at last the conclusions are presented. In Appendix B, a glossary can be found, containing commonly used terms.

References not included in a sentence in the end of a paragraph are referring to the whole paragraph, while a reference included in a sentence refers to just that statement or information in that sentence. Figures do not have any references, they are all own illustrations.

# Chapter 2

# Theory

The purpose of this chapter is to gain basic knowledge about scheduling parameters, properties and algorithms relevant to this thesis work. It starts with an introduction to scheduling, followed by three subchapters describing the characteristics of *FIFO*, *RR* and *EDF*. At the end of this chapter, the Linux implementation of each algorithm is described.

## 2.1 Scheduling in General

The basic principle of scheduling is to decide in which order different tasks should run on the processor core. Scheduling should be invisible to the user, allowing many programs to run at the same time, since the processor time is time-multiplexed between them. If the computational platform is a multicore system, tasks are typically scheduled on all available cores. It is the scheduling algorithms that make scheduling decisions. Scheduling is necessary to achieve a satisfactory execution order of tasks. This means that as few deadlines as possible are missed, most preferably none. An insufficient scheduling method misses many task deadlines. Every miss can result in wasted execution time, deteriorate service or even cause a program crash. Scheduling tasks on a multiprocessor system also minimizes the length of the program schedule, which implies a faster execution.

When the computational platform is a multicore system, a global algorithm is needed to schedule the processes. A global algorithm not only needs to decide the execution order of the tasks; it also needs to decide which core the tasks should run on. Tasks are allowed to migrate between the cores. This means that tasks can switch core during runtime. If two tasks with the same priority are activated at the same time, they are assigned to the available cores arbitrarily.

In the previous paragraphs the word “task” is mentioned several times. In this report, this term is synonymous with thread and process. It is a sequential computation that is executed by the central processing unit, the *CPU*. A *CPU* is a unit, which executes programs in a computer. It retrieves machine code and executes the given instructions. In this report, there is a difference between a task and a job. A task generates a sequence of jobs, since a job is each execution of a piece of code.

### 2.1.1 Task characteristics

There are many parameters characterising a real-time task. The relevant parameters for this report are listed and described in this subchapter. For clarity, they are also illustrated in Figure 2.1. Unless otherwise stated, all information in this section is retrieved from the book Hard

Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. **(**Buttazzo 2011, 23-28)

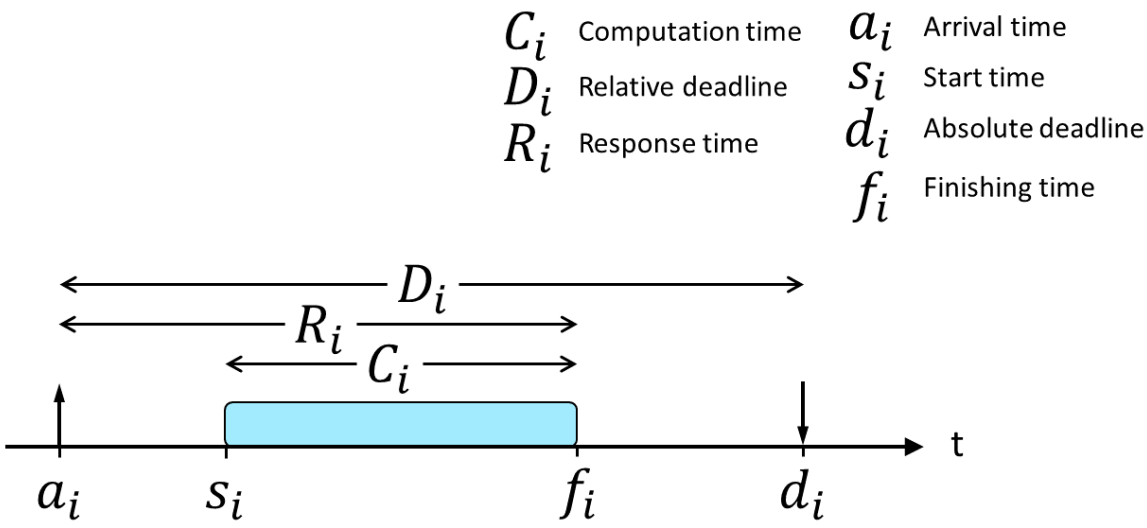


*Figure 2.1: Task parameters characterising a real-time task*

**Arrival time, $a_i$**: It is the instant of time when a task activates/wakes up and enters the ready queue. In other words, it is the time when a task becomes ready for execution.

**Computation time, $C_i$**: Assuming no interrupts, the computation time is the amount of time the processor needs to execute the task.

**Runtime**: Another word for the computation time is runtime.

**Absolute deadline, $d_i$**: The task's finishing time should occur before the instant of time when the absolute deadline occurs. If the absolute deadline is not met, the task may damage the system. This depends on the task's timing constraints; more about this in the section below.

**Relative deadline, $D_i$**: To obtain the relative deadline one must subtract the arrival time from the absolute deadline, that is $relative\ deadline = absoulte\ deadline - arrival\ time$. It is the time between when a task wakes up and when it has to be completed.

**Start time, $s_i$**: The instant of time when the task starts its execution is called start time.

**Finishing time, $f_i$**: The instant of time when the task finishes its execution is called finishing time. It is the time when the task terminates.

**Completion time**: Another word for finishing time is completion time.

**Response time, $R_i$**: To achieve the response time one must subtract the arrival time from the finishing time, that is $response\ time = finishing\ time - arrival\ time$.

**Period, $T_i$**: The period is the distance between two consecutive activations. Observe; this characteristic only exists if the task is periodic. The explanation of periodicity can be found in the section below.

### 2.1.2 Task constraints

When real-time is considered, tasks have timing constraints. A typical timing constraint on a task is called deadline. The deadline of a task is the time when it should be done executing. If the real-time task missing its deadline causes complete failure, the task is said to be hard. If the missed deadline does not cause damage to the system, the task is called firm. In this case, the output has no value and the task needs to execute again. A soft task's output on the other hand is still useful for the system after missed deadline, although the miss causes lower performance. The tasks in the *LTE* application evaluated in this thesis are considered firm. **(**Buttazzo 2011, 26)

Periodicity is another timing characteristic. Periodic tasks consist of identical recurrent jobs that are activated at constant rate **(**Buttazzo 2011, 28). The period is therefore the distance between two consecutive job activations, which Figure 2.2 shows. The jobs in the figure activate every tenth time unit, which means that the period is ten units of time. The showed task also has a computation time of four time units and a relative deadline of eight units of time.

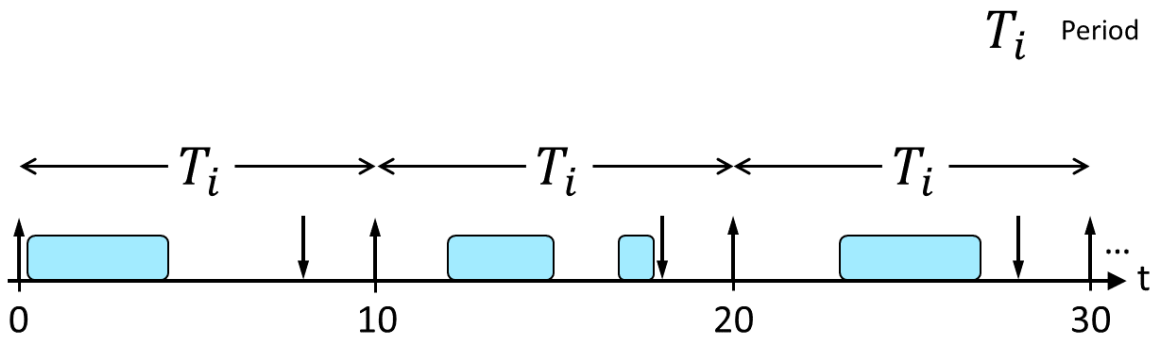


*Figure 2.2: A scheduling diagram showing three consecutive jobs of a periodic task*

The opposite of periodic tasks is aperiodic tasks. An aperiodic task consists, like a periodic task, of an infinite sequence of identical jobs. The difference is that their activations are irregular. If consecutive jobs of an aperiodic task are separated by a lower bound of time, which could be zero, the task is called sporadic. **(**Buttazzo 2011, 28)

Precedence constraints are about dependencies. It is not certain that tasks can execute in an arbitrary order. If they cannot, they have some precedence relations. These relations among tasks are defined in the design stage and are usually illustrated in a graph called precedence graph. This graph consists of nodes representing the tasks and arrows showing which order the tasks can execute. **(**Buttazzo 2011, 28-29)

When a task is running, it is allocated, or has access to, a set of resources. Resources can be private or shared. A private resource is a resource that is dedicated to a specific process while a shared resource can be accessed by more than one task. Simultaneous access of a resource is seldom allowed by the resource. These sequences in the program code are called critical sections and are managed by mutual exclusion. Mutual exclusion means that only one task at a time is allowed access to a shared resource. An example of a shared recourse is a *CPU* and another is a memory, only one task can utilize either of these at once. **(**Buttazzo 2011, 31)

Priorities are essential when deciding which order the jobs are executed in, at each time instance, by the scheduling algorithms. A typical implementation of real-time scheduling algorithms is to give the jobs with highest priority processor time. They allocate available cores according to the highest priority, which has been assigned differently depending on which algorithm is considered. Priority-based algorithms can be divided into different categories: fixed task priority scheduling, fixed job priority scheduling and dynamic priority scheduling. Fixed task priority scheduling means that each task is assigned a unique fix priority. When jobs are generated from a task, they will inherit the same priority. In fixed job priority scheduling, the priority of a job like in the fixed task priority scheduling, cannot change once assigned. The difference is that different jobs of the same task may be assigned various priorities. In dynamic priority, scheduling the job priorities may change in any time instance due to the lack of restrictions in which priorities that are assigned to a job. *EDF* scheduling is an example of fixed job priority scheduling, more about this in chapter 2.4 – *Earliest Deadline First Scheduling*. (Baruah, Bertogna and Buttazzo 2015, 24-26)

### 2.1.3 Algorithm properties

A scheduling algorithm is static if the scheduling decisions are based on fixed parameters and the scheduling decisions are made during compile time. The algorithm needs to store the scheduling decisions by generating a dispatching table off-line. The system behaviour of a static algorithm is deterministic. Dynamic algorithms base their scheduling decisions by dynamic parameters, parameters that may change during runtime. This means that these algorithms are used online. They take a new scheduling decision every time a task become active or a running task terminates. Dynamic schedulers are therefore flexible. They are also non-deterministic. (Kopetz 2011, 240-241)

The next pair of opposite properties is preemptive and non-preemptive. When a running task can be interrupted at any time to give a more urgent task processor time the algorithm is said to be preemptive. In non-preemptive scheduling, the algorithm cannot interrupt the currently executing task. It will be executed until its completion when it releases the allocated resource by its own decision. (Kopetz 2011, 240-241)

Optimality among real-time scheduling algorithms is achieved when some given cost function is minimized. This cost function may not always be defined, if so, optimality is achieved if a feasible schedule exists and the scheduler is able to find it. An algorithm is called heuristic when it compares different solutions and tries to improve itself. This type of algorithm tends toward the optimal schedule. However, it does not guarantee finding an optimal schedule. Heuristic algorithms can often be considered satisfactory enough, even if they may give suboptimal solutions. **(**Buttazzo 2011, 36)

If information about the period of a thread is known, there is a way to assign static priority to threads that is called rate monotonic scheduling (*RMS*). This way of assigning priority is based on the idea of letting the thread that runs most often, also run first. A higher static priority is assigned to a thread with a shorter period. When running on a single core system with preemption enabled, this method of assigning priority can ensure that all tasks get to run in time before their next period, assuming that the system does not have a utilization over 100 %. (Liu and Layland 1973)

### 2.1.4 Metrics for performance evaluation

Various performance metrics can be used to compare the effectiveness of different scheduling algorithms. This part will describe the performance metrics, which can be used in the investigation for comparison purposes between the scheduling algorithms described in this report.

**Utilization bounds**: The utilization factor $u_i$ of a periodic task $\tau_i$ is the ratio of its computation time and its period; that is $C_i/T_i$ **(**Buttazzo 2011, 82-84). This implies that the total utilization over the entire task set $\tau$ (the processor utilization) is defined as in Equation 2.1, which is the sum of the individual task utilizations. Buttazzo (2011) also proves that any algorithm cannot schedule a task set with a utilization factor greater than 1.0. In one way, the utilization bound is the amount of utilized processor time in percentage. It is good when it is large, but not greater than 1.0 (100 %).

$$U = \sum_{\tau_i \in \tau} u_i \tag{2.1}$$

**Worst-case execution time**: The worst-case execution time, *WCET*, of a task is a metric to determine the longest time a task requires on a specific platform (Baruah, Bertogna and Buttazzo 2015, 13-14). This is equivalent with the maximum duration between job start and job finish, assuming no interrupts. This means that the *WCET* is the guaranteed upper bound of the computation time. It is important that the *WCET* is valid for all possible input data and executions scenarios.

**Worst-case administrative overhead**: There exist delays that are not under direct control of the application task. These delays are caused by the administrative services of the operating system and are affecting the running task. All these delays are included in the worst-case administrative overhead, *WCAO*, which is a metric to determine the upper bound caused by the administrative services. If task preemption is forbidden the unproductive *WCAO* is avoided, otherwise it is a part of *WCET*. The *WCAO* is for example caused by cache misses, direct memory access, context switch rate, migrations and scheduling. (Kopetz 2011, 243-248)

**Context switch**: When a core changes from one thread to another, the core is performing a context switch. When a context switch occurs, all registers have to be stored in the memory and the registers for the new thread have to be loaded from the memory. This may take a long time.

**Other**: In addition to the explained performance metrics above, some other task characteristics are used to compare the different scheduling algorithms. These are computation time, period and response time. The number of missed deadlines and the number of scheduling failures are also interesting information to look at, where a failure is when a thread is rejected by the system.

### 2.1.5 Operating system

The operating system, *OS*, is a program that runs in the background of every other program on a computer. Its job is to assign the computer hardware to the different programs running on the machine. Usually it also handles user interface, interrupts and security. One job of the *OS* is to schedule programs to make sure that they all get to run, as transparent as possible to the user.

### 2.1.6 Processors, cores and hardware threads

A processor is the computing unit inside a computer. There are several kinds of processors in a normal desktop computer, such as a central processing unit (*CPU*) or a graphics processing unit (GPU). In this thesis, only the properties of the *CPU* are interesting. A *CPU* contains one or several *CPU* cores. These cores contain different registers, logic units, fetch and decoding units and so on. The cores are visible to the *OS* and the *OS* distributes time on different hardware threads for different software threads. A core may contain several hardware threads. If it does, it is said to be multithreaded. Multithreading is a way to get more performance from the same hardware. Usually many resources that a core has access to are unused at any given time. With multithreading several hardware threads are allowed to share the same resources at the same time. Usually a core needs a slightly larger set of registers to support this. To the *OS*, a multithread core looks like a separate core. A dual-core processor with two hardware threads on each core will be handled as four available cores to the *OS*. It is possible to set affinity to a software thread. This means that it is limited to run on one or more specific hardware threads.

Hardware threads are a feature of the *CPU's* architecture (e.g. Intel's IA-32 Hyper-Threading) while software threads are the processes (or tasks) managed by the *OS*.

### 2.1.7 Memory and caches

The memory in a computer is where programs and the data these programs work on are stored. Inside the computer there will be a large storage device such as a hard drive or a solid state drive, being able to store up to terabytes of data. This storage is called a disc storage and is really slow. It stores all programs in the computer, both the running ones and those currently not running. Below the disc storage in the memory hierarchy comes the main memory, usually a random access memory, *RAM*. When a program is started, it is temporarily loaded to the *RAM* from the disc storage. The *OS* assigns a specific amount of main memory to a program that it has to work with.

Closest to the processor cores are the registers. A single register can store up to a few bytes. These are very fast, taking only a few or even a single clock cycle to access compared with the main memory that takes upwards of hundreds or thousands of clock cycles. The main memory and the registers are what is visible to the program. There are too few registers to store a whole program, so this has to be stored in the main memory. A slow main memory is therefore a huge bottleneck for the processor performance. To handle this problem there are usually one or several layers of cache between the processor and the main memory. A cache is a small fast memory that stores a copy of a part of the main memory close to the processor. A

level 1 cache, L1 cache, is usually directly attached to the core, taking only a few clock cycles to access. In a multicore system there can often be an L2 cache attached to the processor, shared by all its cores or a set of cores (often called a “cluster”). There can also be an L3 cache (or last level cache) shared by several processors and so on. Each level of cache is bigger and slower than the previous. The cache is invisible to the program, so a memory access will take different amount of time depending on what level in the memory hierarchy the data or instruction is stored. An example of a memory hierarchy is shown in Figure 2.3.

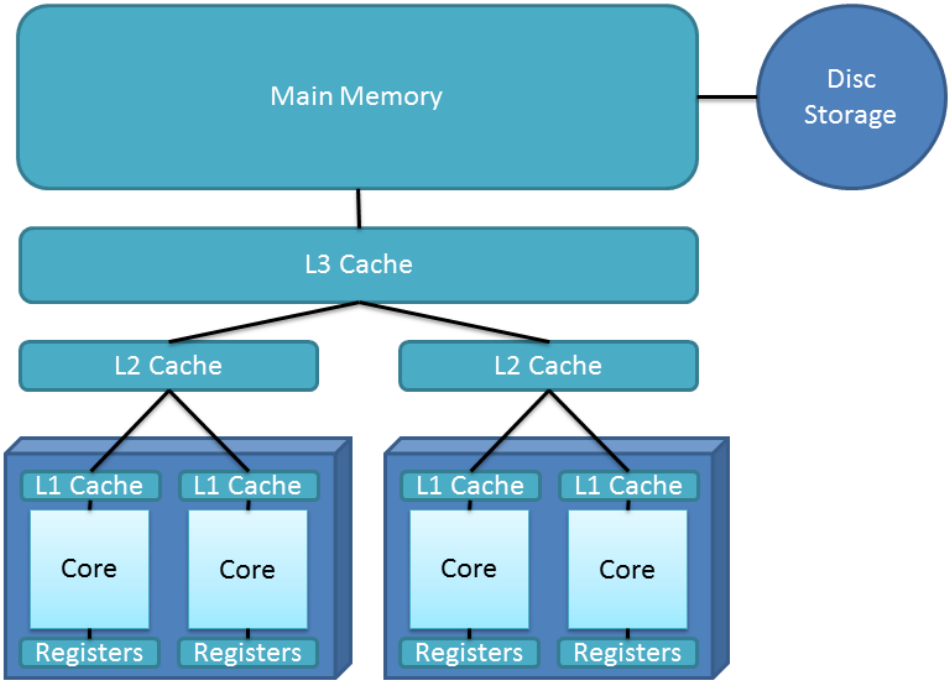


*Figure 2.3: A two processor multicore system with three levels of cache*

A cache miss occurs when the processor attempts to access a part of the memory that is not currently stored in a specific level of cache; it has to be fetched from a higher level of cache or the main memory. If the information is not in the L1 cache, there is an L1 cache miss. In this case, the L1 cache tries to fetch the data from the L2 cache, which in turn can miss and go to the next level in the cache hierarchy.

The number of cache misses is dependent on the platform; a small cache increases the number of cache misses. The number of preemptions also heavily increases it (Buttazzo 2011, 14-15). When a task with higher priority arrives and preempts the running task, the cache needs to reload. This happens when the context of the processor is switched. In the *WCAO* perspective the time required for reloading the instruction cache and the data cache is interesting (Kopetz 2011, 243-248).

Cache thrashing is a phenomenon that occurs when two different caches on the same level contain and work on a copy of the same part of the main memory. This can be detrimental to performance, as the data constantly has to be fetched and written into the first cache level shared by the two caches. An example using Figure 2.3: this could happen if two of the L1 caches that do not share the same L2 cache work on the same part in the main memory. Then every write to any of the two L1 caches would require the request to be transmitted to the L3 cache.

## 2.2 First In, First Out Scheduling

A common scheduling algorithm for real-time tasks is a policy called first in, first out; abbreviated *FIFO*. This policy may seem simple, but this simplicity makes it quite potent and easy to implement. The basic concept of the *FIFO* algorithm is that whichever task arrives first, gets to execute first, and gets to run until finished. Figure 2.4 illustrates this concept. All three threads in this figure have equal priority and a runtime of six time units each. The first thread arrives at $0T$, the second arrives between $1T$ and $5T$ and the third arrives after the second thread. The figure shows that the first arriving task is the first to be scheduled and gets to run before the others.

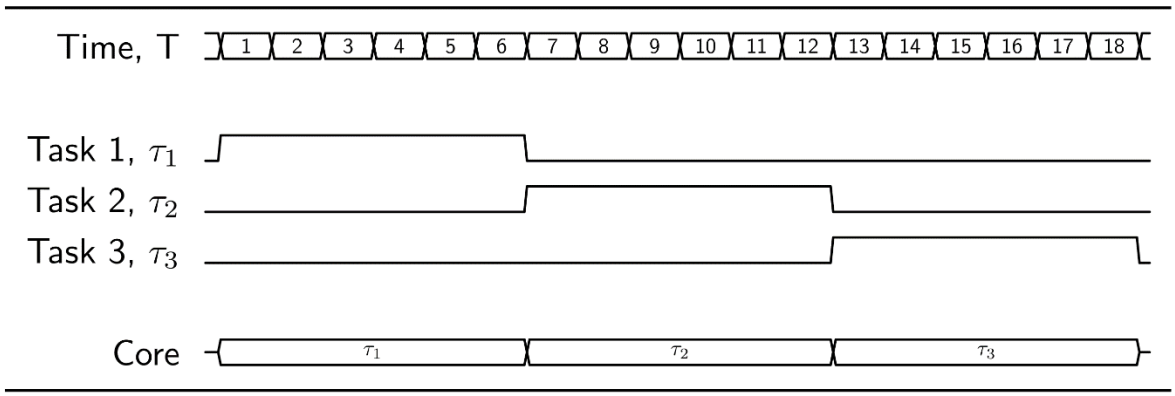


*Figure 2.4: A task set consisting of three tasks scheduled with the FIFO algorithm*

Dellinger, Garyali and Ravindran (2011) showed that the *FIFO* scheduling policy implemented on their system suffered a very small overhead compared to the other implemented policies. It also performs relatively few task migrations for a small task set, performing slightly worse for larger sets of tasks – a trend Dellinger, Lindsay and Ravindran also showed in 2012. However, Dellinger, Garyali and Ravindran (2011) showed that for their *RT-OS* implementation, tasks that run to meet deadlines start to miss them even for low numbers of threads. Global *FIFO* also provides low schedulability if considering periods and deadlines (Dellinger, Lindsay and Ravindran 2012).

The *FIFO* algorithm is not a fair algorithm. Tasks with long runtimes never need to yield the *CPU*; they tend to hog the *CPU* for a long time. However, as long as there are no preemptions from higher priority tasks there should be no context switches, which is good for overhead and from a cache perspective. Another advantage with *FIFO* is that the scheduling is $\mathcal{O}(1)$ complex (Dellinger, Lindsay and Ravindran 2012), meaning that scheduling tasks on a system already running a lot of other tasks costs no additional overhead than scheduling a task on an empty system. It is also independent of the number of cores that the system is running (Dellinger, Lindsay and Ravindran 2012).

## 2.3 Round Robin Scheduling

The round robin algorithm, *RR*, is another real-time scheduling algorithm. It assigns a limited amount of processor time to each task in the system. This time interval is essential for the *RR* algorithm and it is called a time slot or a time slice. Preemption occurs when the time slot

expires and the running task is not finished. This *RR* concept is illustrated in Figure 2.5. All three threads in this figure have equal priority and a runtime of six time units each. The time slot used in the example is three units of time. The figure shows that the *CPU* is switching task every time the end of a time slot is reached, even though the task is not finished. The length of the time slot is for this reason an interesting issue. It is a deciding factor for the performance of the *RR* scheduling algorithm. A too short time slot results in many context switches and a lower *CPU* efficiency, while a too long time slots may result in a behaviour similar to *FIFO*, missing the whole time slice concept.

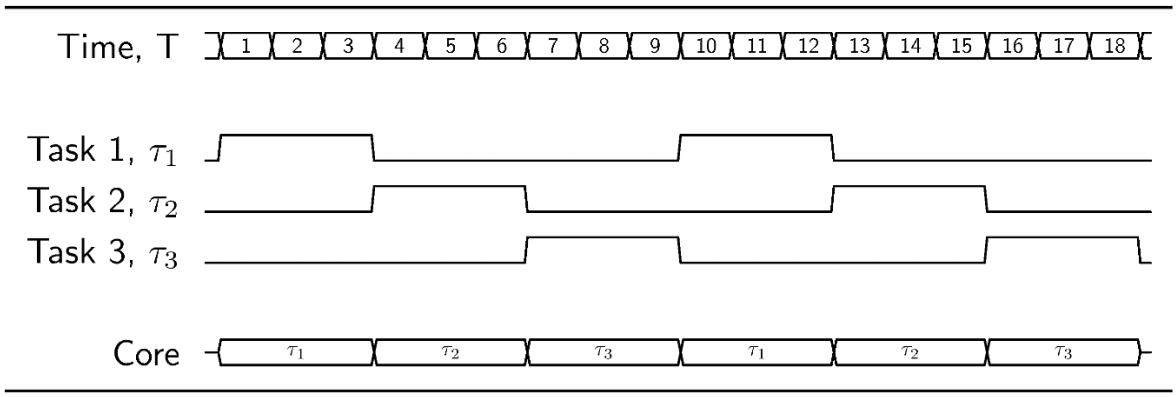


*Figure 2.5: A task set consisting of three tasks scheduled with the RR algorithm*

The *RR* algorithm is a fairer real-time scheduling algorithm than the *FIFO* algorithm since it uses these time intervals. Every active task gets processor time relatively early. Short tasks therefore have the possibility to finish without a long waiting time. *RR* has nevertheless some drawbacks. It has in general a large waiting time and response time. It performs many context switches and it has a low throughput. A low throughput means that the number of tasks completed per time unit is small. The high context switch rate results in a large administrative overhead since the state of the task is stored either in the stack or in a register. These drawbacks affect the system performance negatively. The *RR* algorithm has one more good property though; its scheduling is $\mathcal{O}(1)$ complex (Yuan and Duan 2009) similar to *FIFO* scheduling.

## 2.4 Earliest Deadline First Scheduling

Earliest deadline first (*EDF*) is a scheduling algorithm that attempts to schedule tasks so that they meet their deadlines. The task with the closest absolute deadline will get to run first. Assuming *CPU* utilization less than 100 %, negligible preemption costs and a deadline smaller than the period, this policy guarantees that all tasks on a single core system will meet their respective deadlines. (Baruah, Bertogna and Buttazzo 2015, 29)

On a single core system, the basic idea behind the *EDF* scheduling algorithm is that the scheduler is provided with the deadline of all real-time tasks using the *EDF* policy in the system. The task with the deadline closest to the current time will be allowed to run. This is illustrated in Figure 2.6. The threads in the example are all arriving at the same time, at $0T$, and have equal runtime. The first thread has its deadline at $14T$, the second at $18T$ and the

third at 16*T*. The figure shows that the task with the earliest deadline is the first to be scheduled. In this case, all three tasks meet their deadlines.

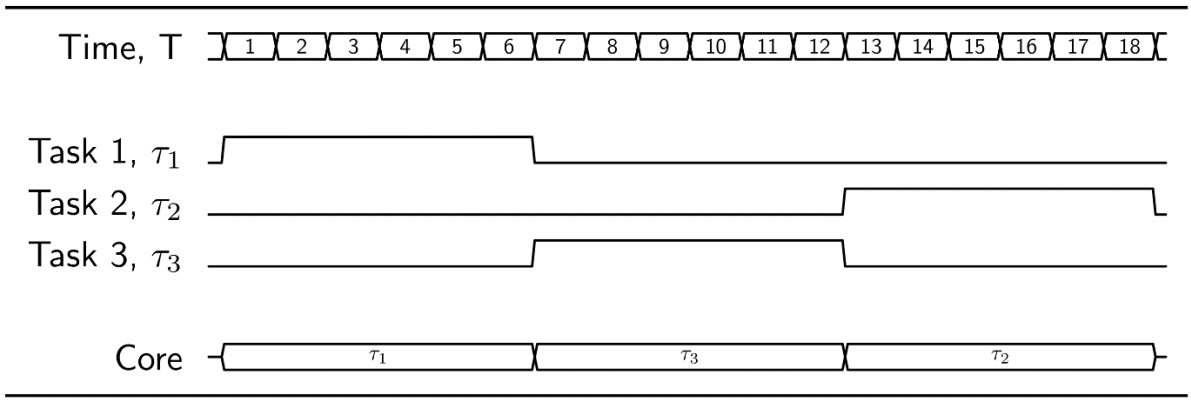


*Figure 2.6: A task set consisting of three tasks scheduled with the EDF algorithm*

On a multicore system, there are several different variants of the *EDF* scheduling algorithm. The one used in Linux is called global *EDF* or *GEDF*. It does the deadline based scheduling combined with a constant bandwidth server on several cores and it needs the deadline, the runtime and the period of all *RT* tasks using the *GEDF* policy in the system. In this context, the deadline is the relative deadline. According to Kerrisk (2015), it is usual to set the runtime larger than the *WCET* in a hard *RT* system, to make sure the tasks finish before their deadline. The runtime should at least be larger than the task's average computation time regardless of the *RT* configuration (Kerrisk 2015). The period is the shortest possible time before the task can start execute again. If the period is set to zero, it is defaulted to the same value as the deadline (Kerrisk 2015).

As in the case with *EDF*, *GEDF* allows the task with the closest absolute deadline to run first. Then the task with the second closest absolute deadline, and so on until all cores in the *CPU* have a running task. This is the *GEDF* algorithm's behaviour. An example of a task set scheduled with *GEDF* can be found in chapter 2.5.3 – *SCHED_DEADLINE*. With *GEDF*, the tasks may migrate between the cores – however, it does not guarantee that tasks meet their deadlines, like the normal *EDF* does. A proof of this is shown in the example below. All three threads are arriving at 0*T*. The first and the second have their absolute deadline at 6*T*, while the third has its deadline at 7*T*. The figure shows that the third task miss its deadline. Figure 2.8 illustrates that there exists a feasible schedule, not using *GEDF*.

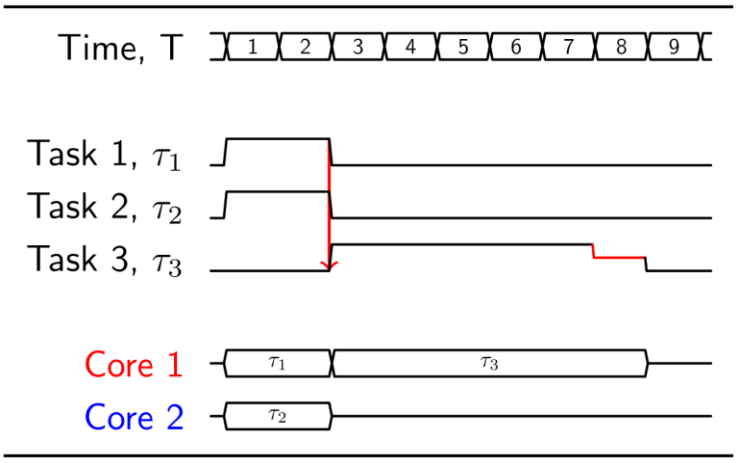


*Figure 2.7: A task set consisting of three task scheduled on two cores with GEDF*

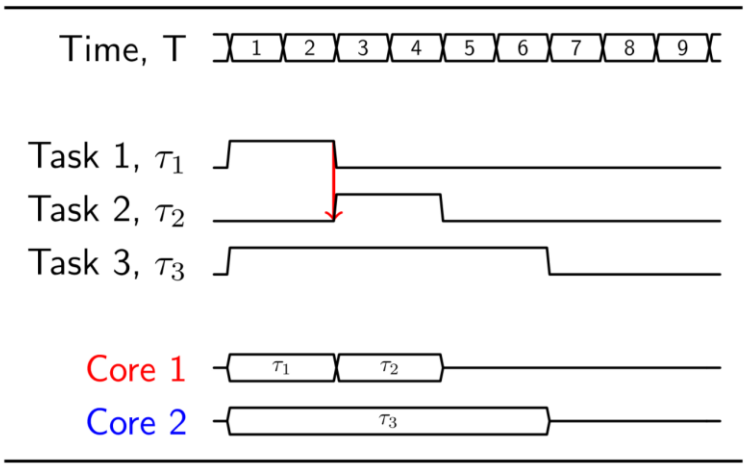


*Figure 2.8: A possible schedule, not using GEDF*

However, if any set of tasks fulfils the two criteria in Equation 2.2 and 2.3, the task set will be schedulable with *GEDF*. This means that all tasks in the task set are guaranteed to meet their deadlines. In the two equations, $U_{sum}$ is the sum of all processor utilization, $m$ is the number of hardware threads and $U_{max}$ is the maximum utilization of any single task. Equation 2.3 indicates that no task can have a *WCET* longer than half of its relative deadline. (Baruah, Bertogna and Buttazzo 2015, 78)

$$U_{sum} \leq \frac{m+1}{2} \tag{2.2}$$

$$U_{max} \leq ½ \tag{2.3}$$

Tasks scheduled with *EDF* or *GEDF* continue to run until they are finished, unless they explicitly yield the processor, or unless they are preempted by a task with a closer deadline. A sleep and a mutex lock are two examples of code, which yield the processor. If two tasks that arrive at the same time instance have the same deadline, random chance decides which task will be allowed to run first (Baruah, Bertogna and Buttazzo 2015, 29).

An advantage with *FIFO* and *RR* is that their scheduling complexity is $\mathcal{O}(1)$. However, *EDF* is also $\mathcal{O}(1)$ complex, at least running on a single core system. The complexity of *GEDF* is $\mathcal{O}(m)$, where $m$ is the number of tasks in the system (Dellinger, Lindsay and Ravindran 2012). This is because *GEDF* has to check the new task against all running tasks. Due to its larger complexity, the *GEDF* algorithm may generate more overhead than the others.

According to Brun, Guo and Ren (2015), *EDF* does not seem to increase the number of times the tasks are preempted when the processor utilization is increased. However, if there is a large difference in the execution time between the tasks, the number of preemptions seem to increase slightly (Brun, Guo and Ren 2015).

## 2.5 Linux Implementation

The Linux scheduler is part of the kernel, which decides the execution order of all active threads. To determine the execution order of the available threads each thread needs at least an assigned scheduling policy and an assigned static priority. Conceptually, the scheduler creates a ready queue for each priority level that contains all threads with that specific priority. It is the thread's scheduling policy, which determines where it will be inserted into the list with equal priority when the thread activates. It sets a dynamic priority on each thread inside this list. The scheduler gives execution time to the task with the highest dynamic priority inside the first nonempty list with highest static priority. (Kerrisk 2015)

The Linux kernel framework consists of different scheduling policies. SCHED_NORMAL, SCHED_BATCH and SCHED_IDLE are the normal policies and for them the static priority must be specified as zero since this priority is not used (Kerrisk 2015). All three policies are completely fair algorithms (Milic and Jelenkovic 2014). SCHED_NORMAL is based on time-sharing and is the default scheduling policy in Linux. It is intended for all threads that do not require real-time response. This policy increases the dynamic priority of threads that are denied to run by the scheduler even if they are ready (Kerrisk 2015). SCHED_OTHER is the traditional name of this policy, but since version 2.6.23 of the Linux kernel the SCHED_OTHER policy have been replaced with the SCHED_NORMAL policy (GitHub 2013). The SCHED_BATCH policy is intended for scheduling batch processes and SCHED_IDLE is used for scheduling background tasks with very low priority (Kerrisk 2015).

Besides the normal scheduling policies, the Linux kernel framework provides several real-time scheduling policies. SCHED_FIFO and SCHED_RR have a static priority higher than the non-real-time policies mentioned above. The range of the priority levels is between 1 and 99, where 99 is the highest static priority. This implies that an arriving thread with a SCHED_FIFO or a SCHED_RR policy always preempts a currently running normal thread. (Kerrisk 2015)

SCHED_DEADLINE is another real-time scheduling policy. For a thread running under the SCHED_DEADLINE policy, the static priority needs to be zero. However, in this case it does not mean that SCHED_FIFO and SCHED_RR threads will preempt SCHED_DEADLINE threads. Instead, the SCHED_DEADLINE policy automatically has the highest priority. Therefore, it will preempt all threads scheduled under one of the other polices. Today, only one kind of task runs before SCHED_DEADLINE and it is the stop task (GitHub 2015a),

typically used by interrupts. The SCHED_DEADLINE policy contains a pointer which points on the *RT* scheduling class in order to know which policy that will be allowed to execute next, and so on (GitHub 2016).

To avoid real-time threads to starve all threads scheduled with the normal scheduling policies, a mechanism called real-time throttling is implemented in the Linux kernel. This mechanism defines how much of the *CPU* is allowed to be used by the *RT* threads (Abeni, Lipari and Lelli 2014). In the testing platforms used in this project, this part is up to 95 %. This means that the *RT* throttling mechanism prohibits *RT* threads to starve all lower priority threads in the system, since the remaining 5 % always will be used for the normal scheduling policies.

SCHED_FIFO, SCHED_RR and SCHED_DEADLINE have different ways to set their dynamic priorities. These differences are described in the subchapters below.

### 2.5.1 SCHED_FIFO

SCHED_FIFO is an implementation of the first in, first out (*FIFO*) algorithm in Linux and it follows the *POSIX* standard, the *IEEE* 1003.1-2008 standard. This standard considers priority. When a task arrives, it is inserted into the end of the ready queue associated with its static priority. The static priority is an input attribute to the task assigned by the user. If the recently arrived task has a higher assigned static priority than the one currently running, it preempts the current one. If it has the same or lower static priority, the current one keeps running. According to the *POSIX* standard, tasks scheduled with SCHED_FIFO can get their static priority changed when they are running or are runnable. If the priority is increased, the task is pushed to the back of the ready queue associated with the new priority. If it is decreased instead, the thread is pushed to the head of the ready queue associated with the new priority. The standard also includes standardized function names in order to access the *FIFO* policy. (IEEE and the Open Group 2013)

In Linux the *FIFO* scheduling policy is a real-time policy, meaning that even SCHED_FIFO threads with the lowest priority are still scheduled before the normal scheduling policies for non-real-time tasks (Kerrisk 2015). No other information than a static priority and a process *ID* is required by the scheduler in order to schedule the tasks.

The figure below is an example of eight tasks running on two *CPU* cores scheduled with the *FIFO* policy. Since there are two cores in the system, global *FIFO* is used. To understand the figure, information about the arrival time, the runtime and the priority for each task is needed. This information is provided in Table 2.1. Task one starts to run on the first core when it activates since there are no other ready tasks in the system. The same thing happens on the second core for task two when it activates. These two threads should have continued running until they were finished if not thread four had been activated at 3$T$. Preemption of task one happens due to the higher priority of task four. At 7$T$, when they are finished, task one is allowed to resume running at the same time as task six (the last thread with the highest priority) is given processor time at the second core. The scheduler schedules the remaining tasks without confusion after this time instant. The task arrived first is executed first and runs until finished. If task three has some real-time deadlines, then task one could cause task three to miss it.

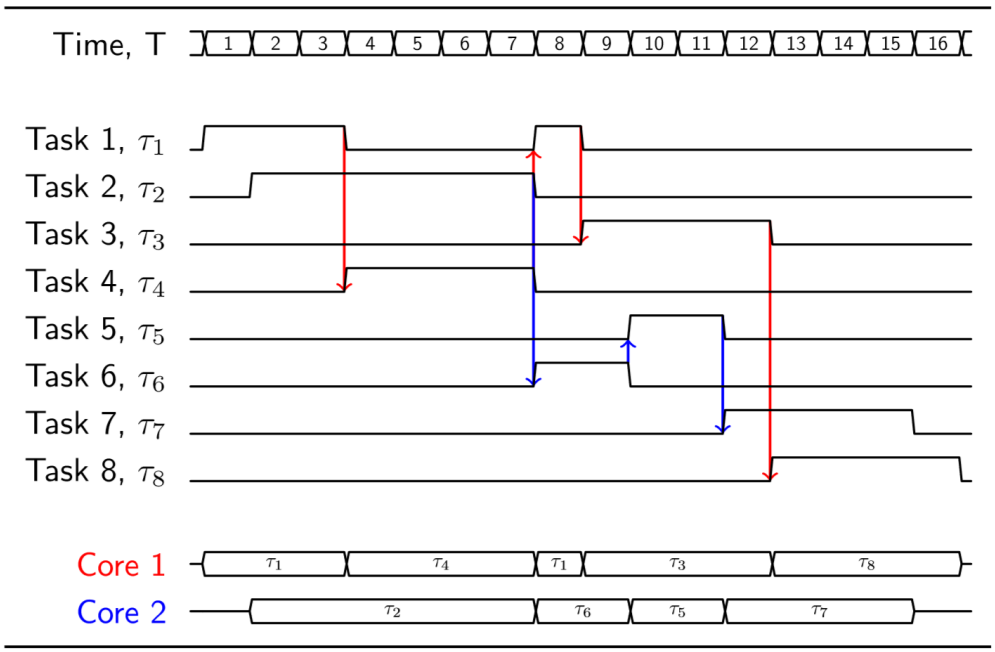


*Figure 2.9: A task set consisting of eight tasks scheduled on two cores with the FIFO policy*

| Task | Arrival time | Runtime [TU] | Priority |
|---|---|---|---|
| Task 1 | 0 | 4 | 1 |
| Task 2 | 1 | 6 | 2 |
| Task 3 | 2 | 4 | 1 |
| Task 4 | 3 | 4 | 2 |
| Task 5 | 4 | 2 | 1 |
| Task 6 | 7 | 2 | 2 |
| Task 7 | 6 | 4 | 1 |
| Task 8 | 7 | 4 | 1 |

*Table 2.1: Task parameters for the task set scheduled in figure 2.9 and figure 2.10*

### 2.5.2 SCHED_RR

SCHED_RR is an implementation of the round robin (*RR*) algorithm in Linux. It is identical to the SCHED_FIFO policy in the priority perspective (IEEE and the Open Group 2013). The task with the highest priority is always scheduled first and the task of a certain priority that arrives first is located first (head) in the ready queue associated with its static priority (IEEE and the Open Group 2013). It even behaves exactly like the SCHED_FIFO policy when there is only one task with the highest static priority; the running task is then executed until it is finished (Milic and Jelenkovic 2014). Otherwise, when there is more than one task with the same (highest) static priority, these tasks will be alternated by the SCHED_RR policy. However, this requires a system with fewer processors than the amount of threads in that specific priority list, for example a uniprocessor system. On a multiprocessor system with a hardware thread number greater or equal to the amount of tasks in the highest static priority list, these actual tasks will be running in parallel, each task on its own hardware thread (Milic and Jelenkovic 2014).

The thread alternation is a significant property of the SCHED_RR policy. When a process has run for a while, it is pushed to the back of its ready queue (IEEE and the Open Group 2013). This time before a swap happens is called a time slot or a time slice. The basic principle with the time slices is described in chapter 2.3 – *Round Robin Scheduling* together with a simple example of alternating treads. If the time interval is not specified, the default value is 100 ms in Linux (GitHub 2015b).

Figure 2.10 shows an example of how *RR* scheduling works when there are tasks with different static priority in the system. It is a global round robin implementation running on two cores using the time slice of two units of $T$. The task parameters are the same as in the *FIFO* example above, to illustrate the differences between the two algorithms. They are therefore specified in Table 2.1. Task one has the earliest arrival time and is therefore given processor time first. It runs for two units of $T$ since this is the length of the time slice in the system. The first core is switched to task three at this moment, but after one time-unit, it is preempted by task four due to its higher priority. Here, at this point, is an example of two threads with the highest priority running in parallel. They get to execute until finished since no other task with greater or equal priority is activated. At $7T$ the scheduler is allowing the third task to resume, since it has not consumed all its runtime. At the same instant, the recently activated task six is given processor time on the second core, because it has the highest priority. Next time unit, at $8T$, when the third task has used its remaining time of the time slice it starts to get interesting. One can see that the first task gets processor time instead of task five, which has the same priority. It happens because of the order in the ready queue for the lowest priority level. Task one is pushed to the end of the priority list in the very first context switch. Since this time instant is before the time instant when task five activates, task one is pushed to the head of the ready queue and is therefore scheduled first of the two threads. Now, at $9T$, the figure is not showing any new uncertainties. The scheduler schedules the remaining tasks in the round robin fashion.

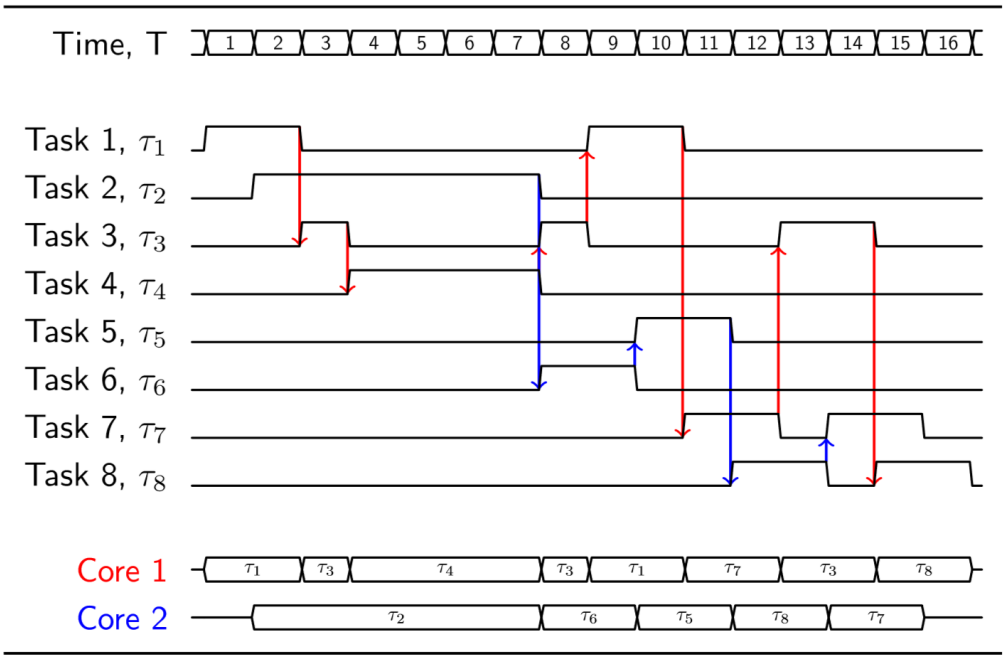


*Figure 2.10: A task set consisting of eight tasks scheduled on two cores with the RR policy*

### 2.5.3 SCHED_DEADLINE

SCHED_DEADLINE is a new real-time scheduling policy which is implemented in the Linux kernel since version 3.14 (Stahlhofen and Zöbel 2015; Abeni, Lipari and Lelli 2014). It is an implementation of the algorithm earliest deadline first, *EDF*, combined with *CBS* (GitHub 2015c). *CBS*, or constant bandwidth server, is an algorithm that provides the tasks with a temporal protection (Abeni, Lipari and Lelli 2014). This means that the tasks do not interfere with each other and their behaviour is isolated. Since this is a key feature of the SCHED_DEADLINE scheduling policy, the *CBS* algorithm will be described further down in this subchapter.

This new scheduling policy includes both single core and multicore processor scheduling. When the platform is a multiprocessor system, global *EDF* scheduling is applied. SCHED_DEADLINE is the first real-time scheduling policy which does not use the static priority (it needs to be zero, as mentioned above) system specified by the *POSIX* 1003.1b standard. Instead, it uses dynamic priority. This priority is set with respect to dynamic timing properties, such as deadlines. (Stahlhofen and Zöbel 2015)

As described in chapter 2.4 – *Earliest Deadline First Scheduling*, the *EDF* algorithm only needs information about the task's relative deadline, runtime and period. The scheduler computes the task's absolute deadline every time the task activates. Since the *EDF* algorithm selects the task with the earliest absolute deadline to be executed next, this task has the highest priority. This means that the priority of all active tasks in the system may update when another task becomes active. This depends on the computed deadline of the task that wakes up. The priority of tasks scheduled by the SCHED_DEADLINE policy is therefore dynamic. (GitHub 2015c)

Figure 2.11 shows an example of eight tasks running on two *CPUs* scheduled with the SCHED_DEADLINE policy. It is an illustration the *GEDF* algorithm's behaviour, since none of the tasks misbehaves in the example. Information about the task set is provided in Table 2.2. It contains the absolute deadline in addition to the relative deadline, which is supplied to the scheduler. The absolute deadline is computed by $absolute\ deadline = arrival\ time + relative\ deadline$. Task one and task two start to run directly when they are activated (at $0T$ and $1T$) since there are no other ready tasks in the system. When task three arrives, at $2T$, task one is preempted due to the closer absolute deadline of task three. The same thing happens when tasks four and five arrive. It is always the running task with the furthest absolute deadline that is preempted. Since no other task activates with a closer absolute deadline, these two tasks get to run until they are finished. When a task finishes, the active task with the closest absolute deadline is allowed to run. At $8T$, there are no rules on which of task two or eight will get processor time first. In this example thread eight is lucky, but it does not matter since both of them will meet their deadlines anyway. The same thing happens at $12T$. In this case, task one could cause task seven to miss its deadline, if it had been allowed to run first. This is an example of the problem with *GEDF* described in chapter 2.4 – *Earliest Deadline First Scheduling*.

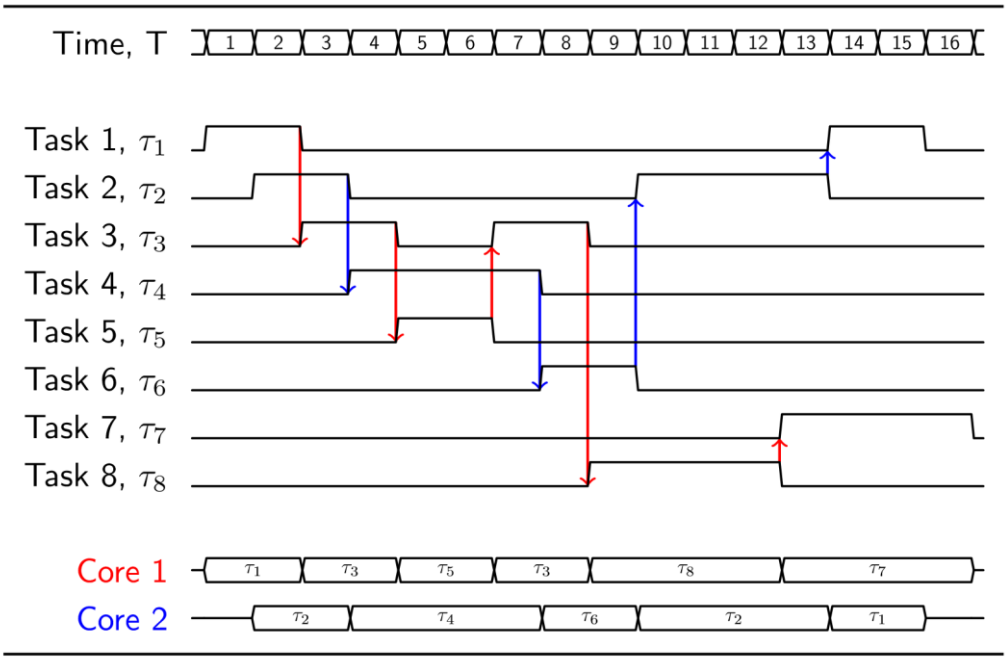


*Figure 2.11: A task set consisting of eight task scheduled on two cores with GEDF*

| **Task** | **Arrival time** | **Runtime [TU]** | **Relative deadline [TU]** | **Absolute deadline** |
|---|---|---|---|---|
| Task 1 | 0 | 4 | 16 | 16 |
| Task 2 | 1 | 6 | 14 | 15 |
| Task 3 | 2 | 4 | 12 | 14 |
| Task 4 | 3 | 4 | 8 | 11 |
| Task 5 | 4 | 2 | 4 | 8 |
| Task 6 | 7 | 2 | 4 | 11 |
| Task 7 | 6 | 4 | 10 | 16 |
| Task 8 | 7 | 4 | 8 | 15 |

*Table 2.2: Task parameters for the task set scheduled in figure 2.11*

In (Lelli et al. 2012) the authors compare *RMS* to different variants of *EDF*. In the report they use the SCHED_DEADLINE policy implemented in the Linux kernel. They compare SCHED_DEADLINE with *RMS* with regards to utilization, cache misses, context switches and migrations. In neither of these categories does SCHED_DEADLINE perform significantly worse than the offline scheduling implemented in *RMS*.

The time complexity for scheduling new tasks is $\mathcal{O}(\log(n))$ for SCHED_DEADLINE, where $n$ is the number of tasks the system is running. (Stahlhofen and Zöbel 2015)

**Constant Bandwidth Server (CBS)**
When the scheduler computes the task's absolute deadline, the *CBS* algorithm is used. It is the *CBS* algorithm which assigns the absolute deadlines to the task's jobs, by adding the arrival time with the relative deadline. The purpose of this algorithm is to ensure that each task will run for at most the desired runtime during its desired period and to protect it from other tasks. (GitHub 2015c)

The *CBS* is a reservation based scheduling algorithm. It reserves execution time for the task every period. It also guarantees that the running task gets to execute in this amount of reserved runtime, while the other tasks in the system are not allowed to interfere. This protection is a temporal isolation and means that the worst-case behaviour of a task does not depend on the other tasks' behaviour. The *CBS* algorithm may also allow tasks to consume some of their future reserved runtime, but this earlier execution is only allowed if there is enough idle time in the system. (Abeni, Lipari and Lelli 2014)

*CBS* is another alternative to real-time throttling. These two accomplish the same thing. The main difference is that *RT* throttling lacks a strong theoretical support according to Abeni, Lipari and Lelli (2014) and that it appears on a higher level (GitHub 2015c). It is therefore not possible to make a complete schedulability analysis with *RT* throttling. *CBS* however, allows a predictable execution and makes sure that no task will consume all *CPU* time simultaneously (Abeni, Lipari and Lelli 2014). The *CSB* algorithm still throttles the tasks if they should misbehave, for example by soft locking. In this case, the throttling occurs when the task has consumed all its reserved runtime within a specific period (Kerrisk 2015). It will then be suspended until it reaches its period, where it will get access to more computation time. At this point, when the throttled task reaches its next period, the *CBS* algorithm updates and reserves the runtime in the new period to the sum of the remaining runtime for the suspended task and the runtime in the new period (GitHub 2015c). The new absolute deadline is calculated as the sum of the last absolute deadline and the period.

To check if the absolute deadlines are respected an admission control is performed by the *CBS*. It is the guarantee foundation of the *CBS* algorithm. It checks if the task set is schedulable every time a SCHED_DEADLINE task arrives or changes. This means that it checks if the current task would push the *CPU* utilization above 100 %; if it does, then the task is rejected. The admission control test for a SCHED_DEADLINE task is partially based on the utilization of the system and partially on the task's bandwidth. In this context the bandwidth is the ratio $runtime/period$, which is the same as utilization of a real-time task scheduled by the SCHED_RR policy or the SCHED_FIFO policy. The admission control checks if the newly activated task has the right parameter characteristics. If $runtime > deadline$, the current task will be rejected by the admission control. Instead, if the runtime is within the deadline, and the deadline is within the period (Equation 2.4), the admission control has passed and the *EDF* algorithm can schedule the task. Assuming a system utilization under 100 %. (GitHub 2015c)

$$runtime \leq deadline \leq period \tag{2.4}$$

To understand the practical importance of the *CBS* algorithm, one example of the scheduling differences with and without *CBS* is given in Figure 2.12 and Figure 2.13. The two figures show the same task set with assigned runtime, deadline and period according to Table 2.3. However, the second task misbehaves (in both figures) in the second period, were it takes four time units to run.

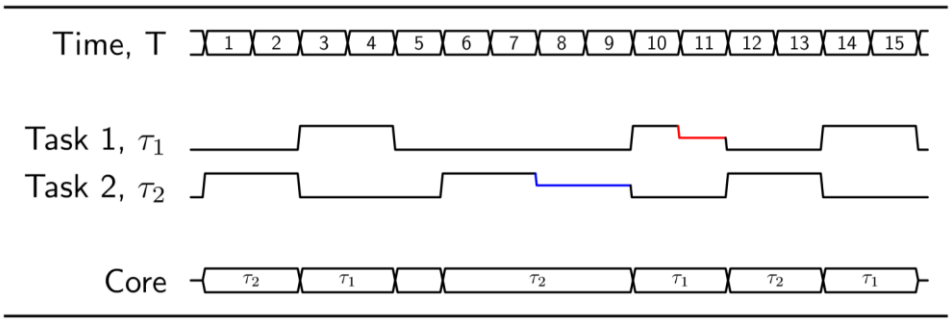


*Figure 2.12: A task set consisting of two tasks scheduled on one core with EDF*

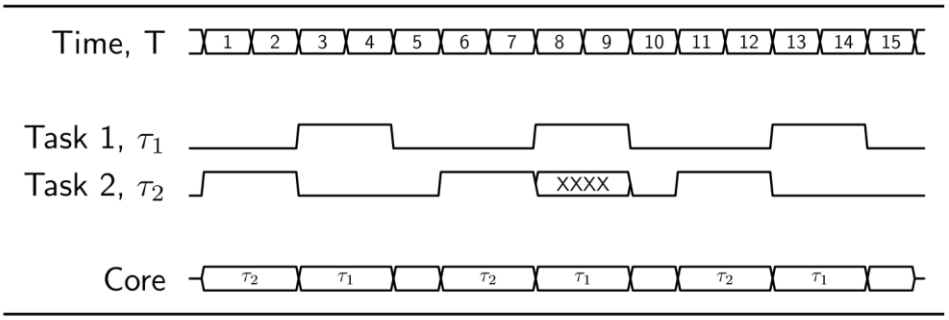


*Figure 2.13: A task set consisting of two tasks scheduled on one core with the SCHED_DEADLINE (EDF + CBS) policy*

| **Task** | **Runtime [TU]** | **Deadline [TU]** | **Period [TU]** |
|---|---|---|---|
| Task 1 | 2 | 5 | 5 |
| Task 2 | 2 | 4 | 5 |

*Table 2.3: Assigned task properties for the task set scheduled in figure 2.12 and figure 2.13*

The scheduler behaves differently, as can be seen in the two illustrations above. In the first figure, the second task is allowed to run the two extra (blue line) time units. The consequence of this is illustrated with the red line; task one misses its deadline. In the second figure task two is aborted early since it attempts to consume more processor time than it was assigned by the scheduler. This allows the first task to run without interference. The *CBS* mechanism is therefore used to prevent other tasks from suffering when one task is misbehaving. It makes sure that no task will consume all of the *CPU*. In order for this to work however, no other scheduling policy can be allowed to take processor time as this will cause the admission control to fail. For this reason, the SCHED_DEADLINE policy in Linux takes priority over other real-time policies.

# Chapter 3

# Method

In this chapter, the scheduler evaluation methodology is described in detail. First, the testing platforms are introduced – the tests have been performed on two different platforms. Second, the implementation is walked through. A synthetic test program has been developed in order to test the properties of the scheduling polices in Linux. The evaluation methods are presented, both for the test program and for an application provided by Ericsson. Finally, a subchapter about how the results are illustrated in figures can be found.

## 3.1 Testing Platforms

The tests have been performed on two different platforms. The first environment was a virtual machine running on Windows 7 Enterprise. This machine contained the Linux operating system in order to get access to the Linux kernel and its real-time scheduling policies. The kernel version used was 4.4.0-22-generic and the virtual machine had access to two of the four hardware threads in the main computer. The hardware used in this platform was Intel® Core™ i5-4310U *CPU* @ 2.00 GHz, 2 GB *RAM* and a last level cache size of 6 144 KB. On top of this, the high performance power option in Windows was used for all tests on this platform.

The second environment was a more powerful machine, containing 32 hardware threads. However, only eight hardware threads were used during test runs. It was because the used hardware threads were powerful enough so that any more would cause the system to be underutilized, even under large load, thus defeating the purpose of the utilization tests. The hardware used in this platform was Intel® Xeon® *CPU* E5-2667 v3 @ 3.20 GHz, 32 GB *RAM*, and a last level cache size of 20 480 KB. Table 3.1 summarizes and compares these two platforms. It also contains version numbers of various applications used in this project.

In chapter 2.5.2 – *SCHED_RR*, one can learn that the time slice used for *RR* scheduling in Linux operating systems is by default 100 ms unless otherwise specified. In Ubuntu (at least in the versions used in this project) the time slice is specified as 25 ms. However, since this is long time compared to the computation time of the jobs in the test cases, a 1 ms time slice is used in this project. This value is the lowest time slice possible in kernel version 4.4.0-22-generic.

| | **Virtual machine** | **Monster machine** |
|---|---|---|
| **Operating system** | Linux | Linux |
| **Linux version** | Ubuntu 16.04 LTS (64-bit) | Ubuntu 14.04.4 LTS (64-bit) |
| **Kernel release** | 4.4.0-22-generic | 3.19.0-51-generic |
| **Number of *CPUs*** | 1 | 2 |
| **Number of cores per *CPU*** | 2 | 8 |
| **Number of threads per core** | 2 | 2 |
| **Processor model** | Intel® Core™ i5-4310U *CPU* @ 2.00 GHz | Intel® Xeon® *CPU* E5-2667 v3 @ 3.20 GHz |
| **RAM** | 2 048.408 MB | 32 700.408 MB |
| **Cache size** | 6 144 KB | 20 480 KB |
| **gcc version** | 5.3.1 | 4.8.4 |
| **LTTng version** | 2.7.1 | 2.7.2 |
| **Python version** | 2.7.11+ | 2.7.6 |

*Table 3.1: Platform specifications*

## 3.2 Implementation

In order to compare the three scheduling policies (SCHED_FIFO, SCHED_RR and SCHED_DEADLINE) in Linux, a program creating treads and setting the real-time characteristics for them was needed. A test program was therefore implemented. How it works is described in the subchapter below. The program is generic; it takes a file containing the task set it should create (including the tasks' parameters) as input. This makes it easy to change the task set, and create different test cases. The test cases used in this project can be found in the second subchapter below. After this, the method used to track the program is described. Helper programs created to make the test cases easy to run are also described.

### 3.2.1 The test program

In order to test the properties of the relevant policies several tests had to be performed. Tasks that behaved in a desired way had to be generated, so that the algorithms could be evaluated. A test program was developed to achieve this. It was able to mimic properties like period, runtime and delay and it was easily reused for different task sets.

The main program started reading the desired task set from an input file, and initiated the desired threads. Then it waited until all threads were finished before shutting down. The duration of the threads was specified in the task set, together with other necessary task parameters, for example policy and priority. The created threads were *POSIX* threads, pthreads. These well-defined threads follow the *IEEE POSIX* standard by the open group. A single process can create multiple threads that all share the same address space, but each thread has its own stack. The standard also specifies that each thread has some properties such as its own process *ID*, its own scheduling policy and more. In Linux, each pthread also has its own *CPU* affinity.

The newly created pthreads slept for a period of time before starting their execution. This was done in order to allow the main program to create all pthreads; otherwise, the main thread

would be preempted by one of the newly created pthreads due to them having a higher (*RT/EDF*) priority from their policies. When the pthreads were created, a check was performed. In the case of SCHED_FIFO and SCHED_RR, the check verified if the thread parameters were valid. In the SCHED_DEADLINE case, an admission control was performed. If it failed, the pthread was forced to sleep until the other threads were finished, then it was allowed to exit.

While running, the pthreads calculated their next period. This corresponds to the fourth blue box from the top in Figure 3.1. This flow chart illustrates the states of one and a half pthreads. The schedule state contains the initiation of a pthread and it is a simplification since it assumes that the thread is allowed to be scheduled. After the next period was calculated, the work portion of the pthread was performed. This work was a series of *NOP* instructions. Then it slept until its next period. If the next period had already begun when the work was finished, the pthread called a yield command to surrender the processor to another pthread. It calculated a new period when it woke up. If the calculated period begun after the desired stop time, the pthread shut down. At this moment, the pthread is in the end state as shown in the figure below.

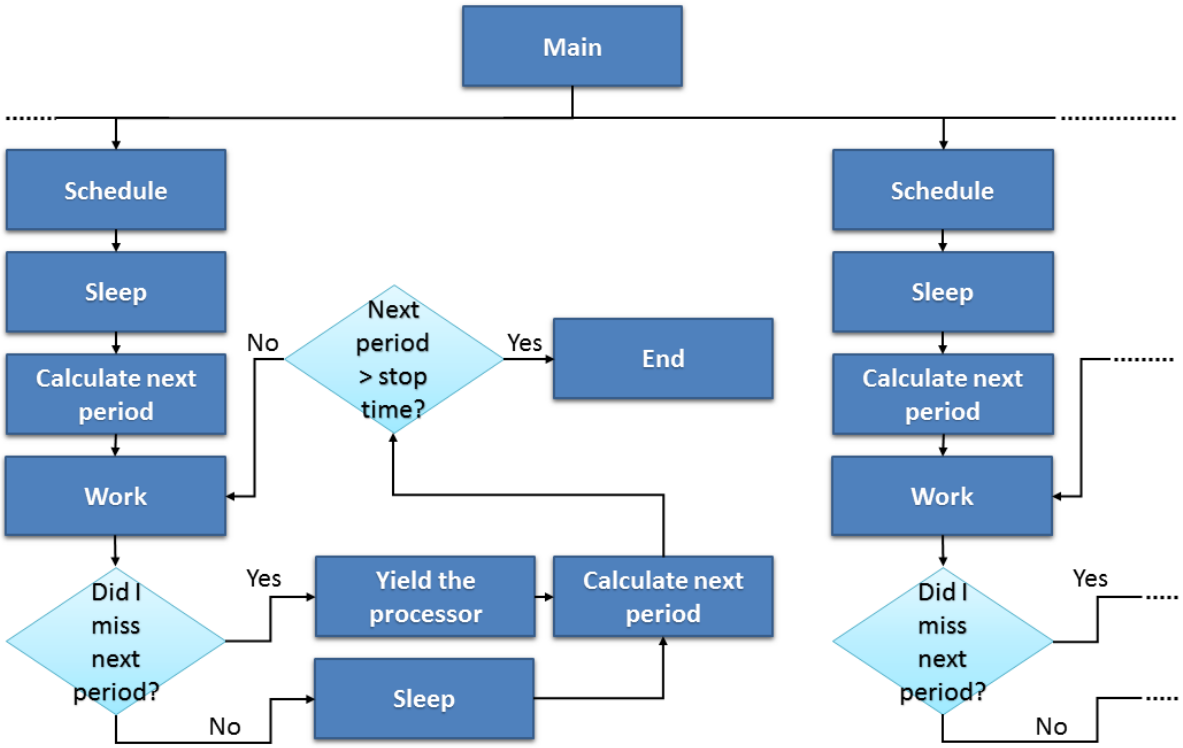


*Figure 3.1: Flow chart of the program flow*

### 3.2.2 Test cases

There were three different test cases, or sets of tasks, where all have been performed with the three relevant scheduling policies. In order to evaluate the performance of the *RT* policies with some kind of static priority, some of them have also been performed with *RMS* implemented with *RR*. *RMS* was chosen for this because it was easy to implement and it was common in the studied theory. A duration of 10 s has been used for all test cases. The first of these test cases generated nine tasks per core, each aiming for a utilization of 10 %, which

resulted in around 90 % total utilization. The goal for this task set was to try the scheduler during heavy load. There was no examining of *RMS* for this task set due to it being the same as *RR* with all tasks having the same priority. The thread parameters in this test case are shown in Table 3.2. Three tasks of each type have been used per core. The priority in the table is the static priority, so there needed to be different values for the different policies. There is therefore more than one value in each priority field. The first value was used for SCHED_FIFO and SCHED_RR, the second for SCHED_DEADLINE and the third for *RMS*. The delay value is the time before a task gets to run the first time. It was used to stagger tasks and to make sure that some tasks were scheduled before others, particularly useful for SCHED_DEADLINE. In this test case, the delay was used to spread out the work. The number of *NOPs* is the number of no operation instructions in the work portion of each thread and the scheduler runtime is a parameter used by the SCHED_DEADLINE policy. The method to obtain the runtime value for the scheduler is described at the end of this subchapter. Periods were chosen as a multiple of the runtime in order to generate the desired utilization and the deadlines were always set to be equal to the period.

| **Parameter** | **Thread 1** | **Thread 2** | **Thread 3** |
|---|---|---|---|
| Number of *NOPs* | 100 000 | 100 000 | 100 000 |
| Deadline [ns] | Sch. runtime * 10 | Sch. runtime * 10 | Sch. runtime * 10 |
| Delay [ns] | 10 000 | Period * 1/3 | Period * 2/3 |
| Period [ns] | Sch. runtime * 10 | Sch. runtime * 10 | Sch. runtime * 10 |
| Priority | 10/0/- | 10/0/- | 10/0/- |
| Scheduler runtime [ns] | 294 520 | 294 520 | 294 520 |

*Table 3.2: Task parameters for test case A*

The second test case was a set of tasks aiming to emulate a master thread distributing work to slave threads. It was intended to emulate parts of the *LTE* application. A single thread with a short period and a short runtime, aiming to occupy around 20 % of the *CPU* time. This thread was supposed to be the master thread. Then there were fifteen threads with a long period and a runtime longer than the period of the master thread. Each of the threads consumed around 5 % of the *CPU* for a total of 95 % utilization. This task set was duplicated for every core. The thread parameters in this test case are shown in Table 3.3. The first thread is master, while the second and the third are slaves. The number of *NOP* instructions were chosen so that the thread supposed to imitate the master ran for a shorter time. This however, is not central to illustrate this test case. Delays were added so the master thread got to run before the slave threads woke up. At this instance, some of the slave threads were staggered.

| **Parameter** | **Thread 1** | **Thread 2** | **Thread 3** |
|---|---|---|---|
| Number of *NOPs* | 10 000 | 100 000 | 100 000 |
| Deadline [ns] | Sch. runtime * 5 | Sch. runtime * 20 | Sch. runtime * 20 |
| Delay [ns] | 10 000 | 100 000 | Period/2 |
| Period [ns] | Sch. runtime * 5 | Sch. runtime * 20 | Sch. runtime * 20 |
| Priority [ns] | 10/0/11 | 10/0/10 | 10/0/10 |
| Scheduler runtime [ns] | 64 286 | 325 622 | 325 622 |

*Table 3.3: Task parameters for test case B*

The third test case was designed to punish non-preemptive scheduling policies. It contained four different threads, each calibrated to be running around 20 % of the time. The first thread has a short runtime and a period five times as long as the runtime. The second thread had a runtime slightly longer than the first thread's period and a period five times as long, and so on for threads three and four. In theory, this should trigger a thread with shorter runtime to fail unless it does not preempt a thread with longer runtime, assuming they are running on the same core. This test was designed with the purpose of generating many context switches. The thread parameters in this test case is shown in Table 3.4. One thread of each type was used per core.

| **Parameter** | **Thread 1** | **Thread 2** | **Thread 3** | **Thread 4** |
|---|---|---|---|---|
| Number of *NOPs* | 1 000 | 90 000 | 600 000 | 4 000 000 |
| Deadline [ns] | Sch. runtime * 5 | Sch. runtime * 5 | Sch. runtime * 5 | Sch. runtime * 5 |
| Delay [ns] | 10 000 | 10 000 | 10 000 | 10 000 |
| Period [ns] | Sch. runtime * 5 | Sch. runtime * 5 | Sch. runtime * 5 | Sch. runtime * 5 |
| Priority [ns] | 10/0/13 | 10/0/12 | 10/0/11 | 10/0/10 |
| Scheduler runtime [ns] | 43 323 | 283 416 | 1 619 450 | 11 070 066 |

*Table 3.4: Task parameters for test case C*

In Figure 3.2, the three different test cases are clarified. The figure illustrates the runtime of the different thread types in relation to each other. It is important to notice that the scale is not accurate.

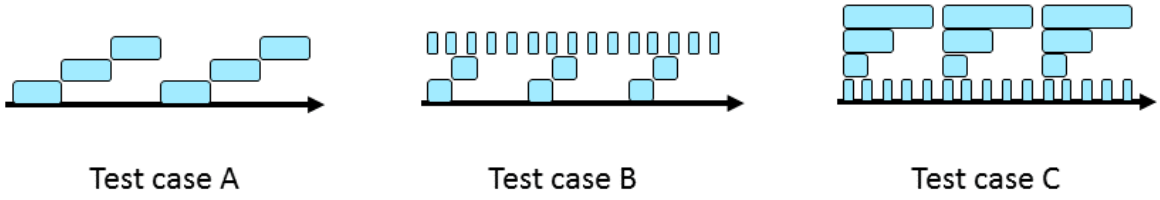


*Figure 3.2: Illustration of the test cases*

In order to correctly create the test cases described above, knowledge about the tasks' runtimes were required. They were approximated, as they differed a bit between different runs. The median runtime was collected for each task during a calibration run. In this run, SCHED_FIFO was used. The number of *NOP* instructions was set and each task was given a long period relative to its runtime. In order to keep the processor running for efficiency rather than power saving, the calibration was done with a program running a busy loop in the background. This program was not scheduled using a *RT* policy. This means that the threads being calibrated had a higher priority than the busy program. The new runtimes were entered into the task set and used to calculate values for some other parameters, see Table 3.2-3.4. This process had to be repeated for each new platform.

In the SCHED_DEADLINE test, the runtime has been alternated between two values. To separate these two, they were named *SCHED_DEADLINE median* and *SCHED_DEADLINE*

*median plus* later in the result chapter. The first one used the runtime calculated from the calibration run, while the second used a 10 % longer runtime.

### 3.2.3 Helper programs

To make the test cases easy to run, several tools were developed. First, the tests were running inside a bash script that enabled the trace tool, ran the test, and disabled the trace tool again. This made it possible to rapidly perform the tests and it kept the size of the log files small. A program was also written that could generate a large amount of parameters based of only a few. This program took a file with thread parameters and scaled it up based on a number assigned to each thread.

### 3.2.4 Linux kernel events

The kernel events in Linux are a way to analyse the kernel behaviour and most importantly, to know which processes are running on which kernel thread and when. To record these events, an open source framework called LTTng (LTTng 2014) was used. LTTng is a software, which can trace both kernel events and user applications at the same time. The events were recorded when LTTng was running and saved in a trace file. To view and analyse this trace file, a tool converting the trace format was needed. Babeltrace was therefore used. It creates a chronological list of all recorded events with timestamps from when they occurred. In Appendix A – *Output File from Babeltrace* an example can be found, illustrating what this list looks like. The next step in the analysis was to analyse the file created by Babeltrace. How this was done is described in chapter 3.3 – *Evaluation of the Test Program*.

Not all kernel events can be traced at the same time, since the trace file becomes very large, but most importantly, since depending on the trace throughput, some traces may be lost, making the trace file analysis erroneous. The following list of trace events was used in this project:

- sched_wakeup
- sched_wakeup_new
- sched_switch
- sched_migrate_task
- sched_process_fork
- sched_process_exec
- sched_process_exit
- All system calls

There were also contexts added to the trace file:

- vtid
- procname

Not all of the events above were used for evaluating the test cases. Sched_process_exec for example was only used to understand the behaviour of the tasks. The same goes for all the system calls, except syscall_entry_sched_yeild, which is called by the test program.

Sched_process_fork and sched_process_exit events were used to mark when all threads had been created and when the first thread was closing down.

## 3.3 Evaluation of the Test Program

The evaluation of the different test cases was based on the event list created with Babeltrace. This list was not comprehensible, though. A python program was therefore written, which read through the file (containing the events list) and stored timestamps for the relevant events. It computed various metrics and parameters, and plotted them in bar charts and histograms. Before the timestamps were stored, the event list was filtered. The start and the end of the file were removed, since only the periodic behaviour of the test program was relevant for this project. This means that all events before the last thread creation (sched_process_fork) was discarded, and also all events after the first task termination (sched_process_exit).

When the python program read through the file, it searched for a specific key word. This key word was the name of the application intended to be analysed, the application traced by LTTng. When the key word was found, the program checked if a relevant kernel event was performed and in this case, stored the timestamp for that event. These events were sched_wakeup/sched_wakeup_new, sched_switch, syscall_entry_sched_yeild and sched_migrate_task. The python program kept track of which event corresponded to which timestamp, and also the thread *ID* it was associated with.

After the timestamps were collected, they were processed. Kernel events from threads failing to schedule were removed. Switch in and switch out events occurring after the last wakeup call were removed too, on a thread specific level. Various metrics and parameters were then calculated. The runtime of the program was retrieved by subtracting $last\ wakeup - first\ wakeup$, with respect to all threads. The utilization $u_i$ for thread $i$ cannot be calculated as described in chapter 2.1.4 – *Metrics for performance evaluation*, since $T_i$ varies. The program runtime was used instead. By summing the computation time for all periods and dividing with the runtime of the program, the utilization $u_i$ could be retrieved. This computation can be seen in Equation 3.1. However, Equation 2.1 can still be used to calculate the total utilization $U$ over the entire task set on a uniprocessor system. On a multiprocessor system, $U$ needs to be divided by the number of cores over which the application is running on in the *OS*.

$$u_i = \frac{\sum_{j=0}^{n-1} C_{ij}}{program\ runtime} \quad , \quad n = number\ of\ jobs \tag{3.1}$$

The computation time was obtained in two steps. First, all runtime segments were calculated. A runtime segment is the time difference between a task's switch in event and its following switch out event. Second, all runtime segments within two consecutive wakeup calls (a period) was summarized. In other words, $C_{ij}$ was obtained by calculating the runtime in $T_{ij}$ for job $j$ of thread $i$.

Response time is another interesting metric to look at. It is the time between threads' wakeup events and their last switch out every period, see Equation 3.2. Response time, utilization and computation time were used to compare the characteristic differences between the three central scheduling algorithms in this project. Runtime segment and period were also plotted in histograms, but they were only used to verify that the test program was running as expected.

$$Response\ time = switch\ out - wakeup \tag{3.2}$$

The yield event was counted for each thread and plotted in a bar chart diagram. It was called by the test program every time the program calculated a missed deadline of a task. The migrate event was also counted. The difference is that they were separated into two types; migrates before runtime and migrates during runtime. Migrates before runtime counted all migrations between tasks' wakeup event and their first switch in event, while migrates during runtime counted all migrations between the first switch in event and next wakeup event.

## 3.4 Evaluation of Ericsson's Application

The application provided by Ericsson was fed by a test driver, which simulated the *UE* traffic towards the application. Which cores the application should run on was specified when the application was started. The priority of the application and its pthreads (created by the application) were also given at this point. A benchmark program named lmbench was running on the same cores as the application in order to stress the processor.

The kernel event traced for Ericsson's application was the same as for the test program. Babeltrace and the python program were used as well. The python program works slightly different when processing data from another program than the test program. When it read through the file generated by Babeltrace, it searched for more than one specific key word since Ericsson's application created pthreads with different names. Another difference is the yield event – it was not required since it was used only for the purposes of the synthetic test program. If a pthread scheduled with SCHED_DEADLINE failed its admission control, this information was provided by the test program, not by Ericsson's application. It could therefore only be evaluated for the test program.

In the test program, it was known which threads had the same thread parameters, since the test cases were known. These threads were given equal similarity indexes. When Ericsson's application was evaluated, the similarity index was created from the names of the pthreads. All pthreads with the same name have the same similarity index. This index was used in the creation of the result figures.

## 3.5 Creating Result Figures

To view and analyse the metrics produced from the event list created with Babeltrace, they were presented in different figures. Bar charts were used when only one value per thread should be compared, for example the thread utilization. When more than one value per thread

should be analysed, histograms where used to visualize the data since it is a good way to show large data sets.

### 3.5.1 Bar chart

Bar charts were generated in the python program to visualize the data when only one value per thread was given. If the number of traced threads were greater than five, threads with the same similarity index were bunched together. This was done in order to make the figures readable.

### 3.5.2 Histogram

Histograms were generated in the python program when more than one value per thread was given. They were generated by grouping data into so called bins. The maximum value of the data set, the median value of the data set and the number of desired bins were needed to create the bin size. Equations 3.3-3.6 and 3.7-3.9 show how the bin sizes were calculated. The first three equations were used for odd number of bins and the other three were used for even number of bins. The bin sizes were smaller closer to the median value. The largest bin was forced to the maximum value in the data set in order to avoid missing data due to rounding errors in the recursive algorithm.

$$\begin{cases} bin[0] = 0 \\ bin[n] = bin[n-1] + h_1 k^{\frac{N+1}{2}-n}, & 0 < n \leq \dfrac{N+1}{2} \\ bin[n] = bin[n-1] + h_2 k^{n-\frac{N+1}{2}}, & \dfrac{N+1}{2} < n \leq N-1 \\ bin[N] = \max \end{cases} \tag{3.3}$$

$$\alpha = \frac{k^{\frac{N+1}{2}} - k}{k-1} \tag{3.4}$$

$$h_1 = \frac{median}{\alpha + \frac{1}{2}} \tag{3.5}$$

$$h_2 = \frac{max - h_1(\alpha+1)}{\alpha} \tag{3.6}$$

$$\begin{cases} bin[0] = 0 \\ bin[n] = bin[n-1] + h_1 k^{\frac{N}{2}-n}, & 0 < n \leq \dfrac{N}{2} \\ bin[n] = bin[n-1] + h_2 k^{n-\frac{N}{2}-1}, & \dfrac{N}{2} < n \leq N-1 \\ bin[N] = \max \end{cases} \tag{3.7}$$

$$h_1 = \frac{median(k-1)}{k^{\frac{N}{2}} - 1} \tag{3.8}$$

$$h_2 = \frac{(max - median)(k-1)}{k^{\frac{N}{2}} - 1} \tag{3.9}$$

The value $k$ in the equations is a scaling factor. A bin one step further away from the median will be $k$ times larger. The value of $k$ in the histograms was set to 1.5.

If many threads should be visualized in the same figure, the threads were bunched together in the same way as for the bar chart figures.

# Chapter 4

# Results

The purpose of this chapter is to present the results of this project, both for the test program and for the application provided by Ericsson.

## 4.1 The Test Program

The results from the test program have been collected from the testing platform called virtual machine. This means that the test cases have been performed on two hardware threads. Below follow three subchapters presenting the results from the three different test cases described in chapter 3.2.2 – *Test cases*.

### 4.1.1 Test case A

Before studying the figures below, it is necessary to know how many threads have failed to schedule due to admission control and of which type they are. This information can be found in Table 4.1. SCHED_FIFO and SCHED_RR are not included in the table, since threads scheduled with those policies cannot fail. The reason for this is that they do not perform an admission control.

The response time is illustrated in Figure 4.1-4.4. The computation time is shown in Figure 4.5-4.8 and the thread utilization for each policy is shown in Figure 4.9-4.12. The number of missed deadlines computed by the test program can be found in Figure 4.13-4.16 and the number of task migrations is presented in Figure 4.17-4.20. Note that there are six threads of each type in the figures.

| **Test** | **Thread 1 [pc]** | **Thread 2 [pc]** | **Thread 3 [pc]** |
|---|---|---|---|
| SCHED_DEADLINE median | 0 | 0 | 0 |
| SCHED_DEADLINE median plus | 1 | 0 | 0 |

*Table 4.1: Number of threads failing to schedule in test case A*

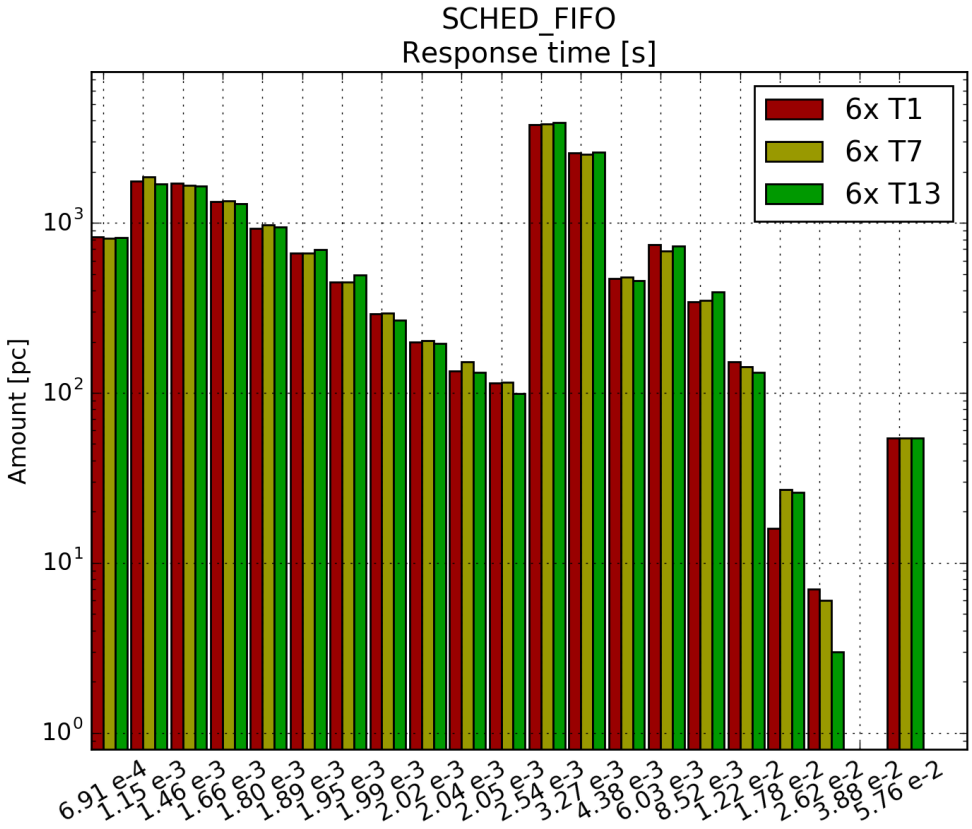


*Figure 4.1: Test case A, response time, SCHED_FIFO*

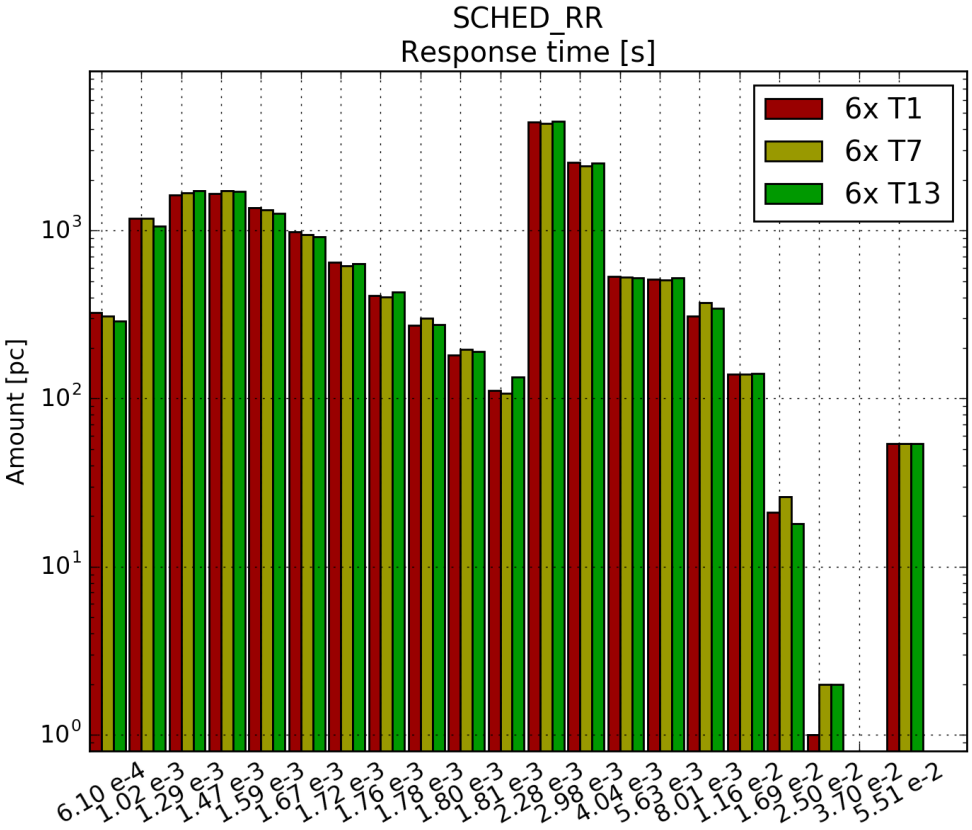


*Figure 4.2: Test case A, response time, SCHED_RR*

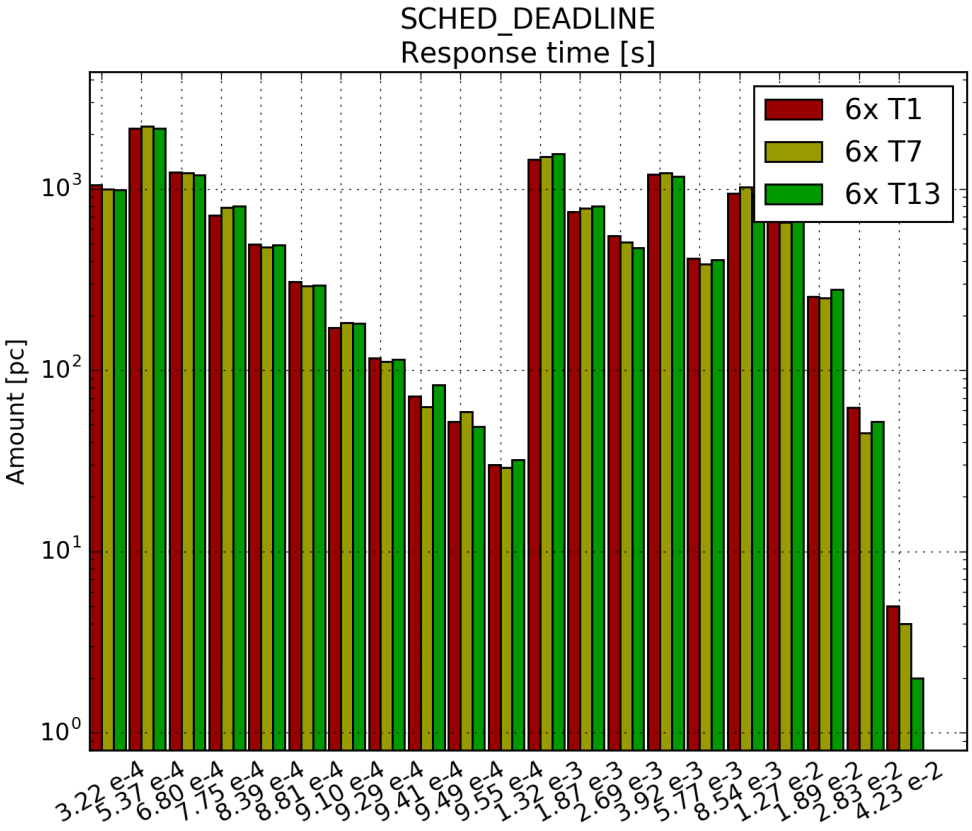


*Figure 4.3: Test case A, response time, SCHED_DEADLINE median*

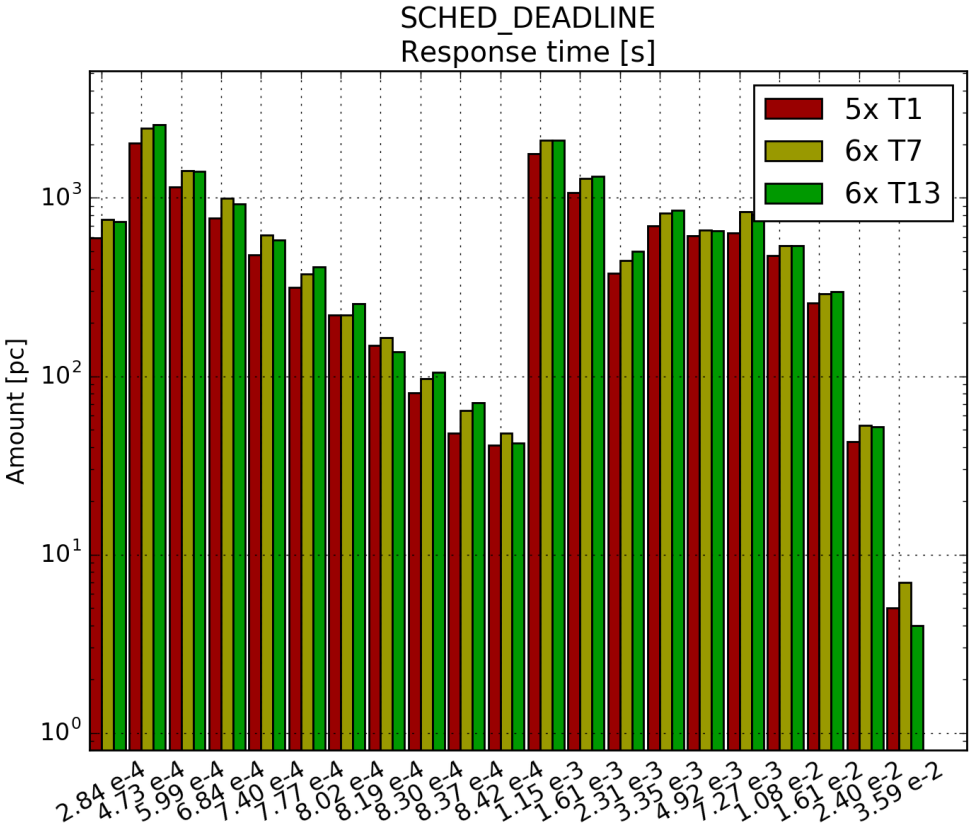


*Figure 4.4: Test case A, response time, SCHED_DEADLINE median plus*

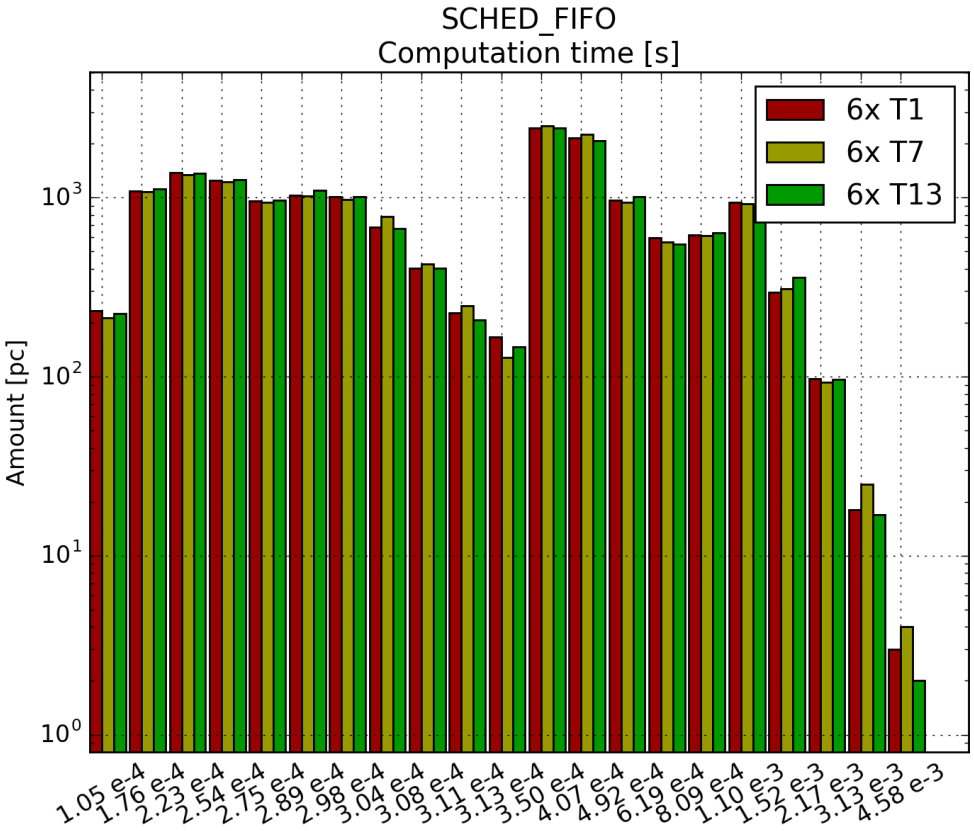


*Figure 4.5: Test case A, computation time, SCHED_FIFO*

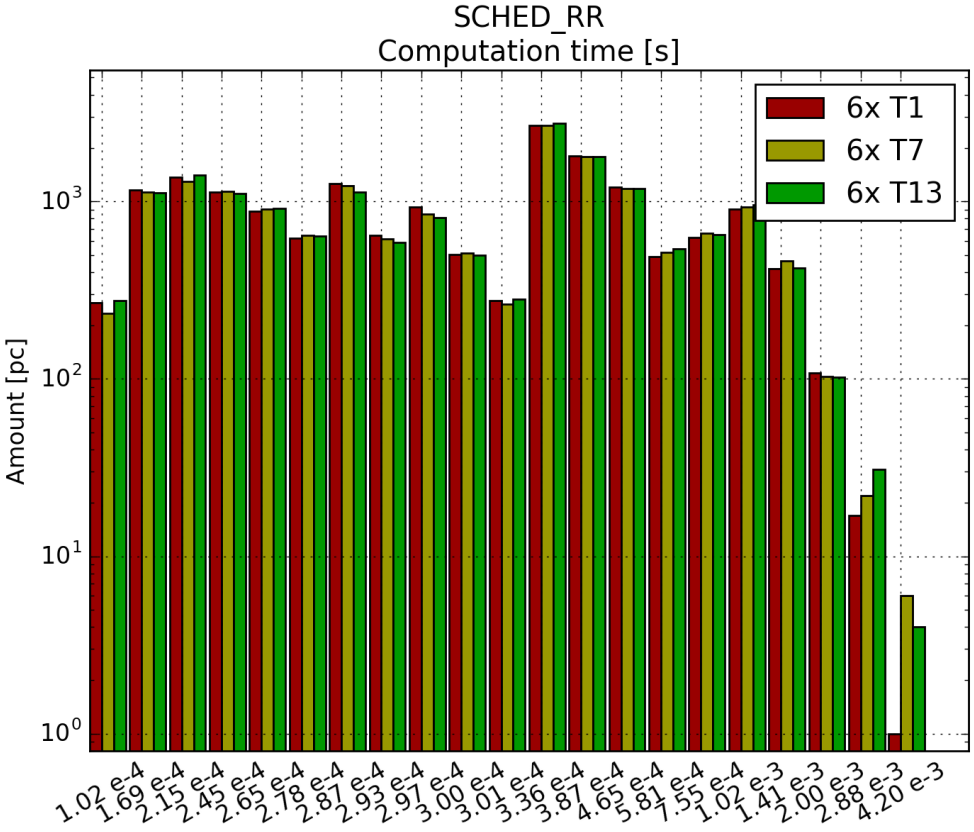


*Figure 4.6: Test case A, computation time, SCHED_RR*

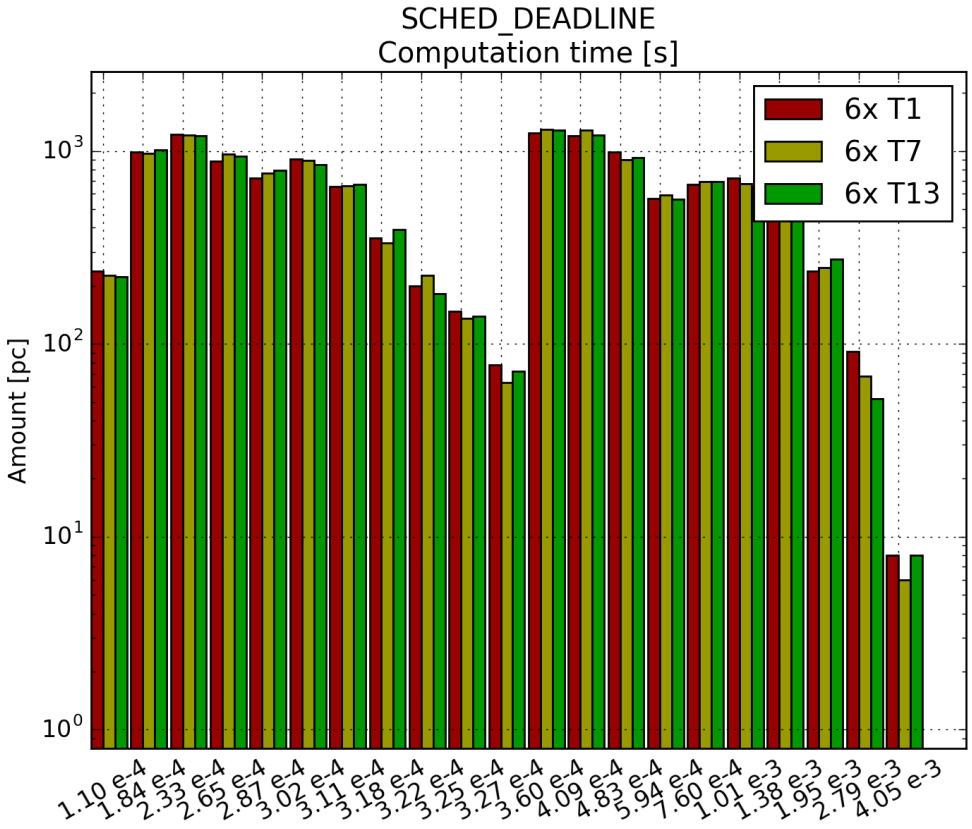


*Figure 4.7: Test case A, computation time, SCHED_DEADLINE median*

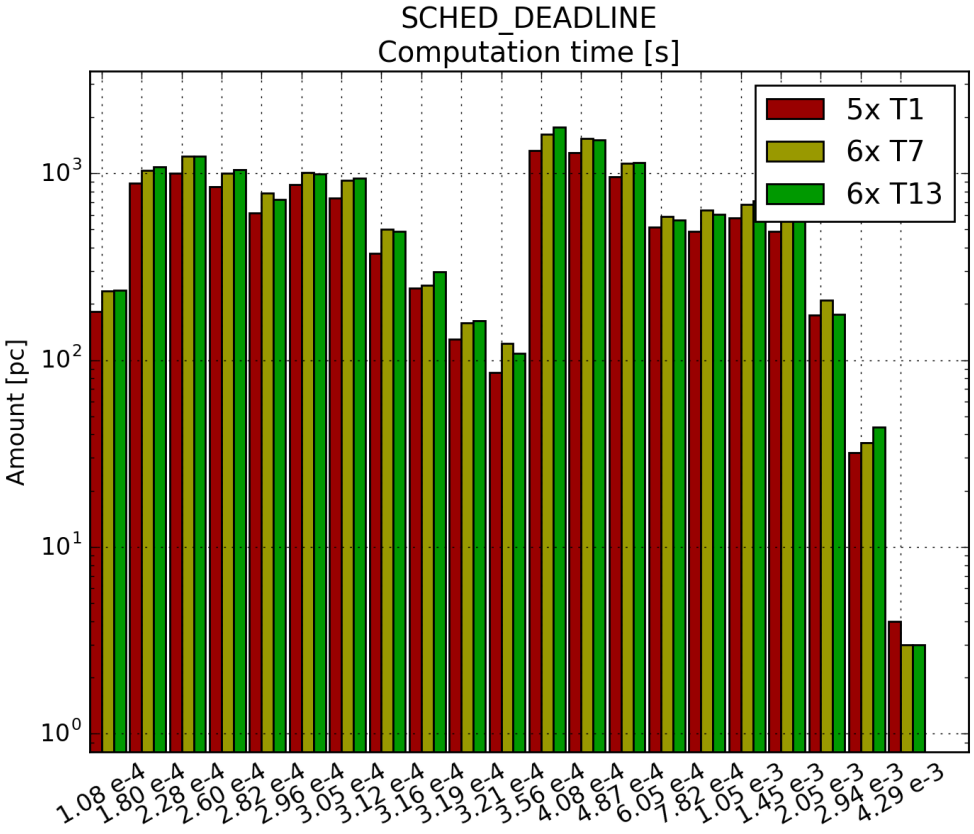


*Figure 4.8: Test case A, computation time, SCHED_DEADLINE median plus*

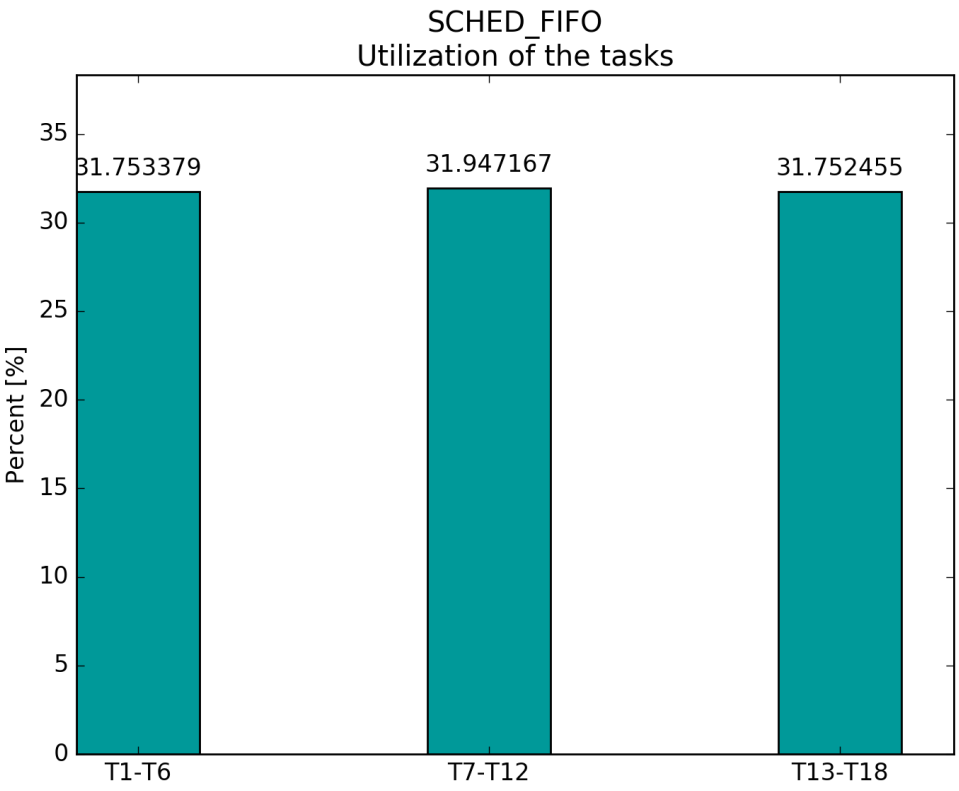


*Figure 4.9: Test case A, utilization, SCHED_FIFO*

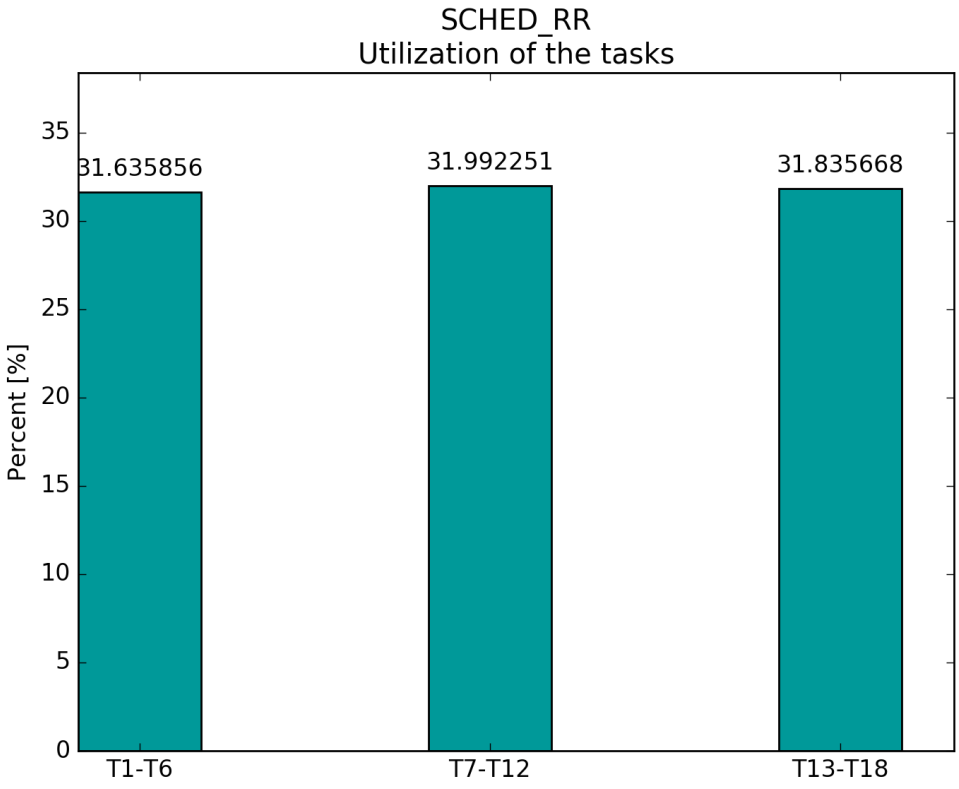


*Figure 4.10: Test case A, utilization, SCHED_RR*

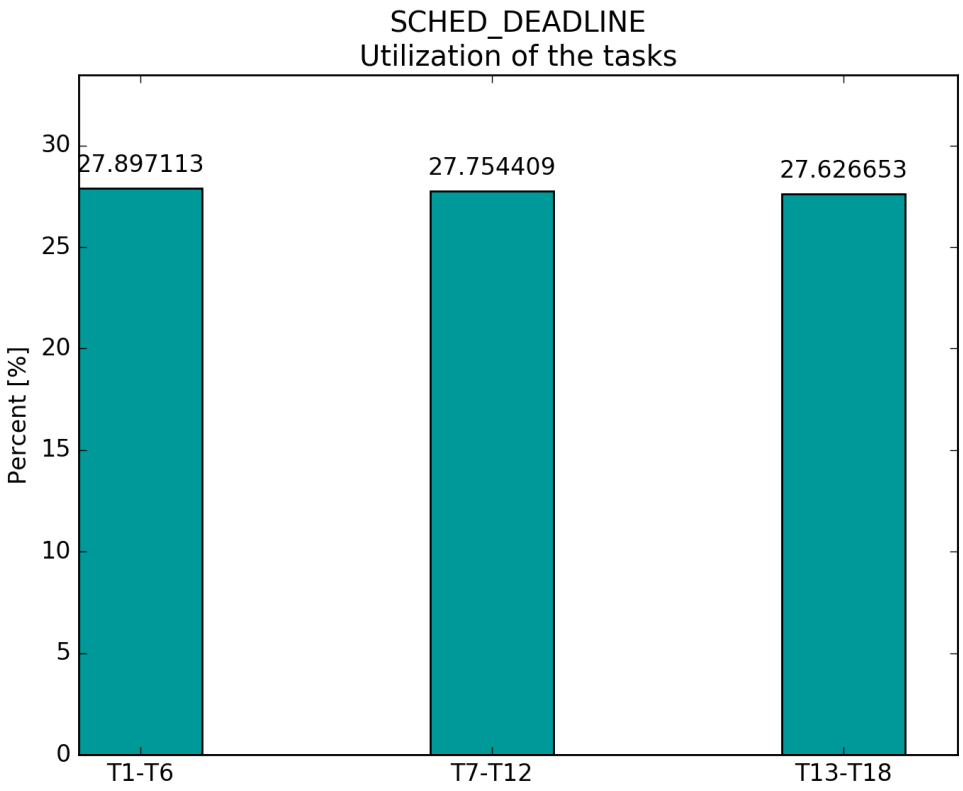


*Figure 4.11: Test case A, utilization, SCHED_DEADLINE median*

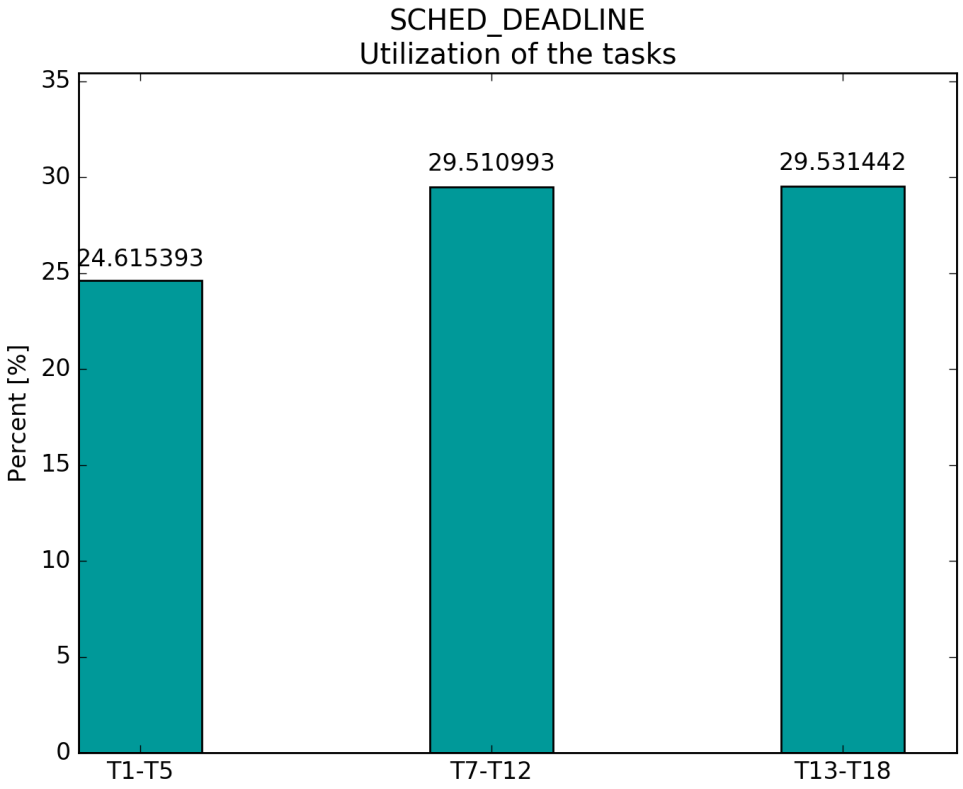


*Figure 4.12: Test case A, utilization, SCHED_DEADLINE median plus*

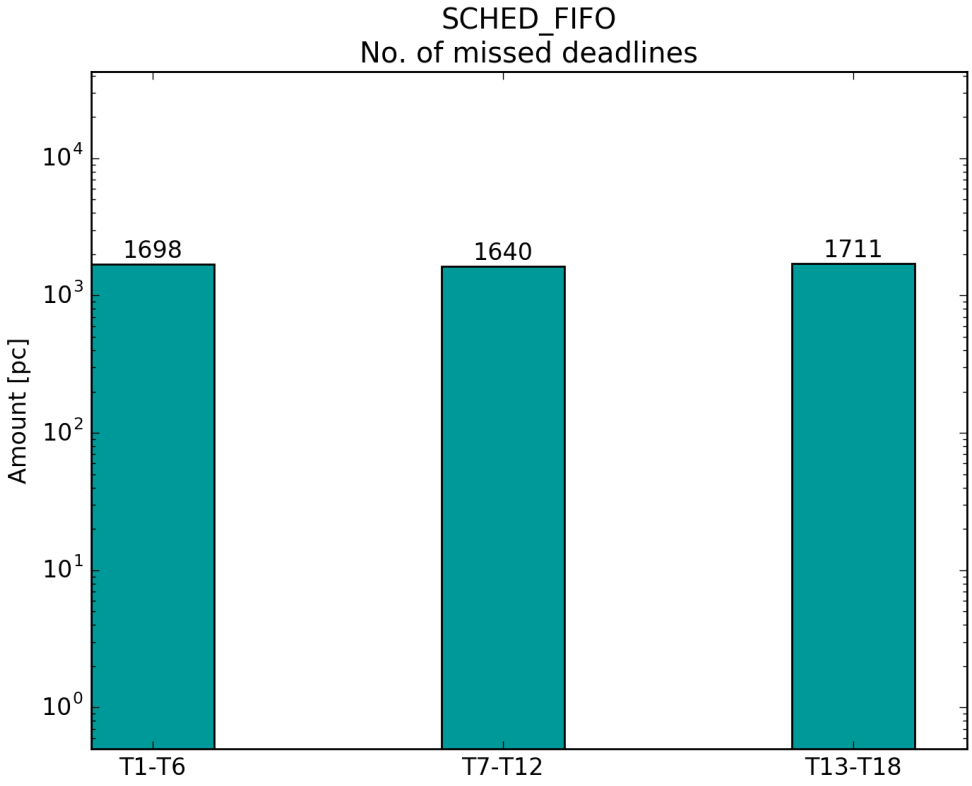


*Figure 4.13: Test case A, missed deadlines, SCHED_FIFO*

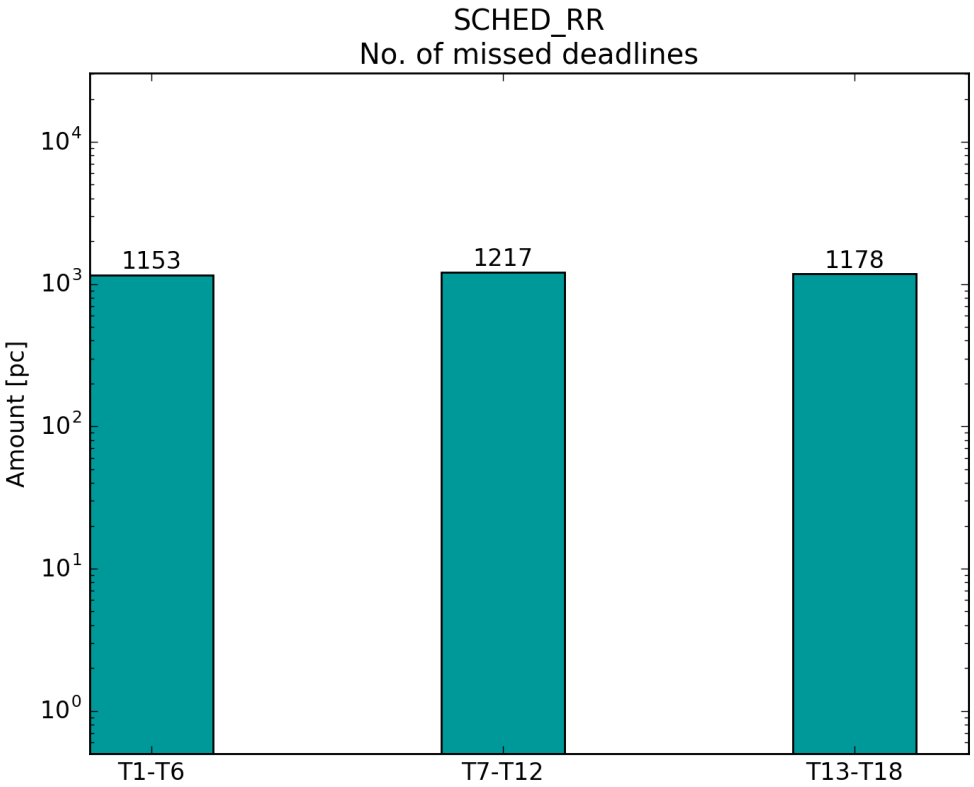


*Figure 4.14: Test case A, missed deadlines, SCHED_RR*

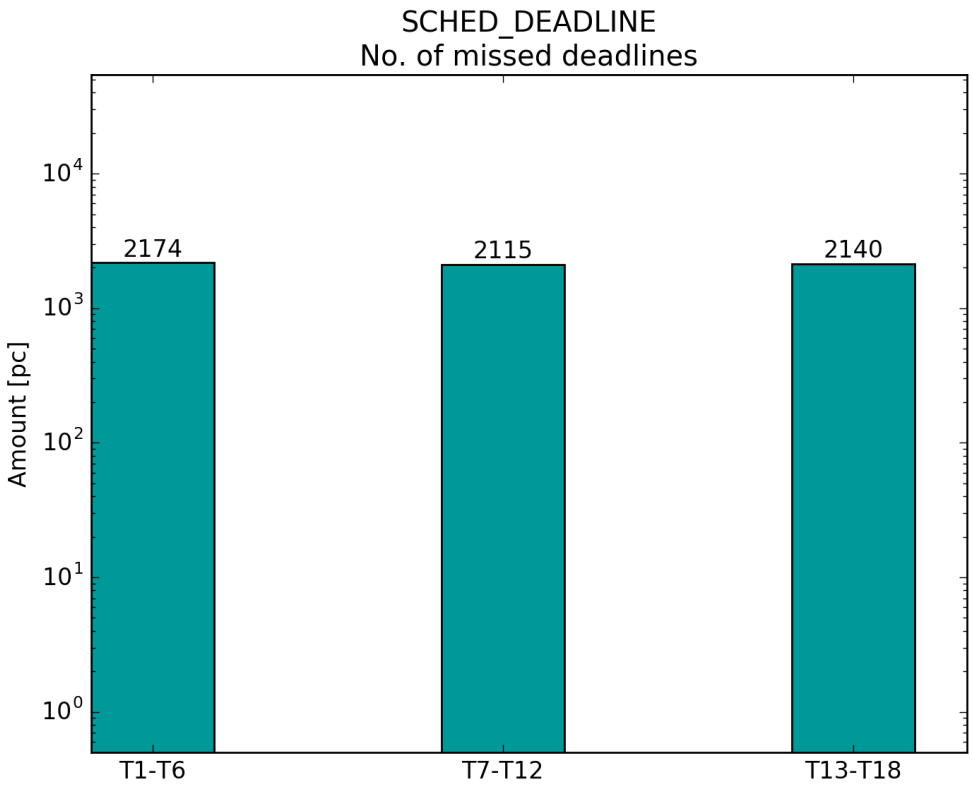


*Figure 4.15: Test case A, missed deadlines, SCHED_DEADLINE median*

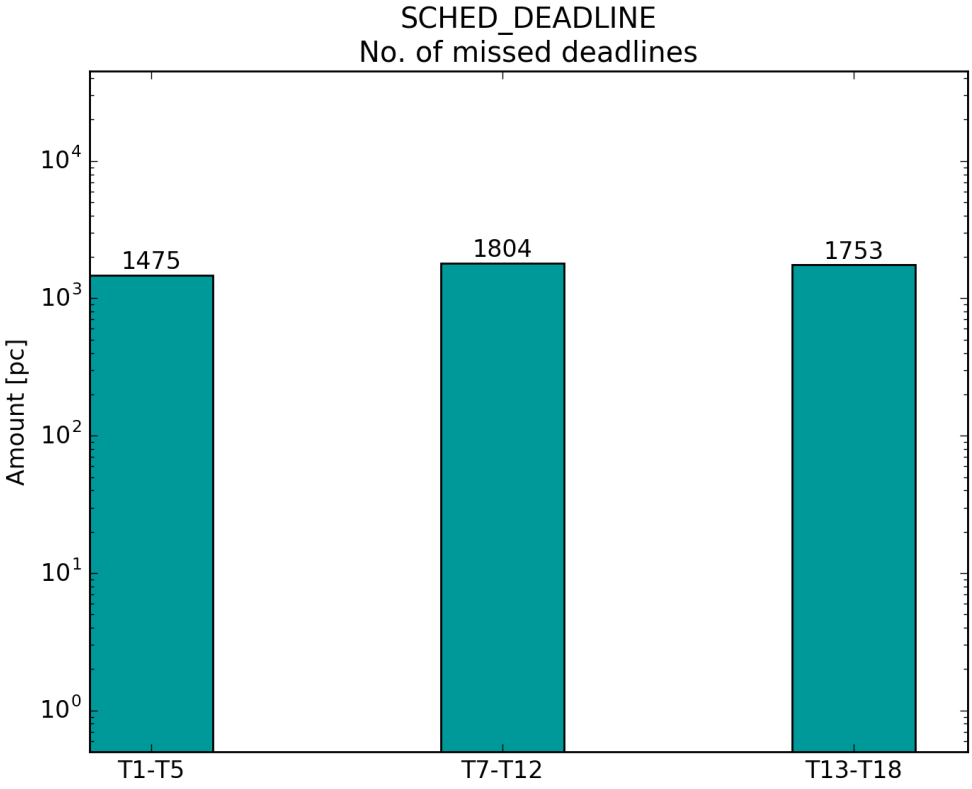


*Figure 4.16: Test case A, missed deadlines, SCHED_DEADLINE median plus*

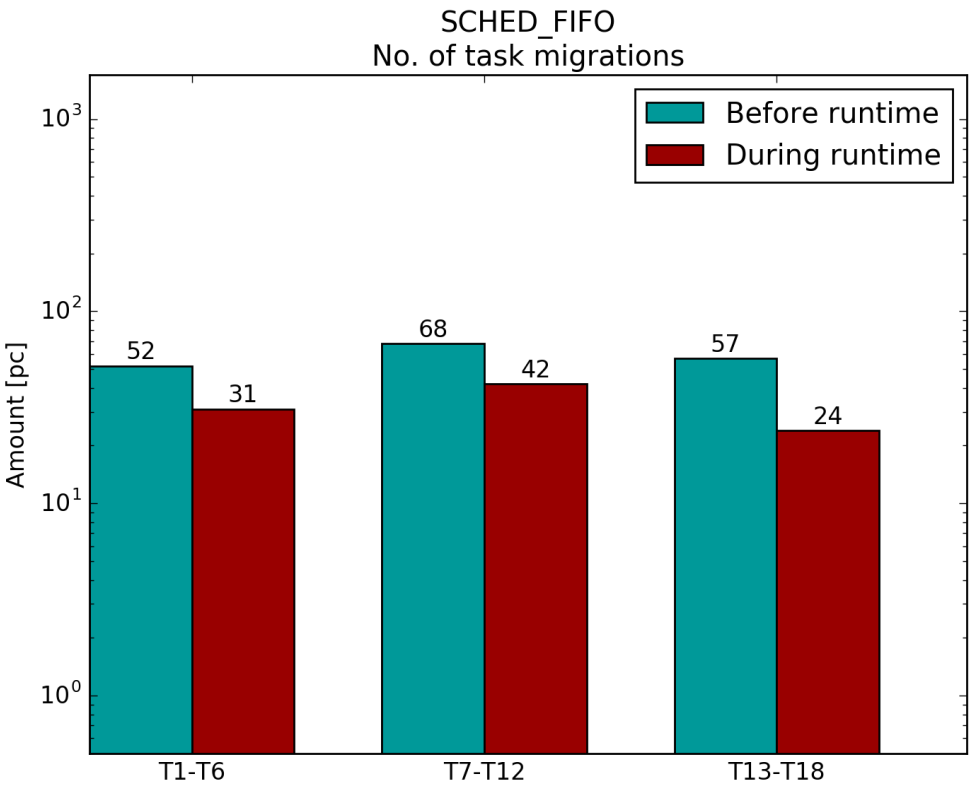


*Figure 4.17: Test case A, migrations, SCHED_FIFO*

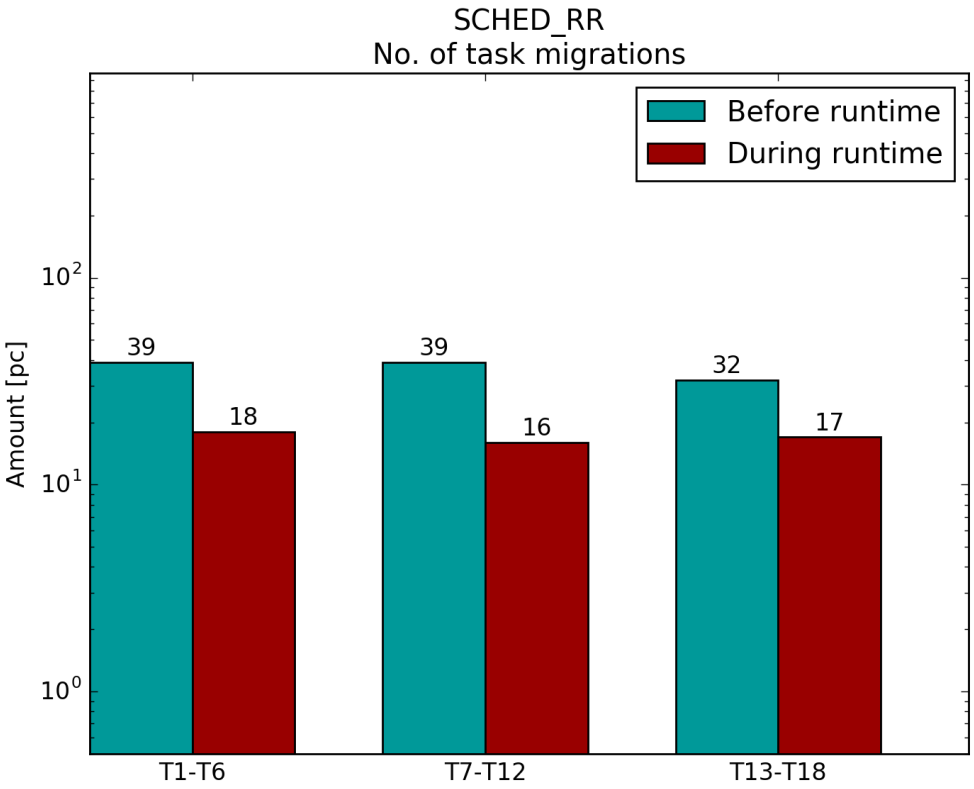


*Figure 4.18: Test case A, migrations, SCHED_RR*

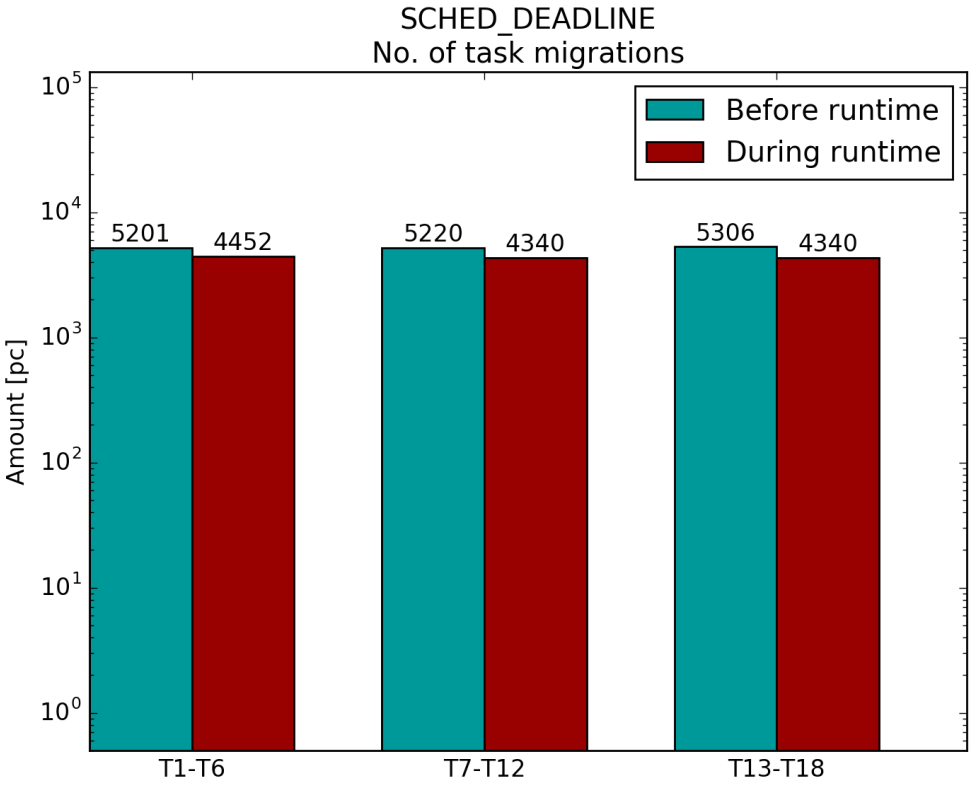


*Figure 4.19: Test case A, migrations, SCHED_DEADLINE median*

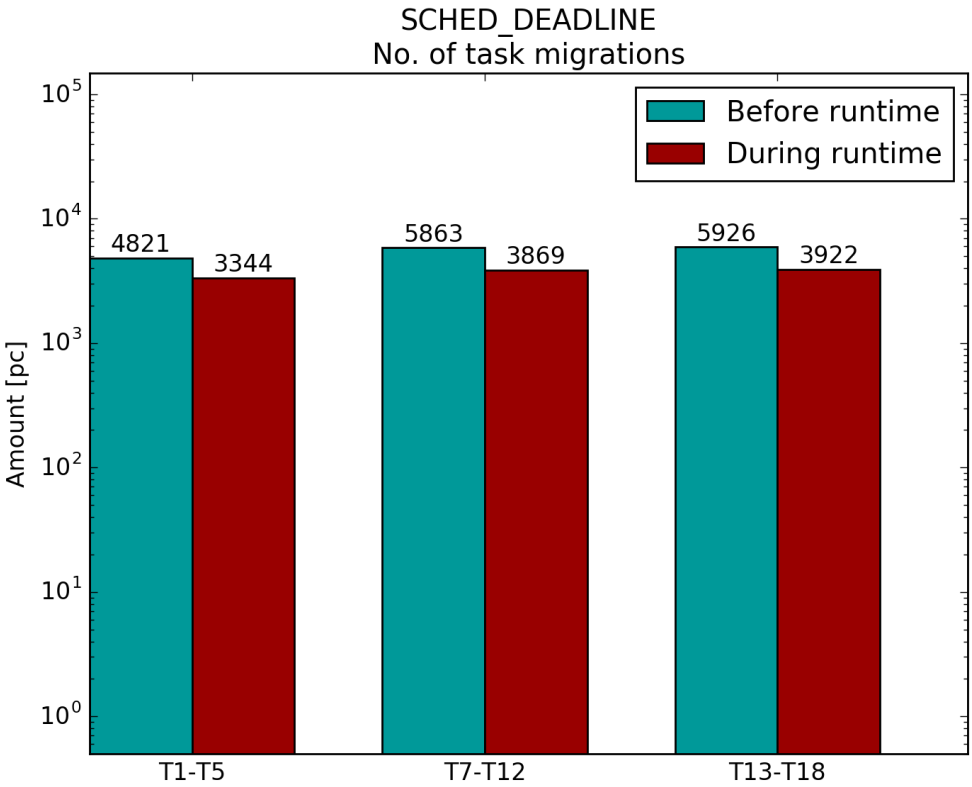


*Figure 4.20: Test case A, migrations, SCHED_DEADLINE median plus*

### 4.1.2 Test case B

The threads that failed to schedule in this test case are presented in Table 4.2. Figure 4.21-4.25 shows the response time results for the different test runs and Figure 4.26-4.30 shows the computation time. Figure 4.31-4.35 shows the utilization of the tasks, Figure 4.36-4.40 the missed deadlines and Figure 4.41-4.45 the number of migrations.

| Test | Thread 1 [pc] | Thread 2 [pc] | Thread 3 [pc] |
|---|---|---|---|
| SCHED_DEADLINE median | 0 | 0 | 0 |
| SCHED_DEADLINE median plus | 0 | 0 | 4 |

*Table 4.2: Number of threads failing to schedule in test case B*

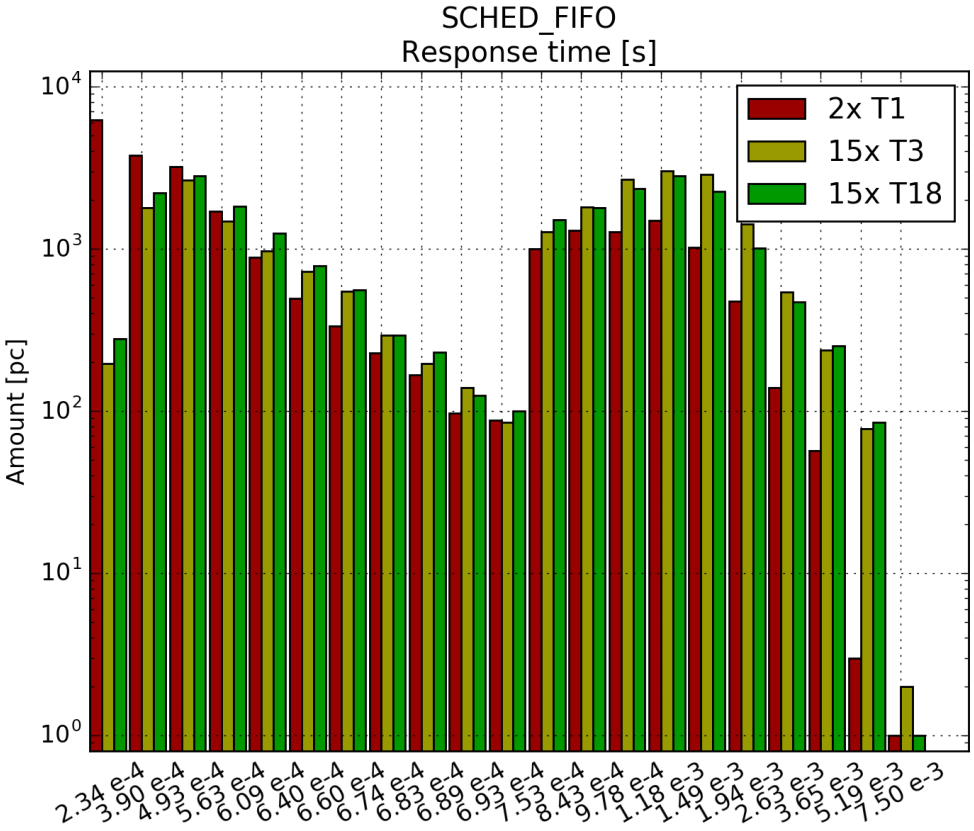


*Figure 4.21: Test case B, response time, SCHED_FIFO*

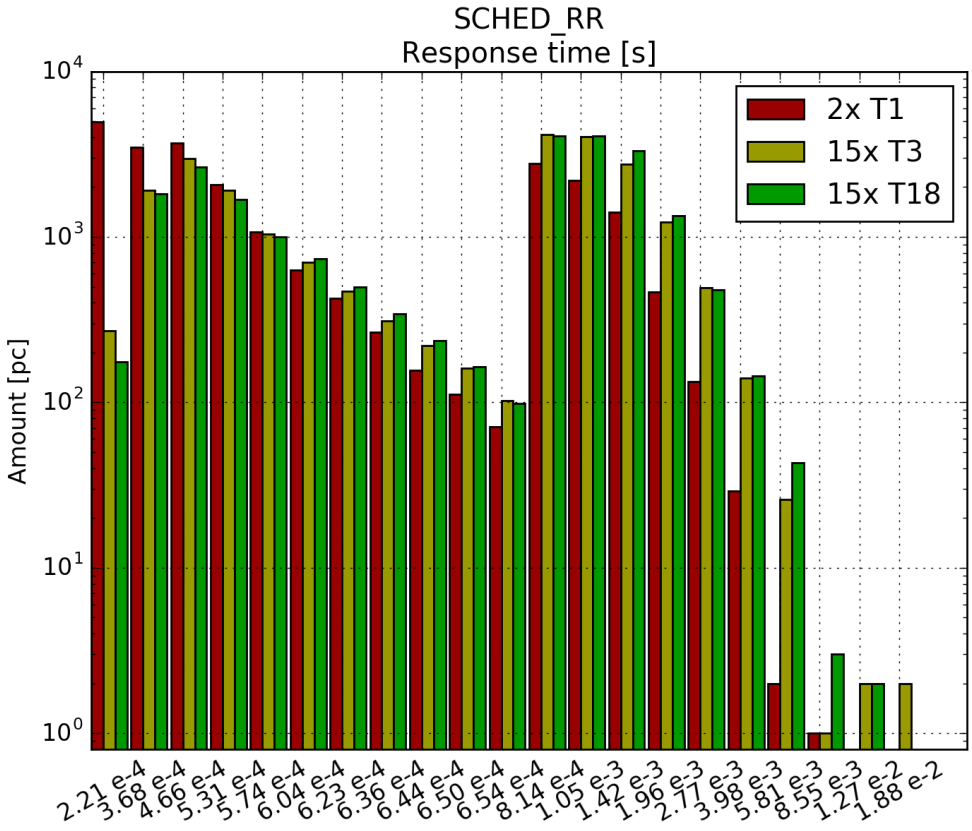


*Figure 4.22: Test case B, response time, SCHED_RR*

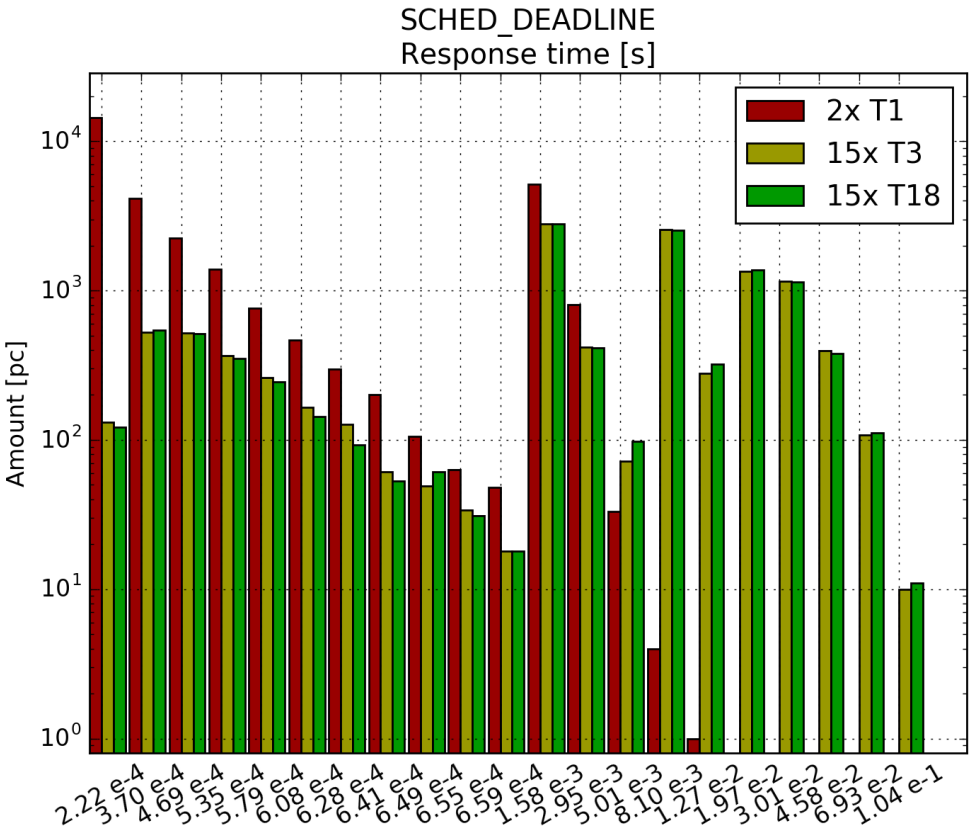


*Figure 4.23: Test case B, response time, SCHED_DEADLINE median*

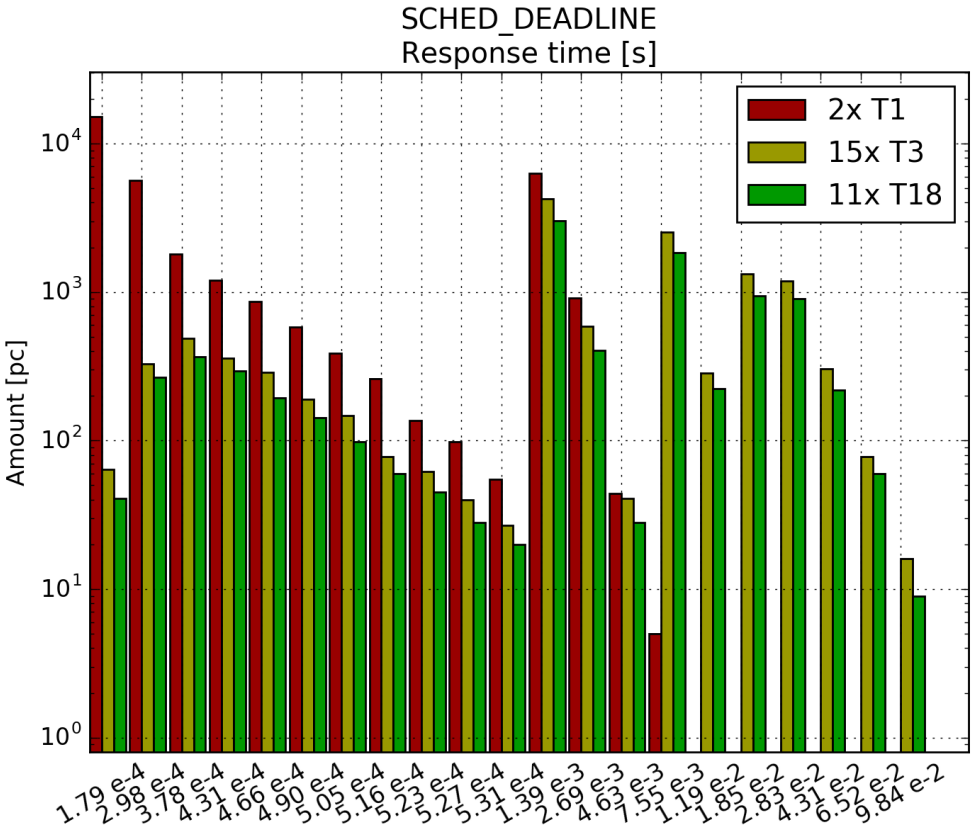


*Figure 4.24: Test case B, response time, SCHED_DEADLINE median plus*

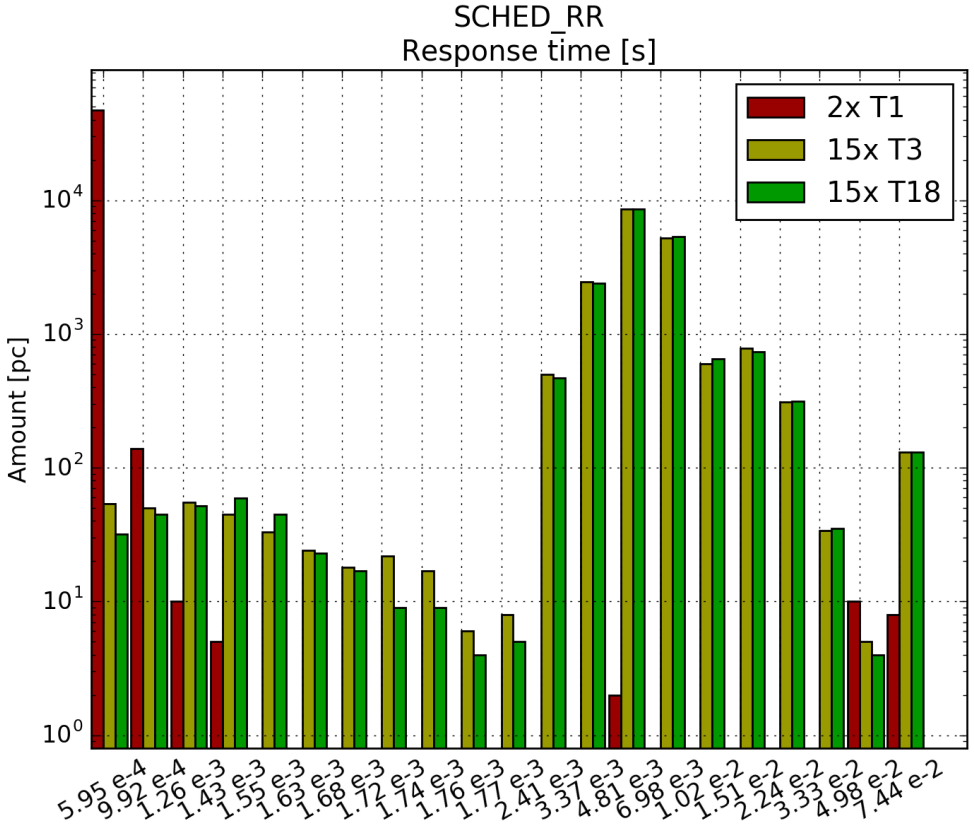


*Figure 4.25: Test case B, response time, RMS implemented with SCHED_RR*

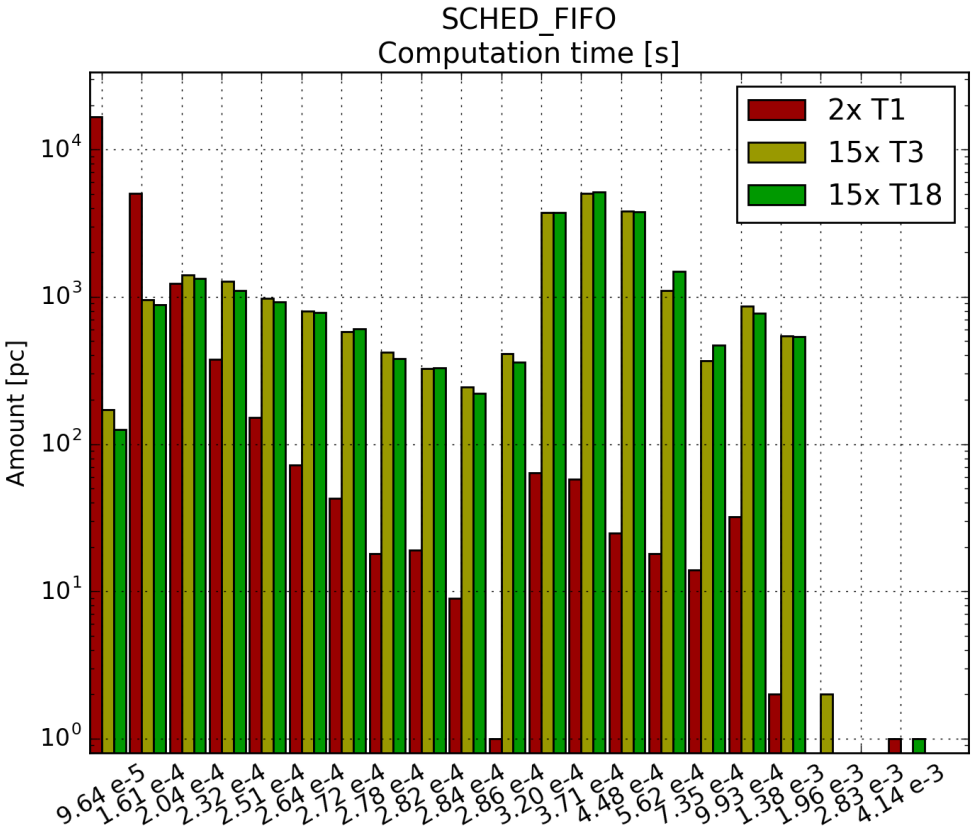


*Figure 4.26: Test case B, computation time, SCHED_FIFO*

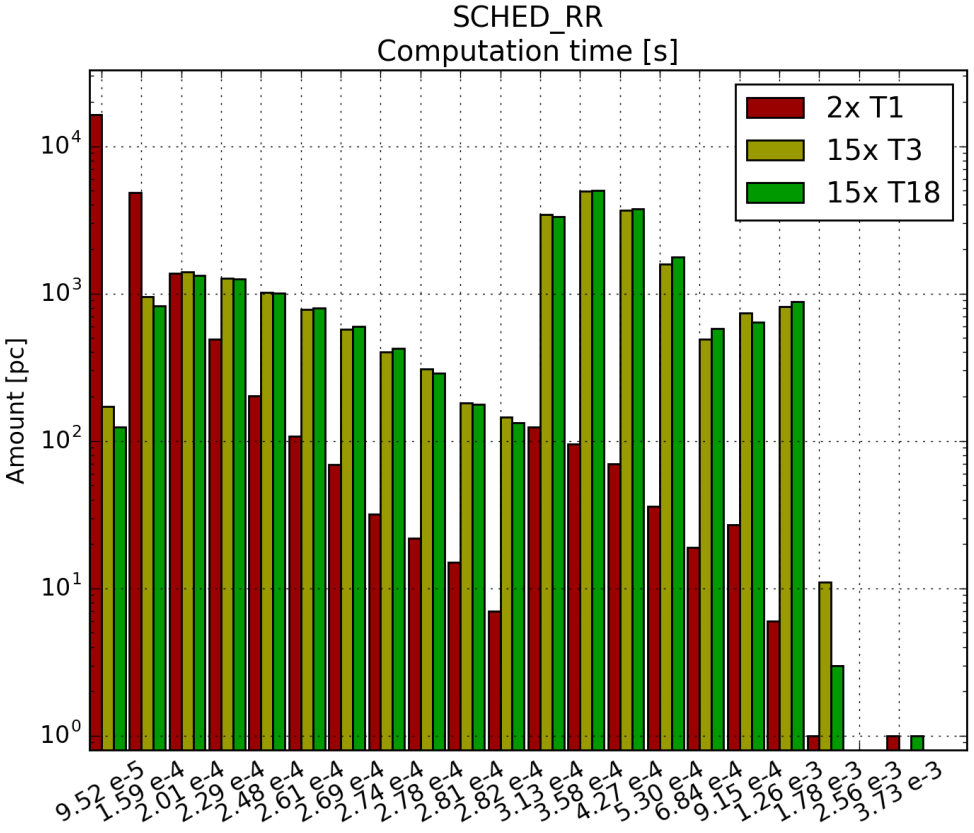


*Figure 4.27: Test case B, computation time, SCHED_RR*

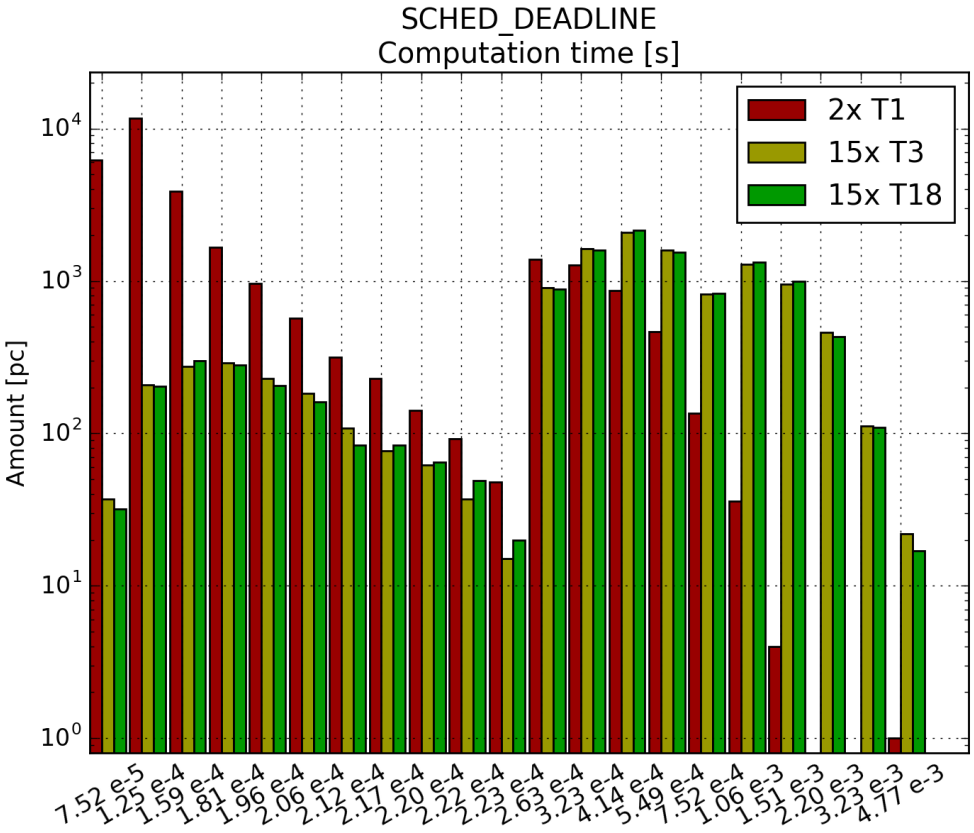


*Figure 4.28: Test case B, computation time, SCHED_DEADLINE median*

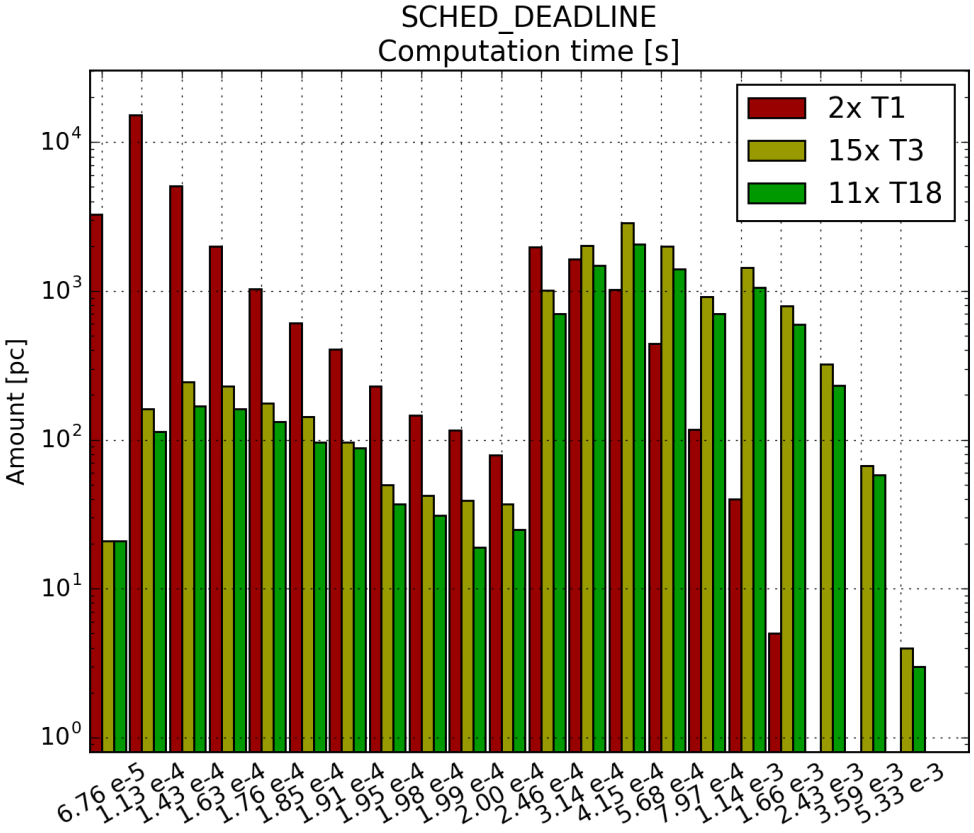


*Figure 4.29: Test case B, computation time, SCHED_DEADLINE median plus*

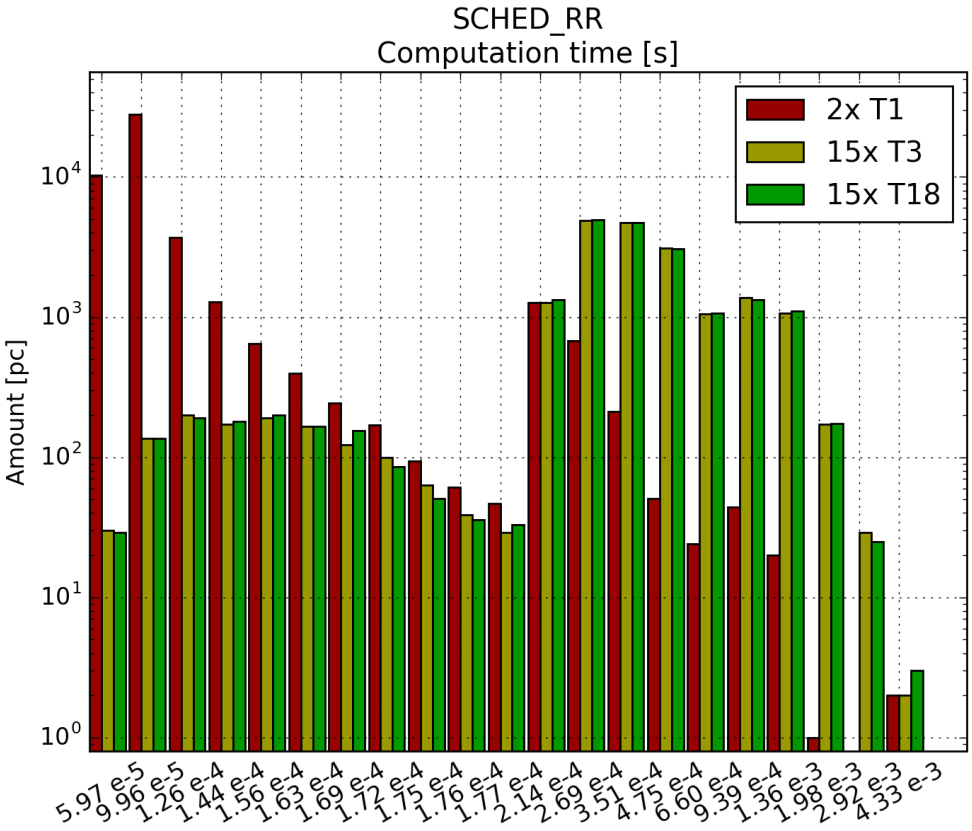


*Figure 4.30: Test case B, computation time, RMS implemented with SCHED_RR*

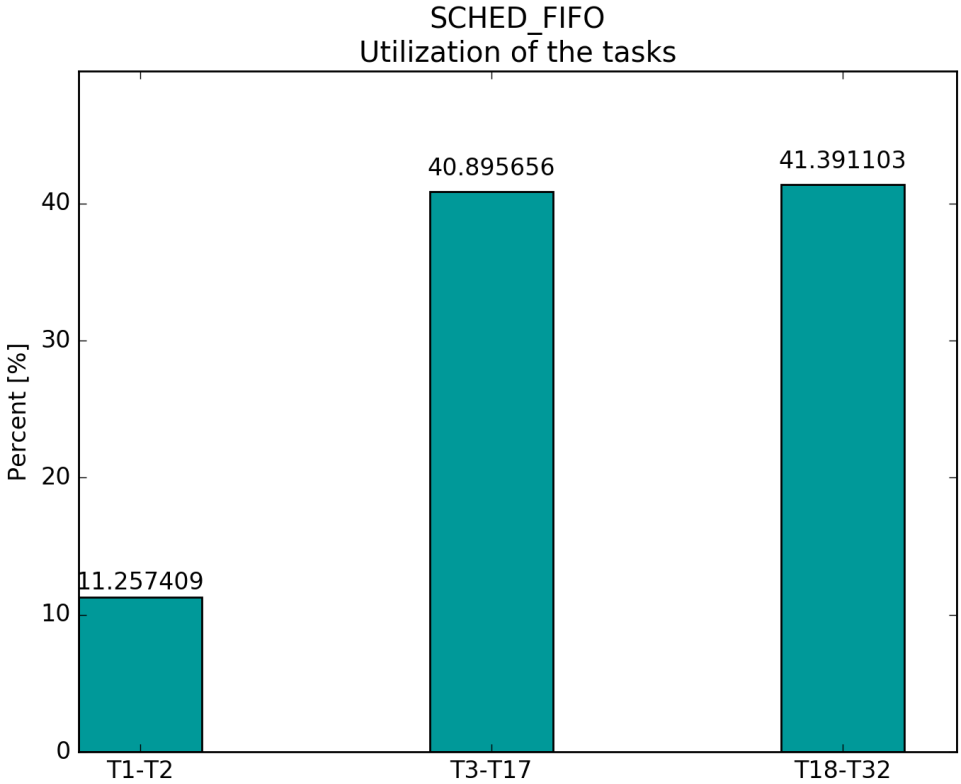


*Figure 4.31: Test case B, utilization, SCHED_FIFO*

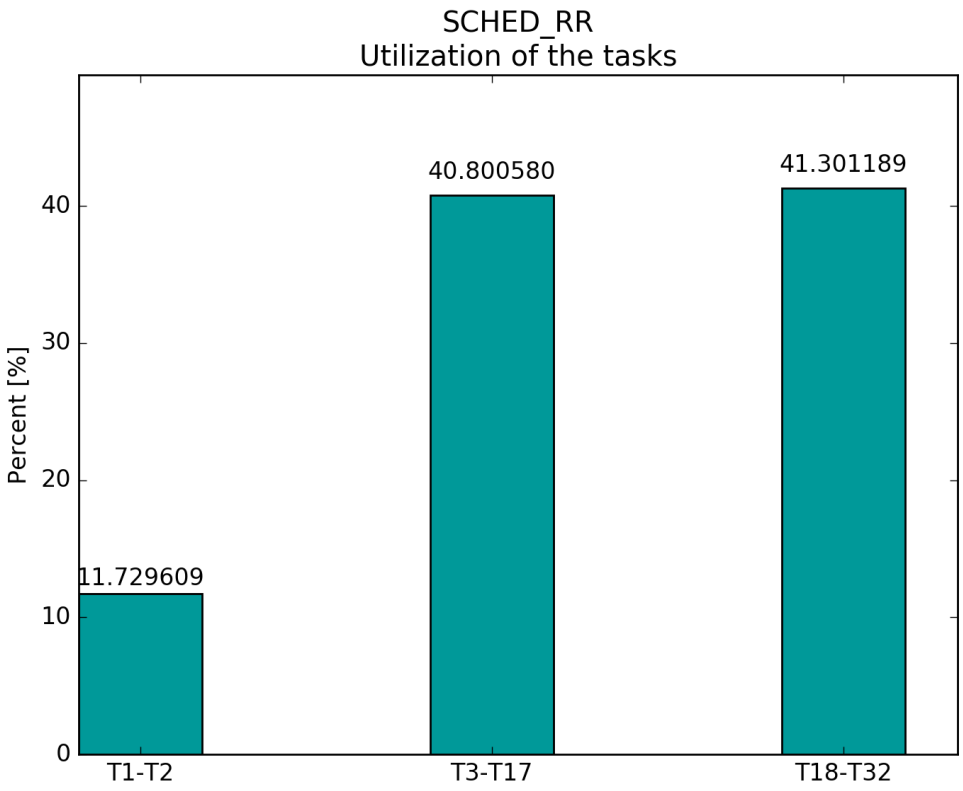


*Figure 4.32: Test case B, utilization, SCHED_RR*

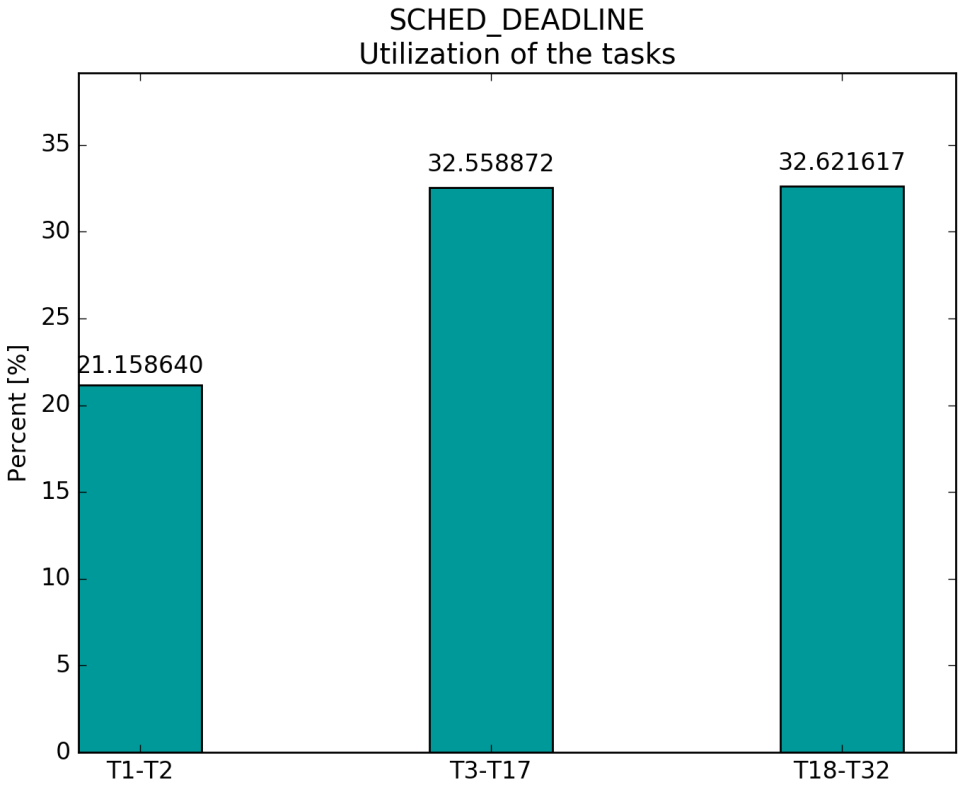


*Figure 4.33: Test case B, utilization, SCHED_DEADLINE median*

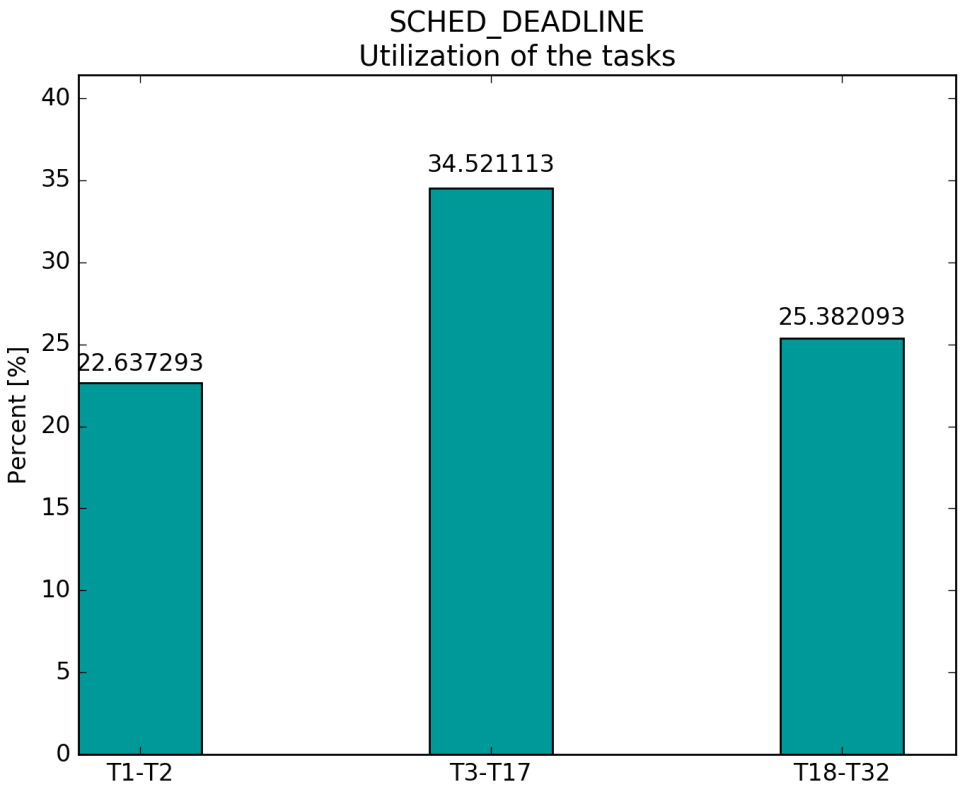


*Figure 4.34: Test case B, utilization, SCHED_DEADLINE median plus*

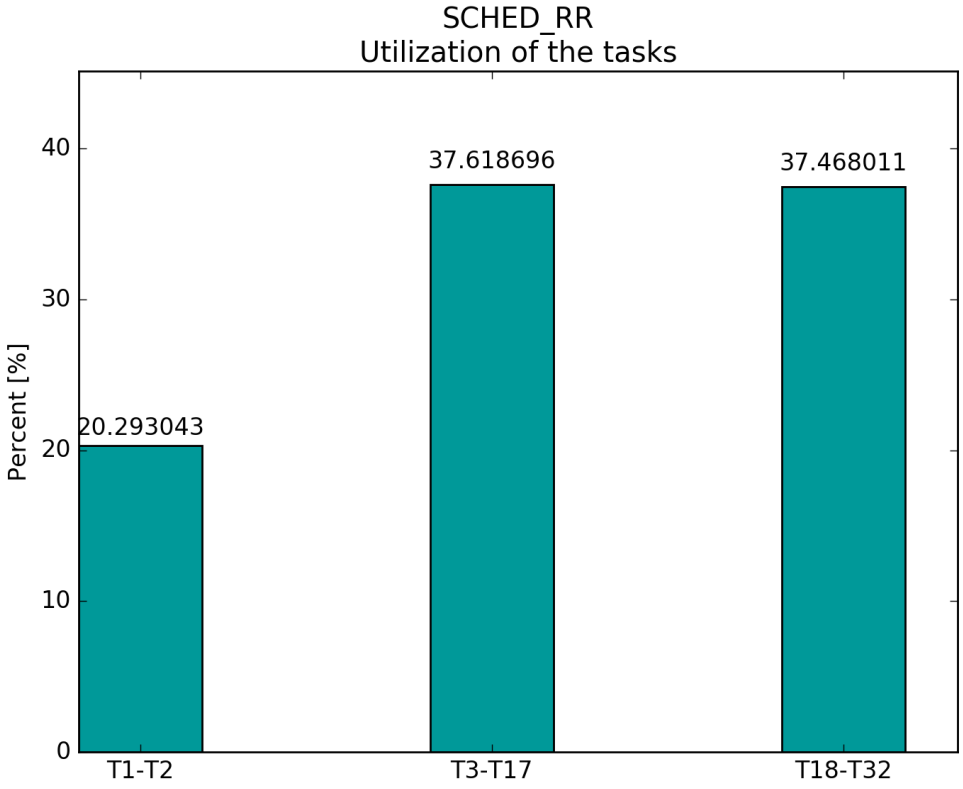


*Figure 4.35: Test case B, utilization, RMS implemented with SCHED_RR*

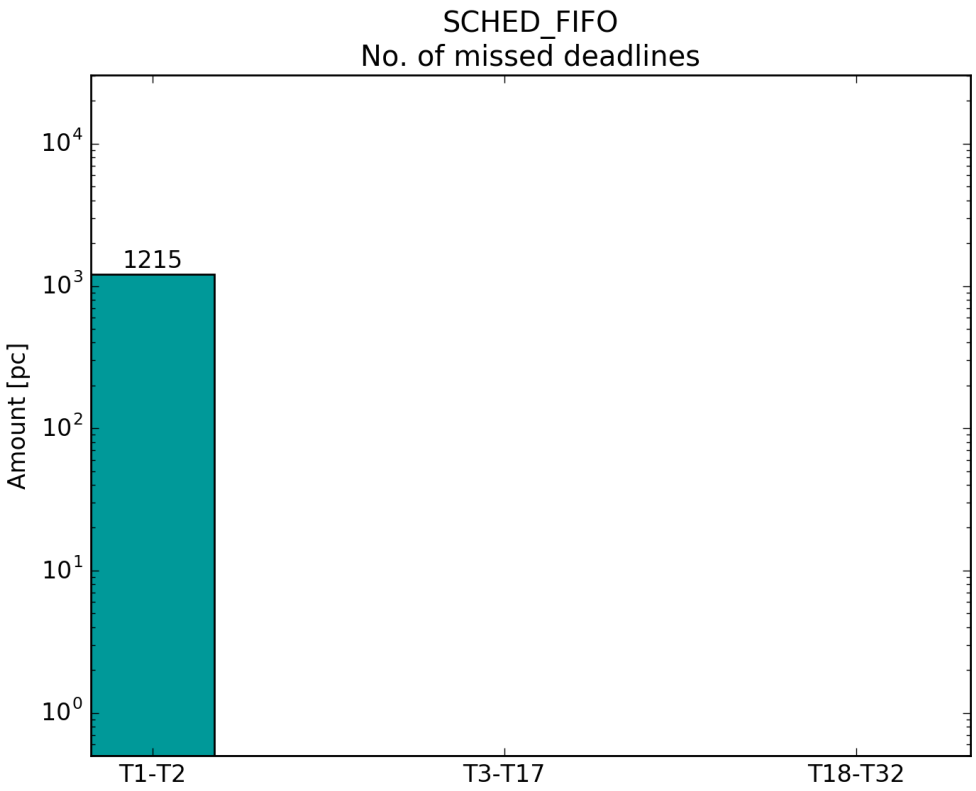


*Figure 4.36: Test case B, missed deadlines, SCHED_FIFO*

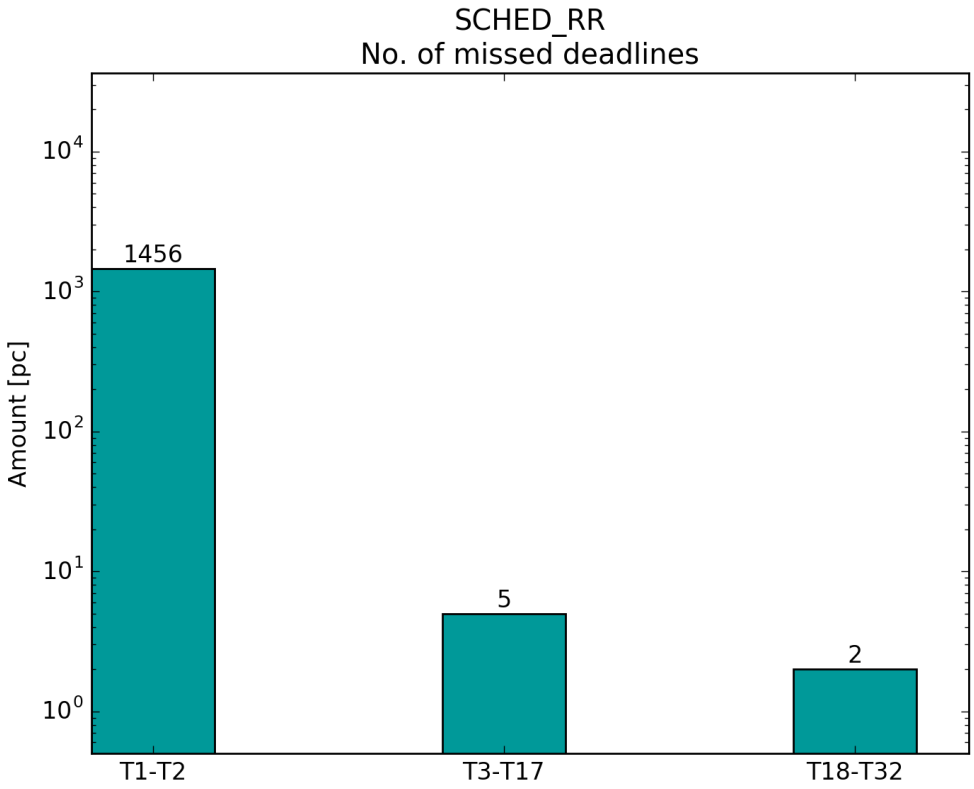


*Figure 4.37: Test case B, missed deadlines, SCHED_RR*

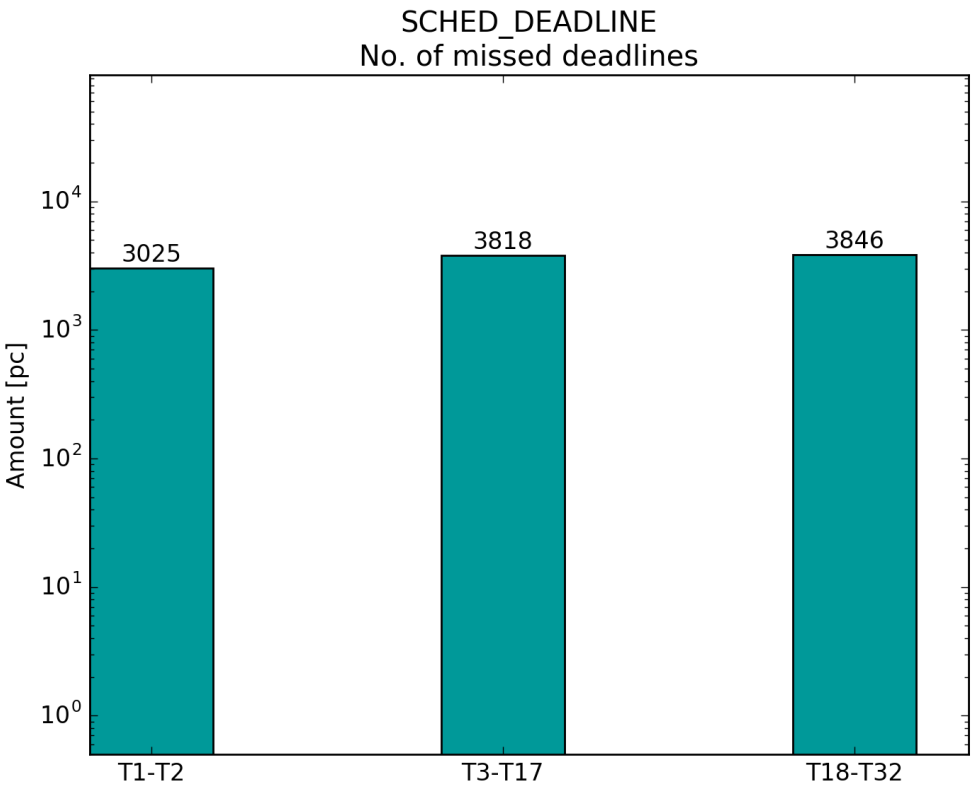


*Figure 4.38: Test case B, missed deadlines, SCHED_DEADLINE median*

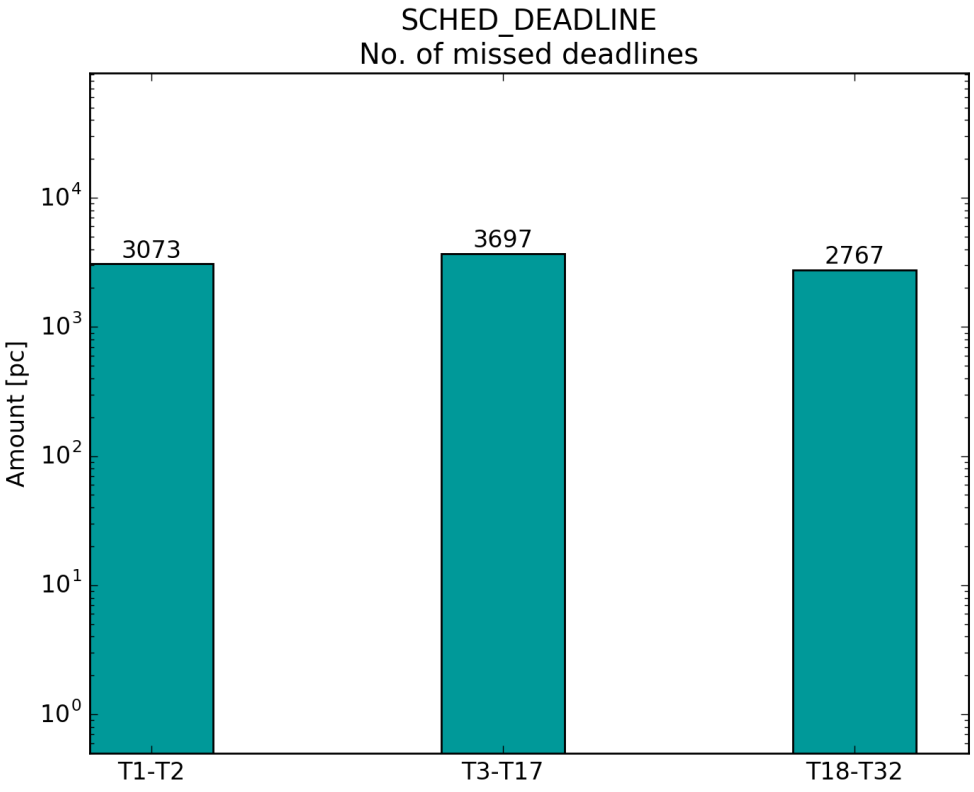


*Figure 4.39: Test case B, missed deadlines, SCHED_DEADLINE median plus*

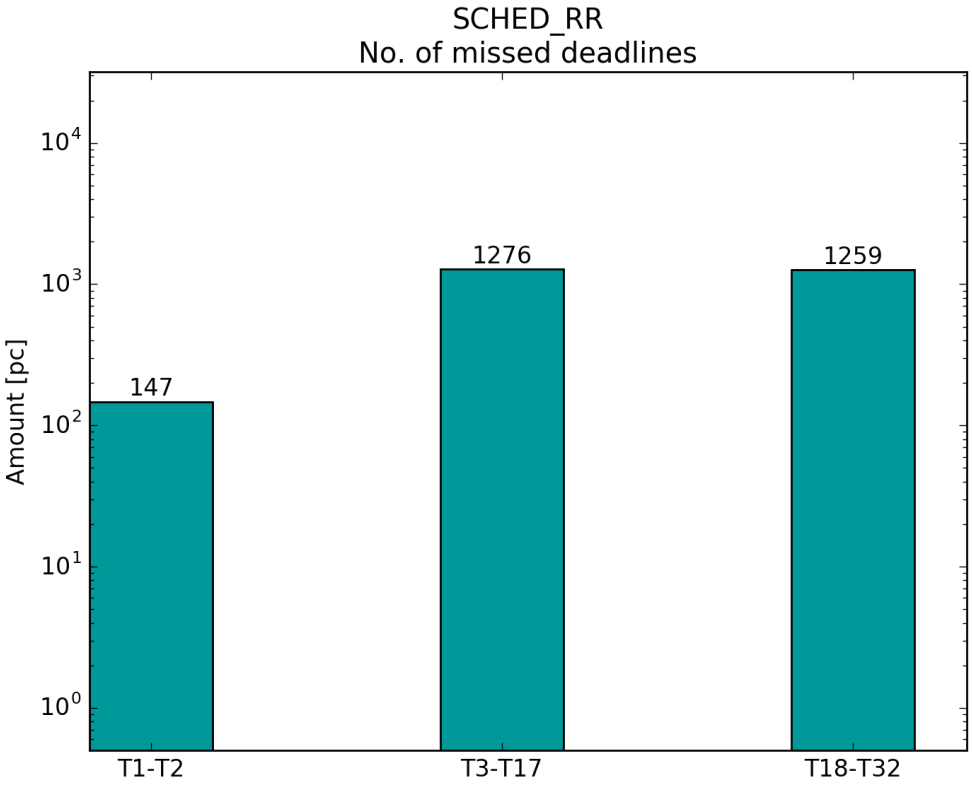


*Figure 4.40: Test case B, missed deadlines, RMS implemented with SCHED_RR*

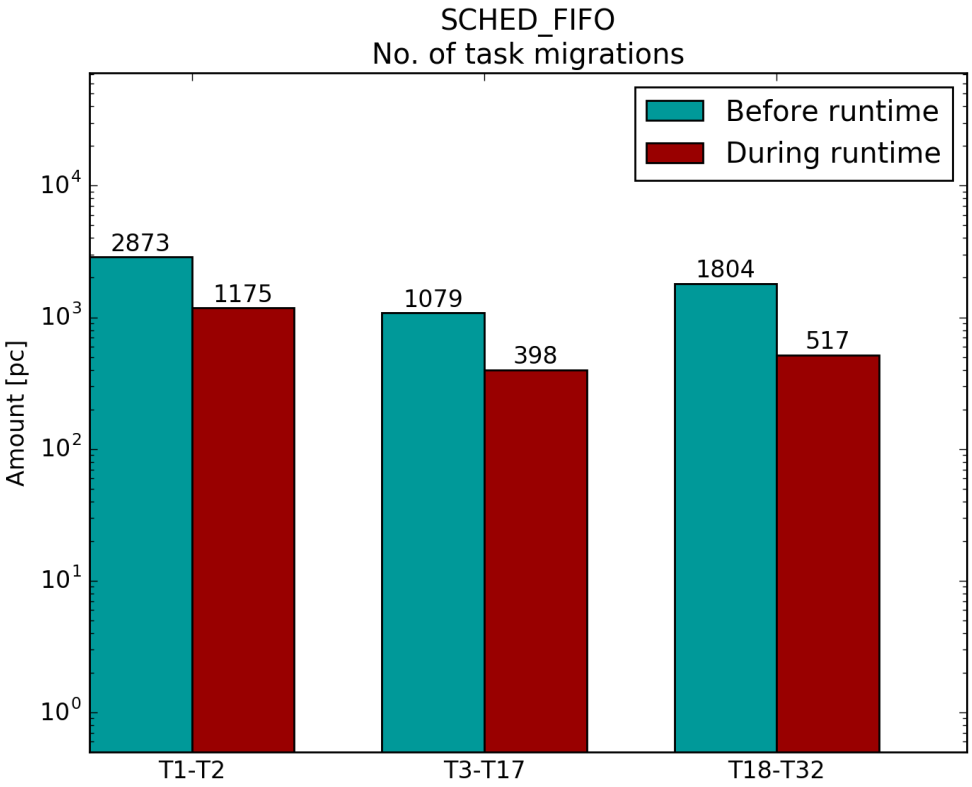


*Figure 4.41: Test case B, migrations, SCHED_FIFO*

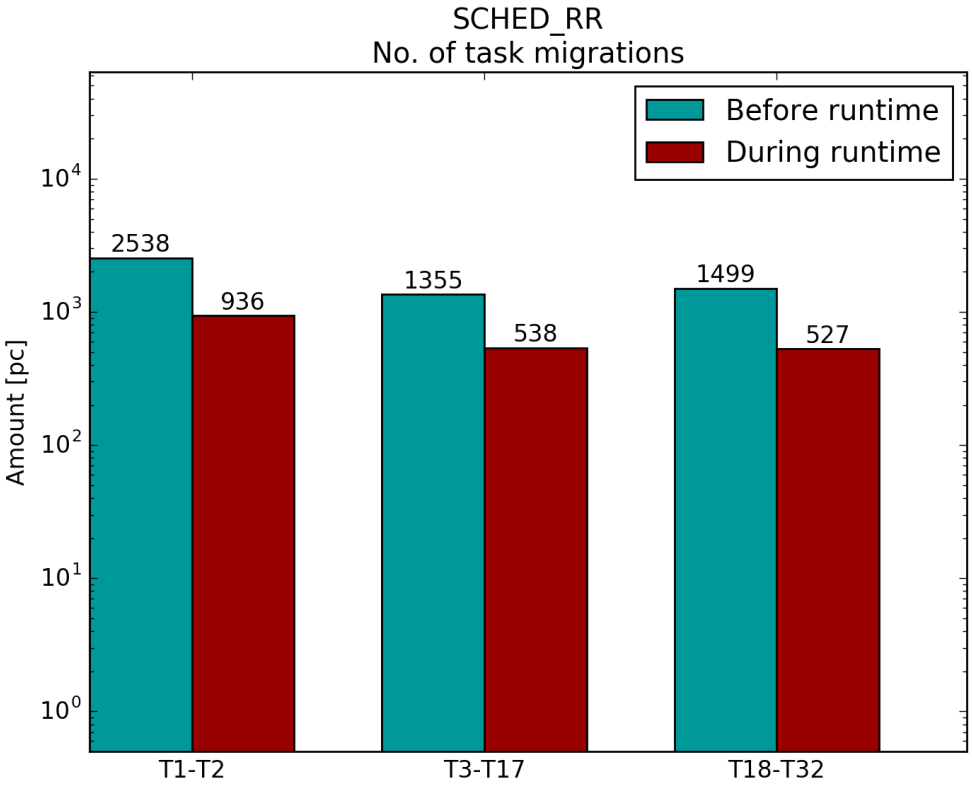


*Figure 4.42: Test case B, migrations, SCHED_RR*

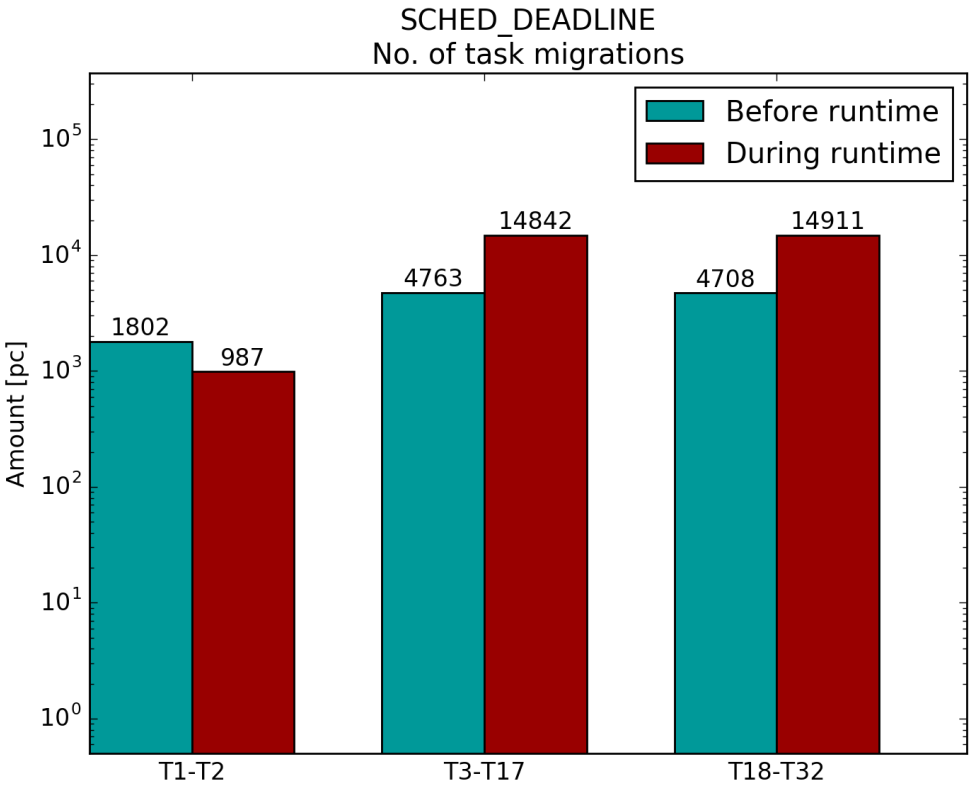


*Figure 4.43: Test case B, migrations, SCHED_DEADLINE median*

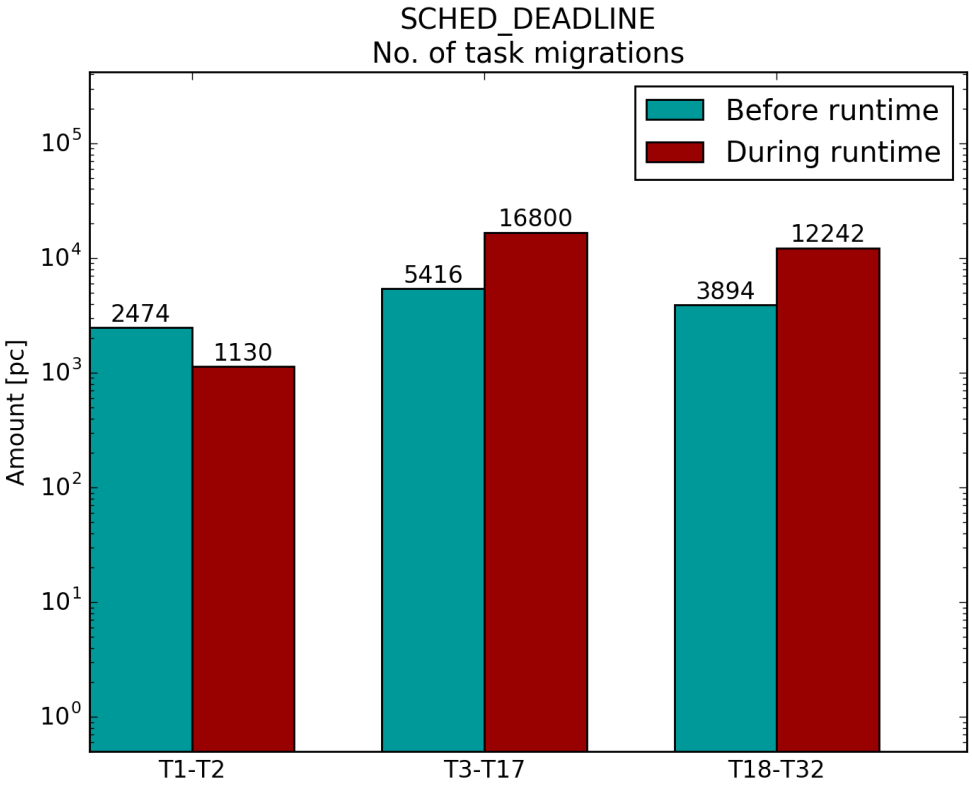


*Figure 4.44: Test case B, migrations, SCHED_DEADLINE median plus*

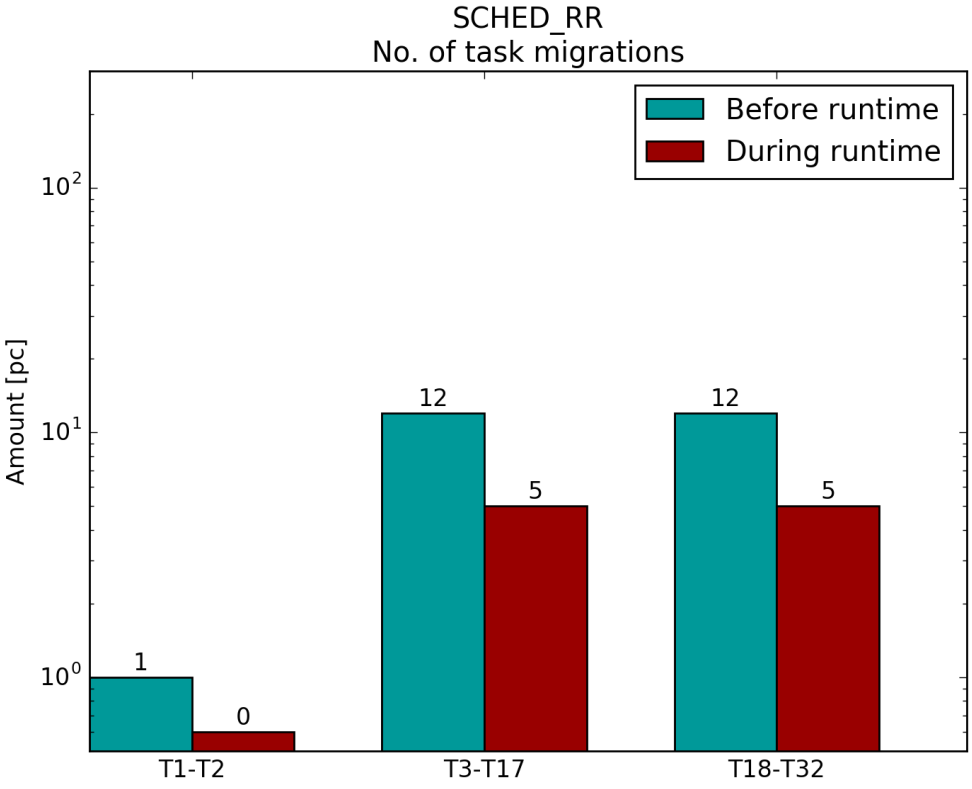


*Figure 4.45: Test case B, migrations, RMS implemented with SCHED_RR*

### 4.1.3 Test case C

Neither *SCHED_DEADLINE median* nor *SCHED_DEADLINE median plus* did fail to schedule a single thread.

As before, the response time from the different test runs are presented in the first five figures (Figure 4.46-4.50). Then follows the computation time in Figure 4.51-4.55 and the thread utilization in Figure 4.56-4.60. Next, the number of missed deadlines are shown in Figure 4.61-4.65. Finally, in Figure 4.66-4.70, task migrations are presented.

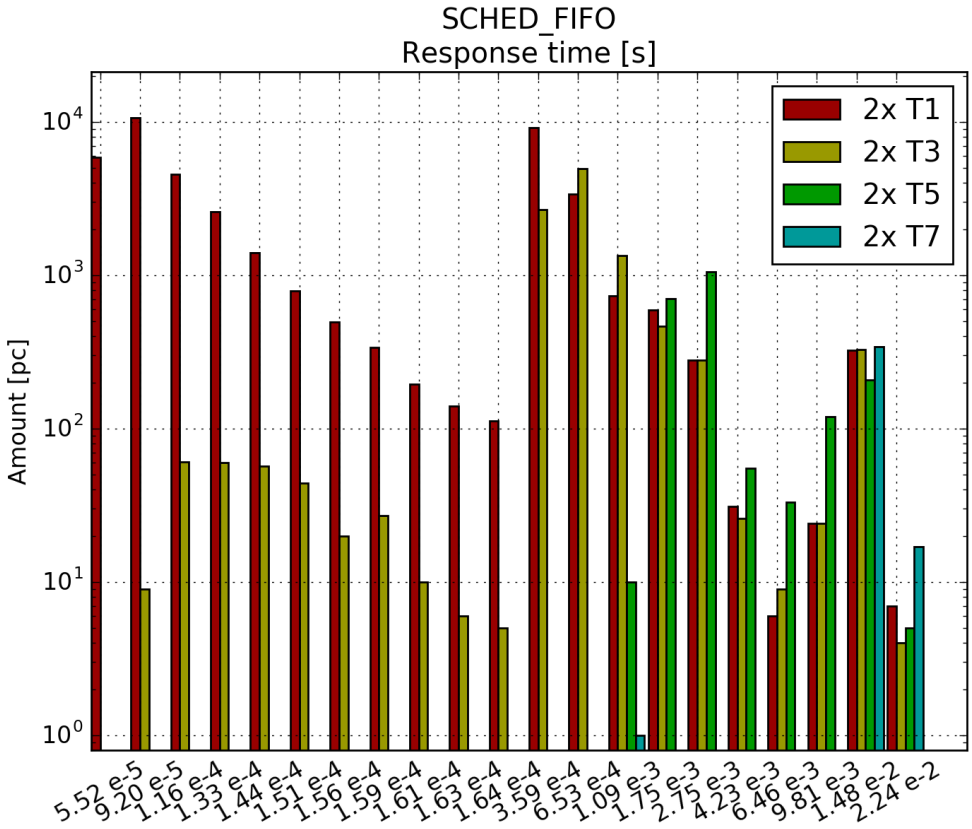


*Figure 4.46: Test case C, response time, SCHED_FIFO*

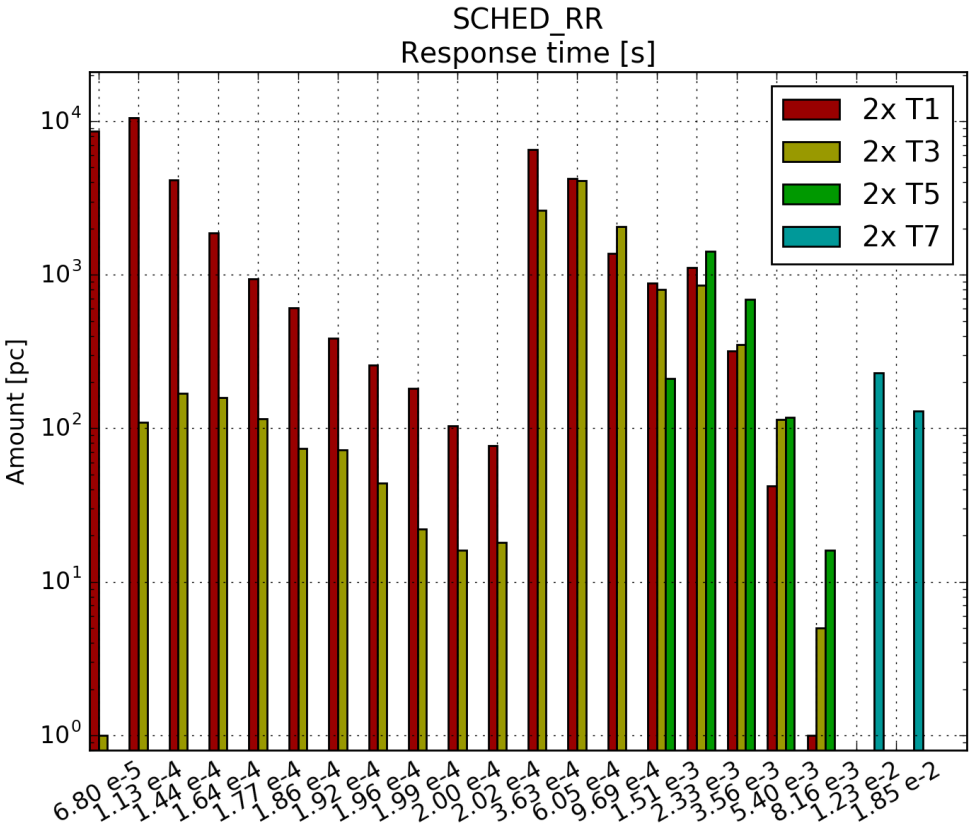


*Figure 4.47: Test case C, response time, SCHED_RR*

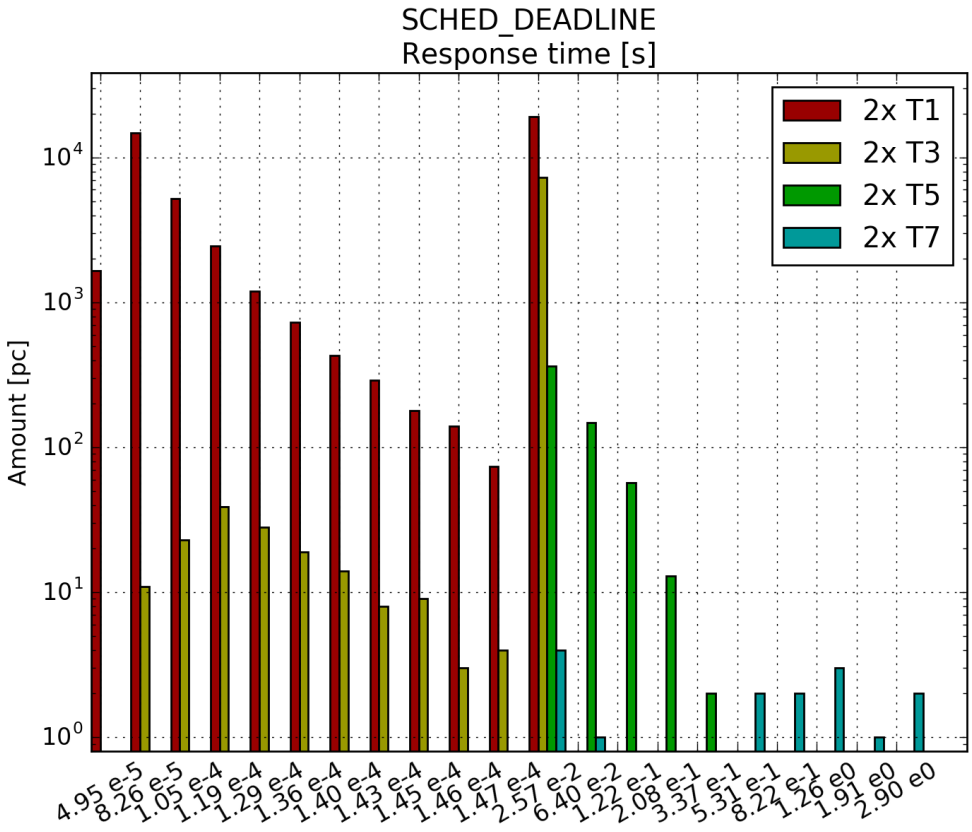


*Figure 4.48: Test case C, response time, SCHED_DEADLINE median*

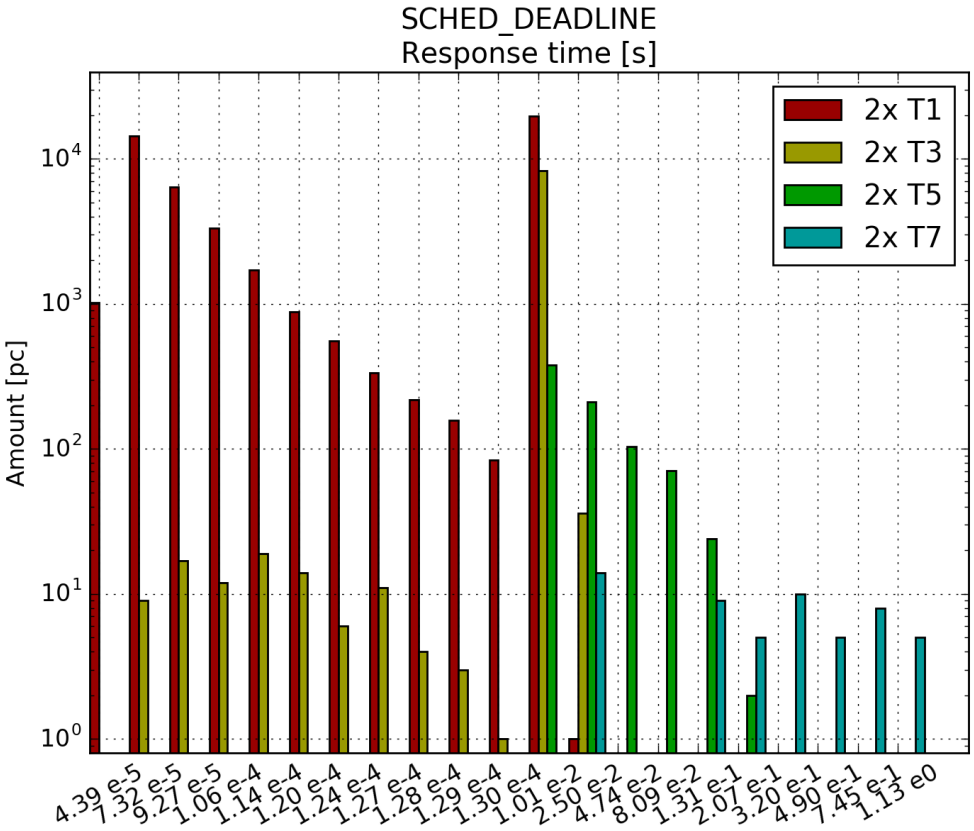


*Figure 4.49: Test case C, response time, SCHED_DEADLINE median plus*

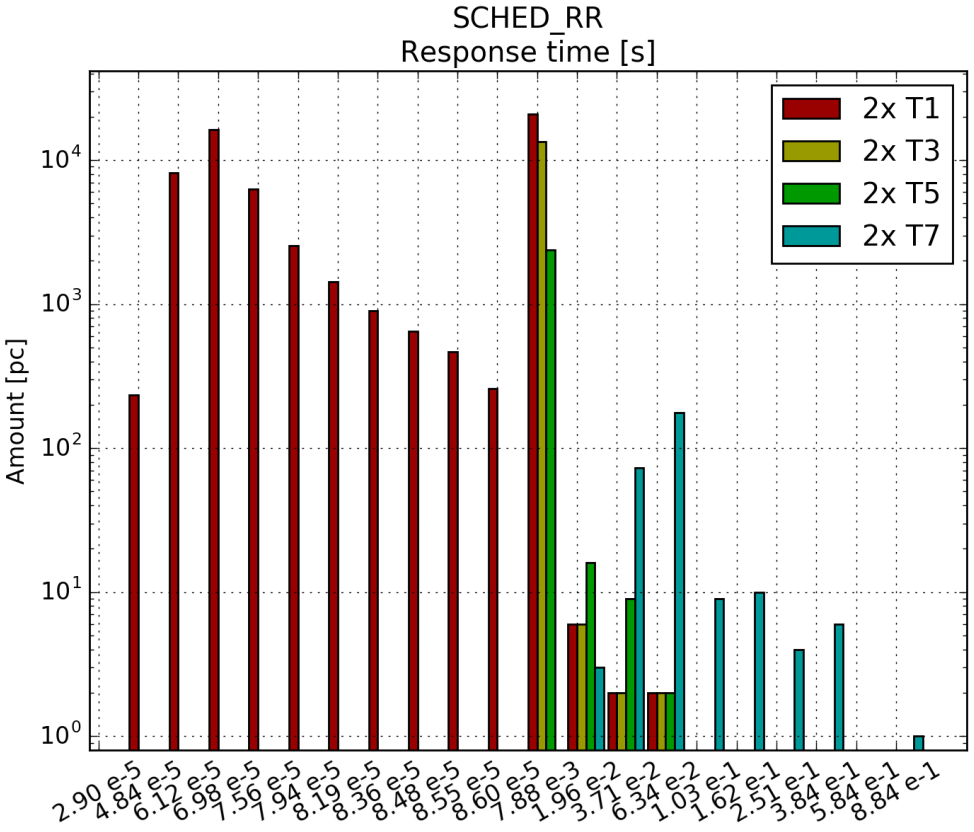


*Figure 4.50: Test case C, response time, RMS implemented with SCHED_RR*

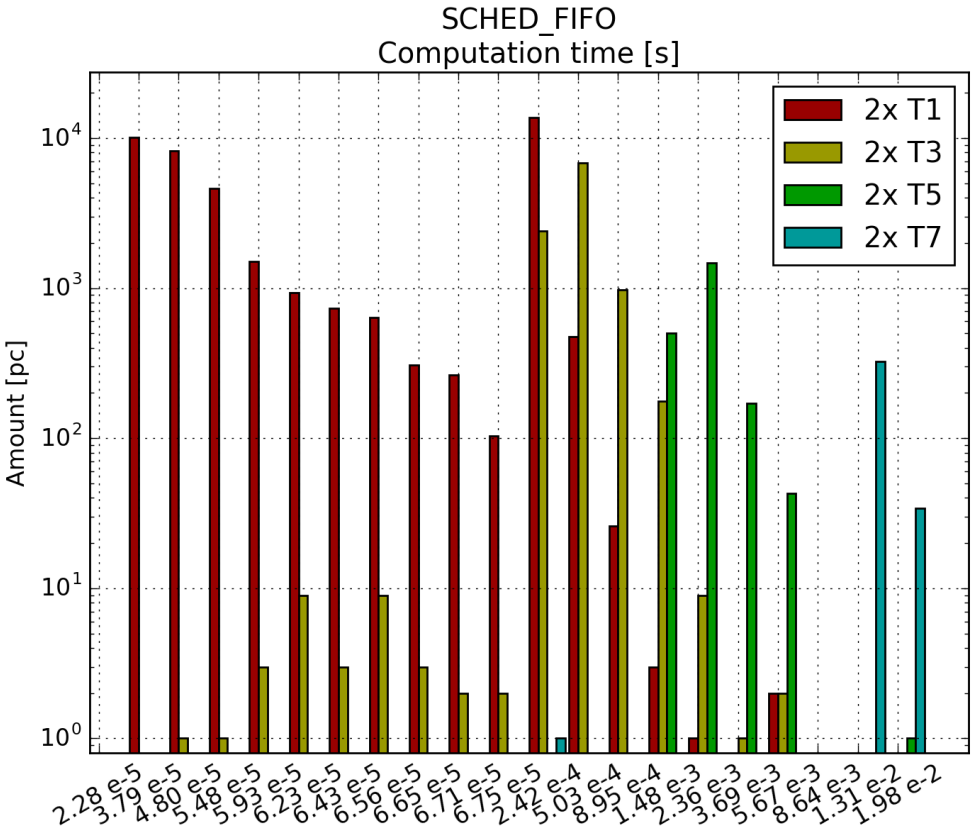


*Figure 4.51: Test case C, computation time, SCHED_FIFO*

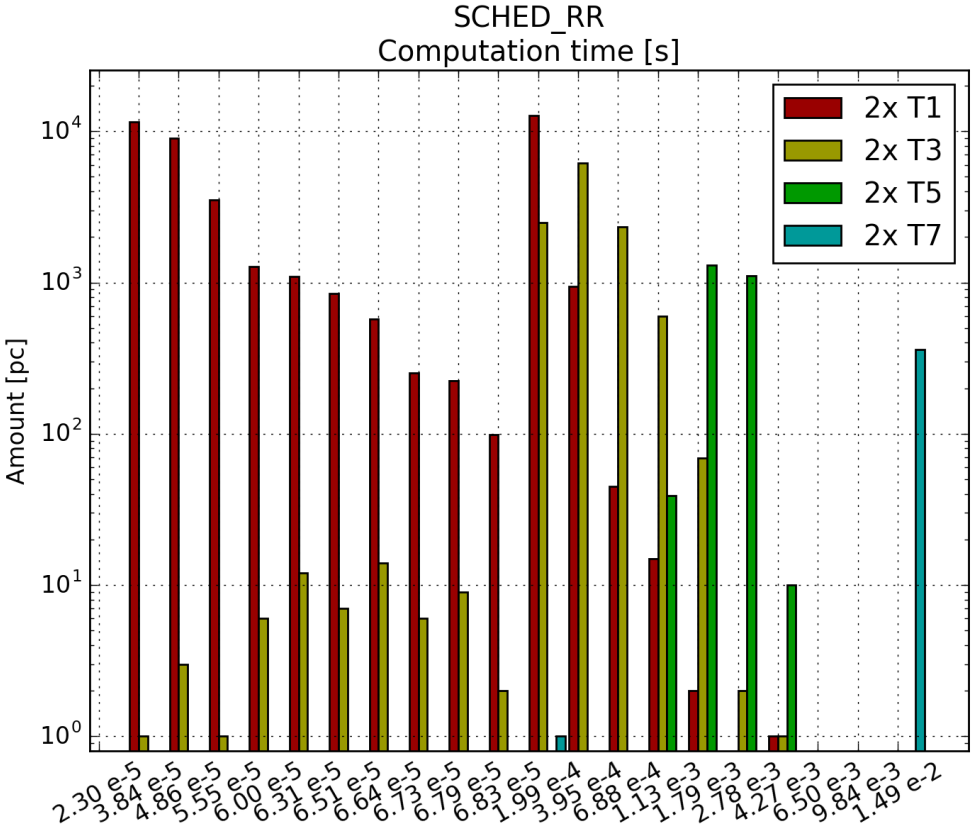


*Figure 4.52: Test case C, computation time, SCHED_RR*

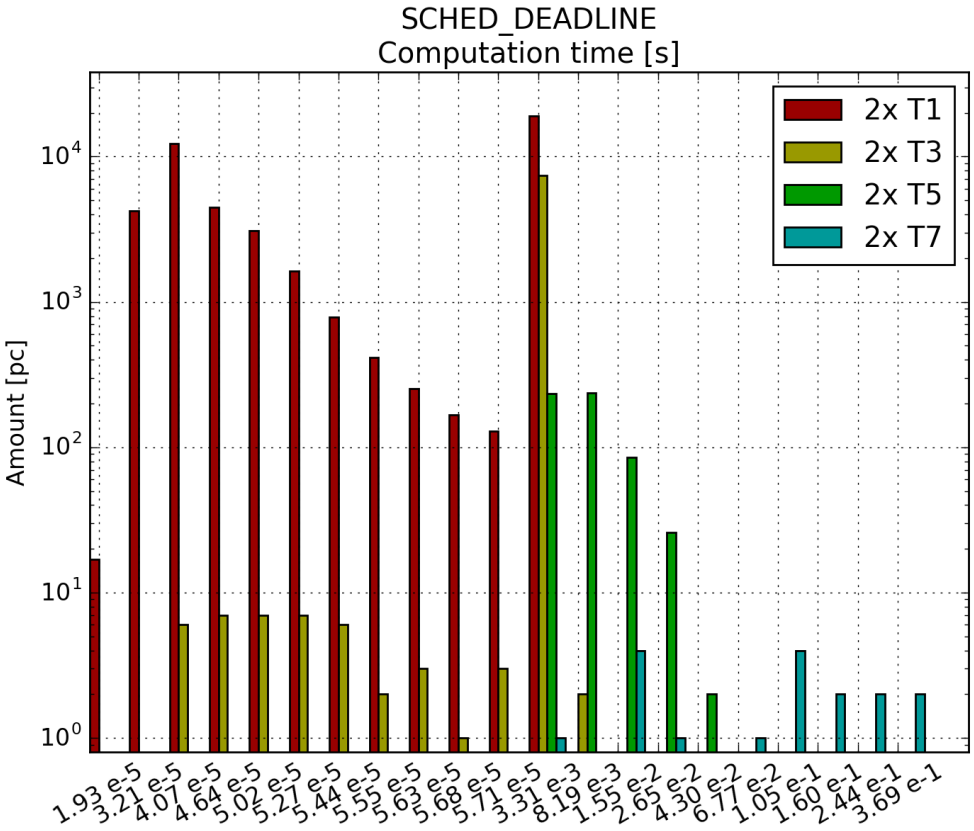


*Figure 4.53: Test case C, computation time, SCHED_DEADLINE median*

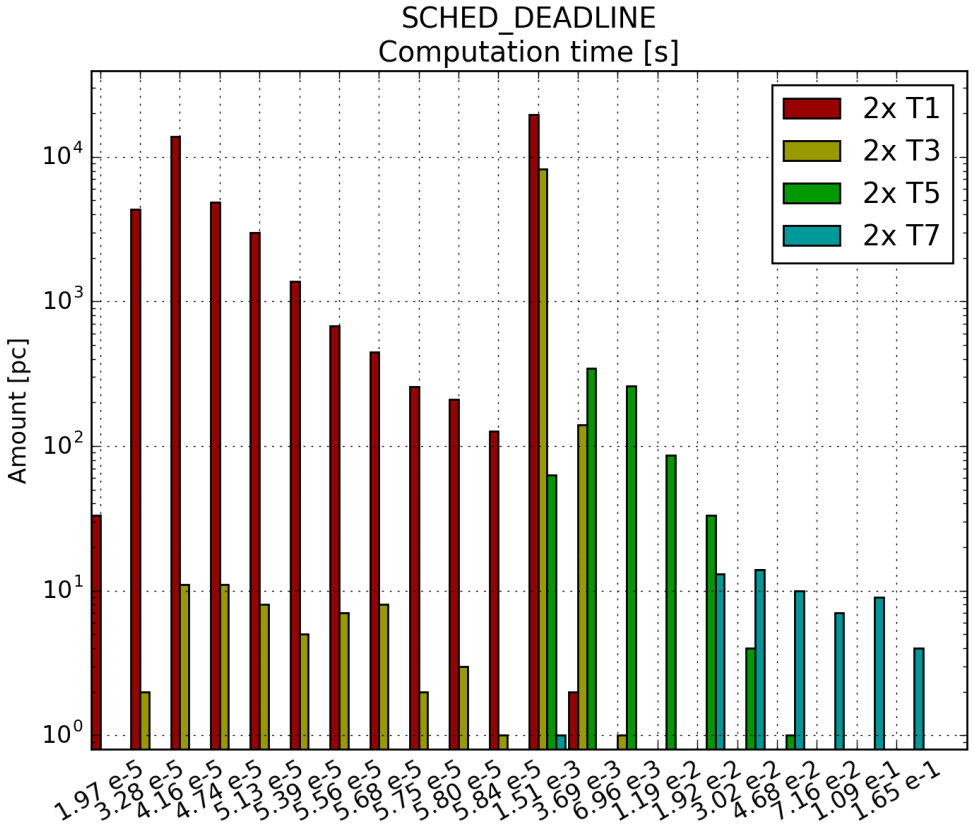


*Figure 4.54: Test case C, computation time, SCHED_DEADLINE median plus*

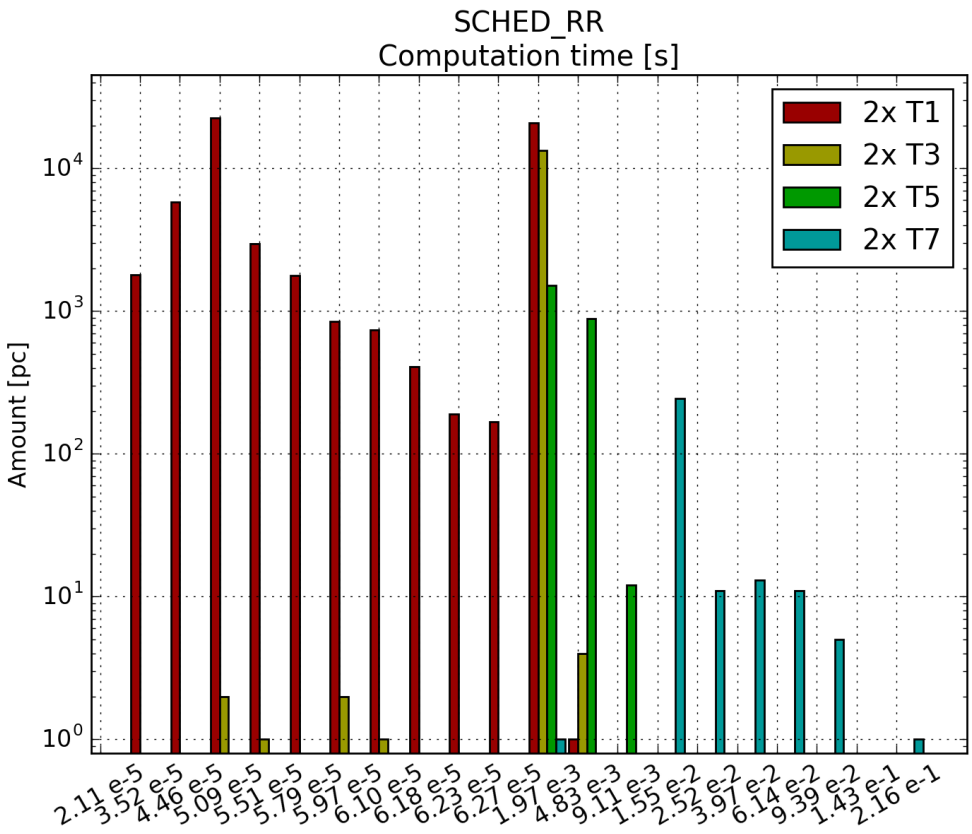


*Figure 4.55: Test case C, computation time, RMS implemented with SCHED_RR*

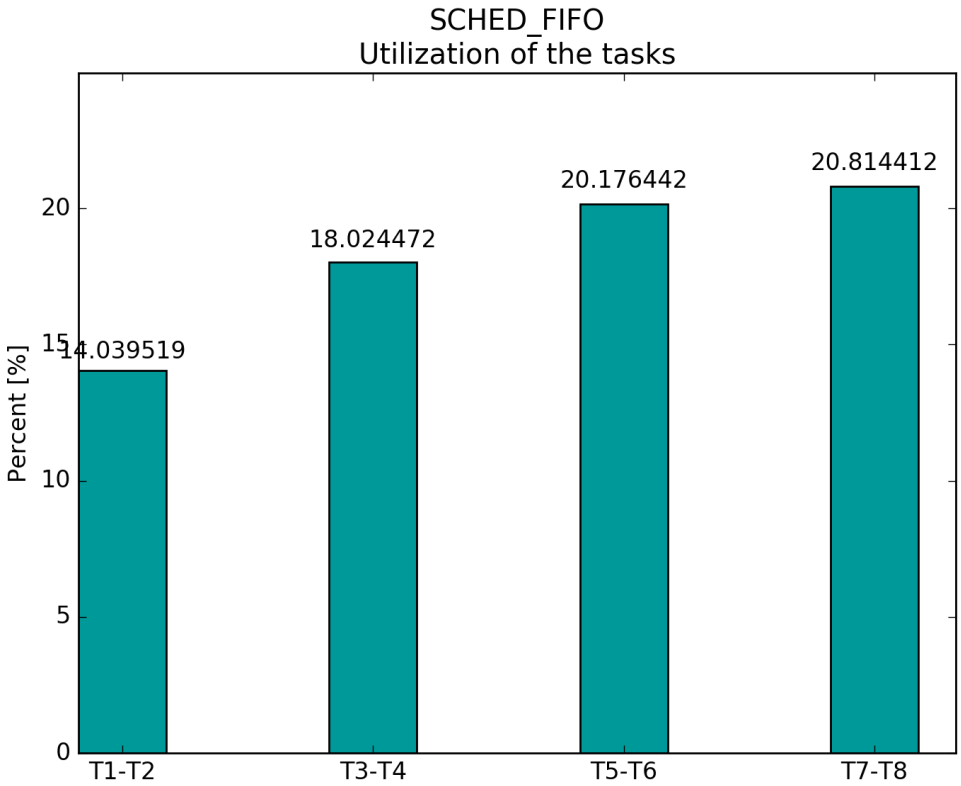


*Figure 4.56: Test case C, utilization, SCHED_FIFO*

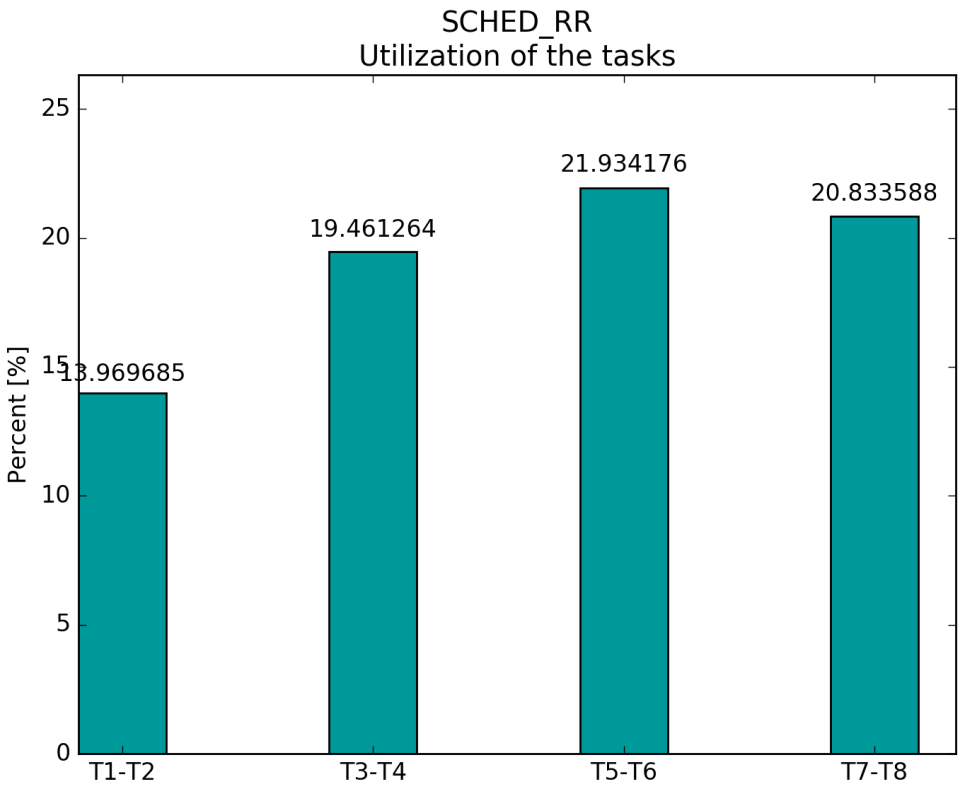


*Figure 4.57: Test case C, utilization, SCHED_RR*

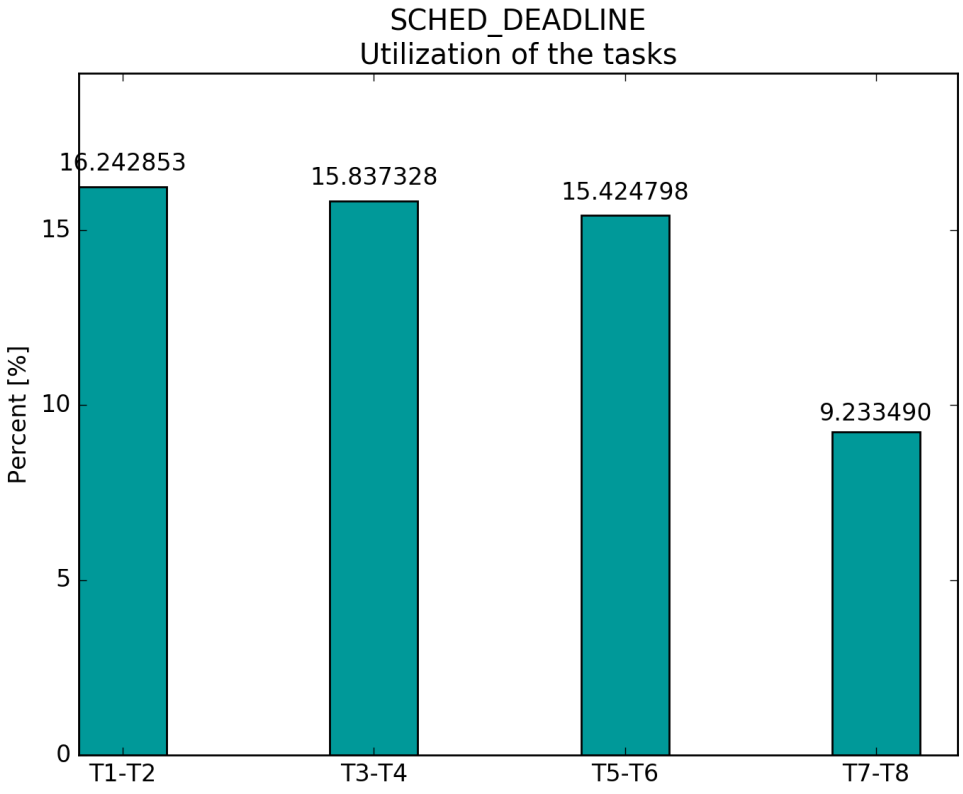


*Figure 4.58: Test case C, utilization, SCHED_DEADLINE median*

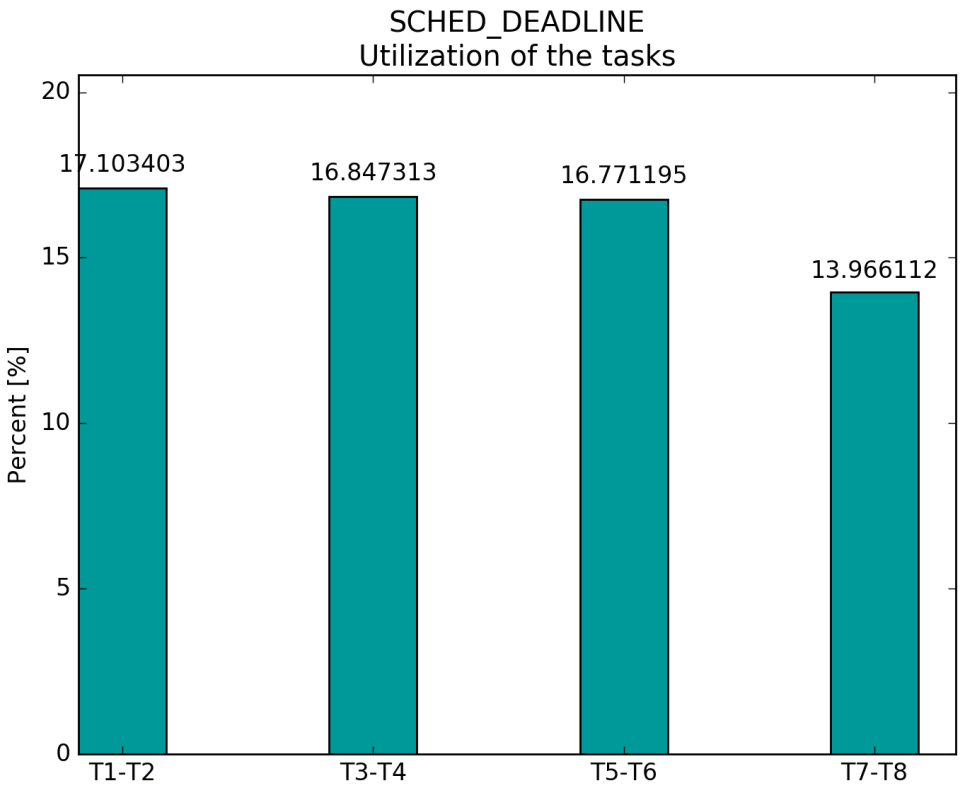


*Figure 4.59: Test case C, utilization, SCHED_DEADLINE median plus*

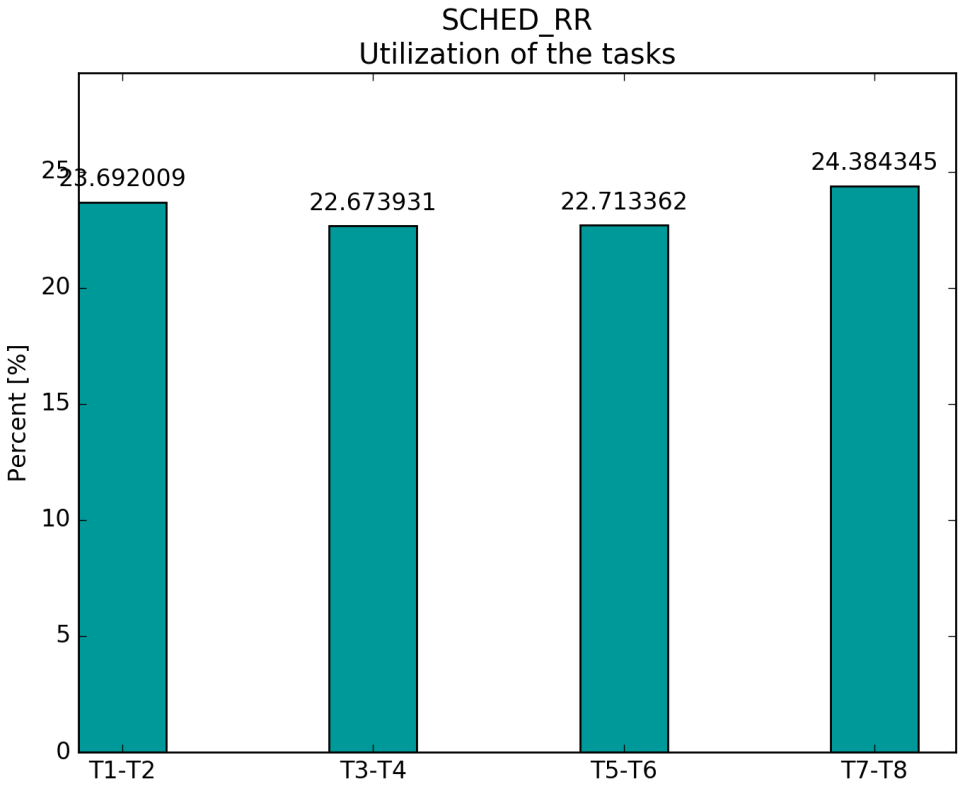


*Figure 4.60: Test case C, utilization, RMS implemented with SCHED_RR*

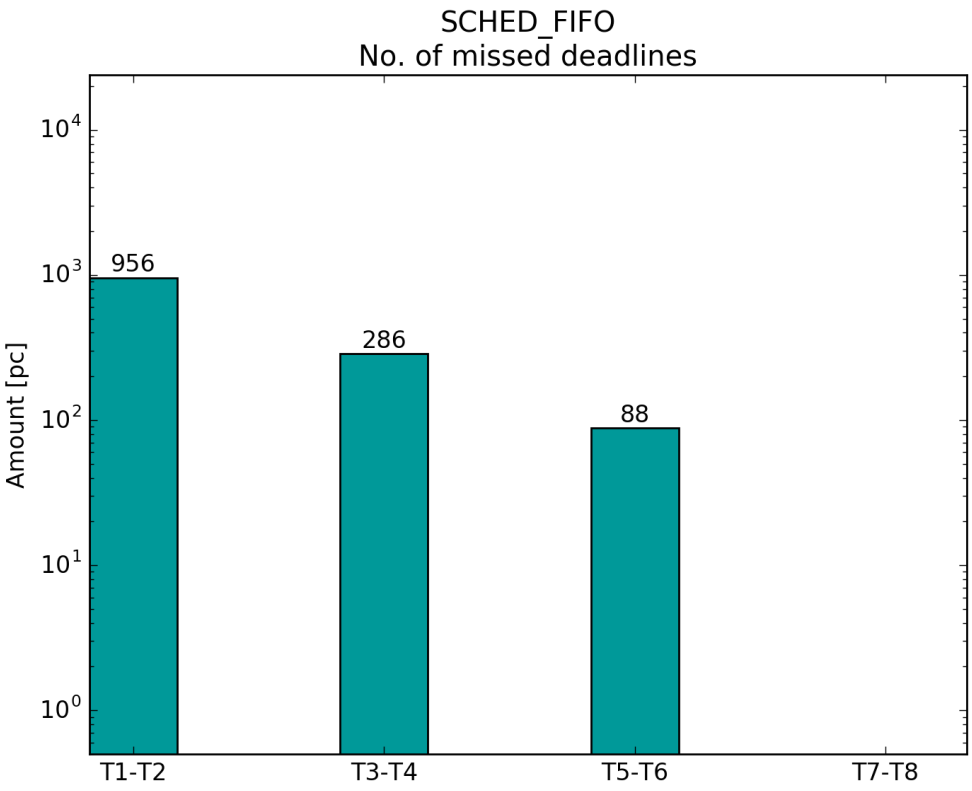


*Figure 4.61: Test case C, missed deadlines, SCHED_FIFO*

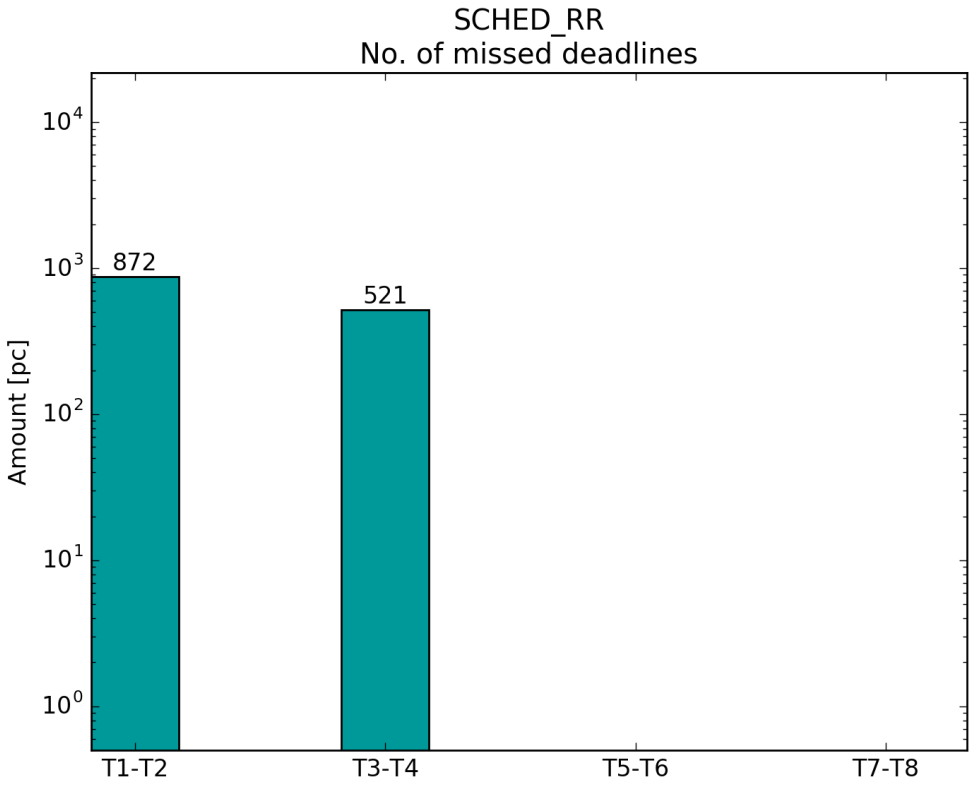


*Figure 4.62: Test case C, missed deadlines, SCHED_RR*

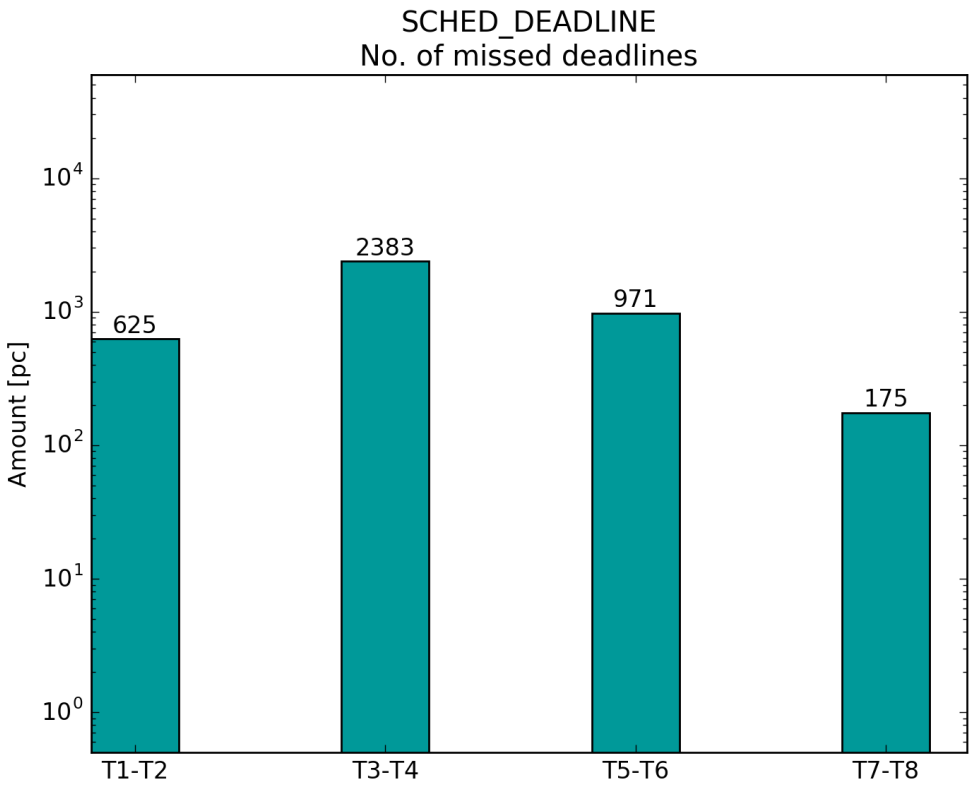


*Figure 4.63: Test case C, missed deadlines, SCHED_DEADLINE median*

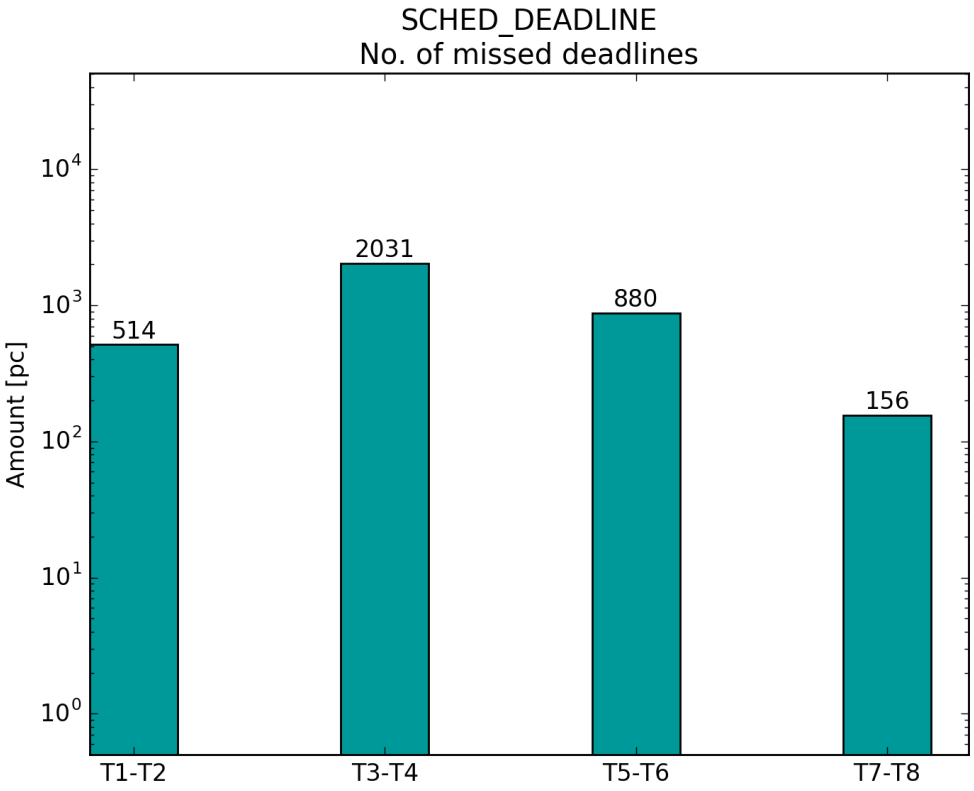


*Figure 4.64: Test case C, missed deadlines, SCHED_DEADLINE median plus*

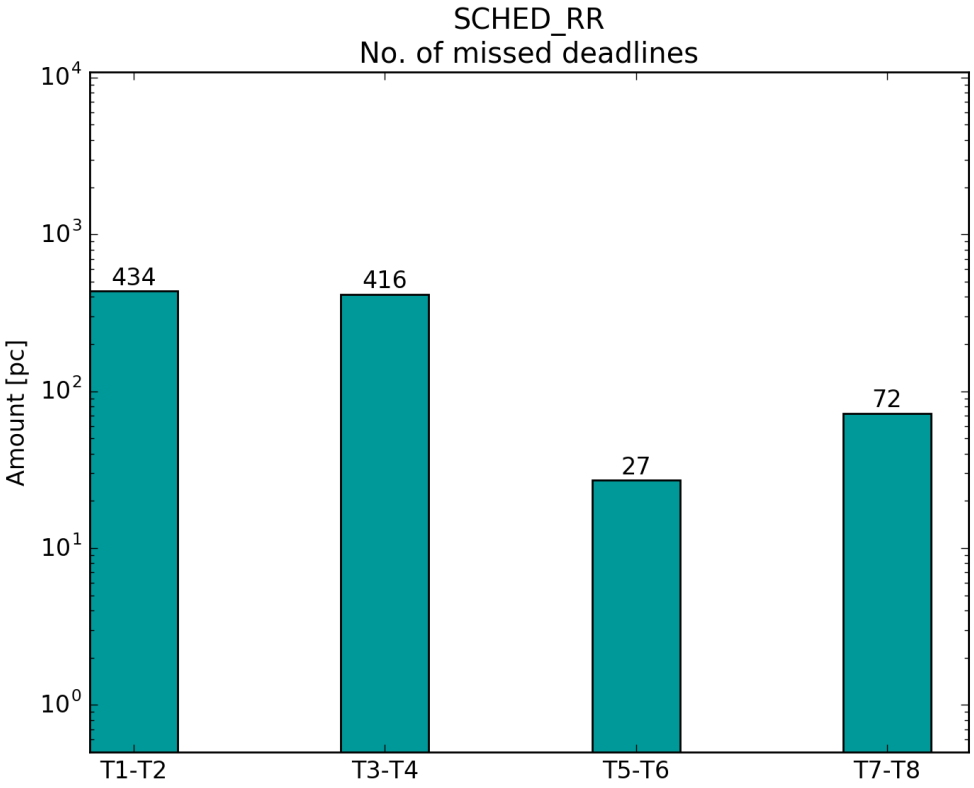


*Figure 4.65: Test case C, missed deadlines, RMS implemented with SCHED_RR*

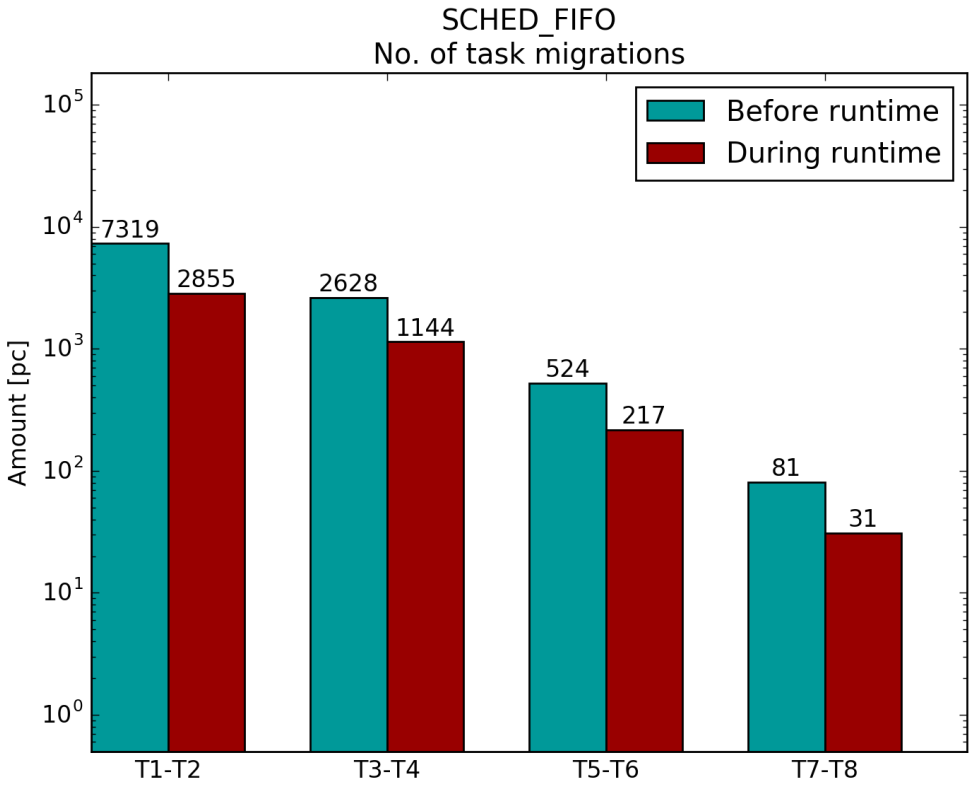


*Figure 4.66: Test case C, migrations, SCHED_FIFO*

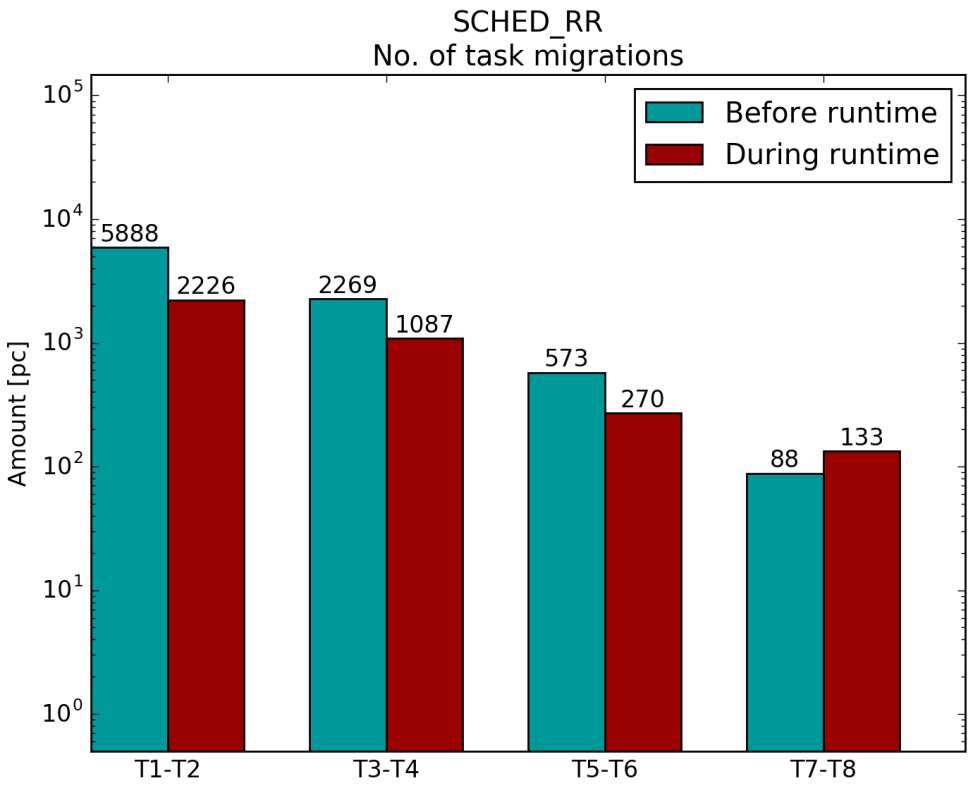


*Figure 4.67: Test case C, migrations, SCHED_RR*

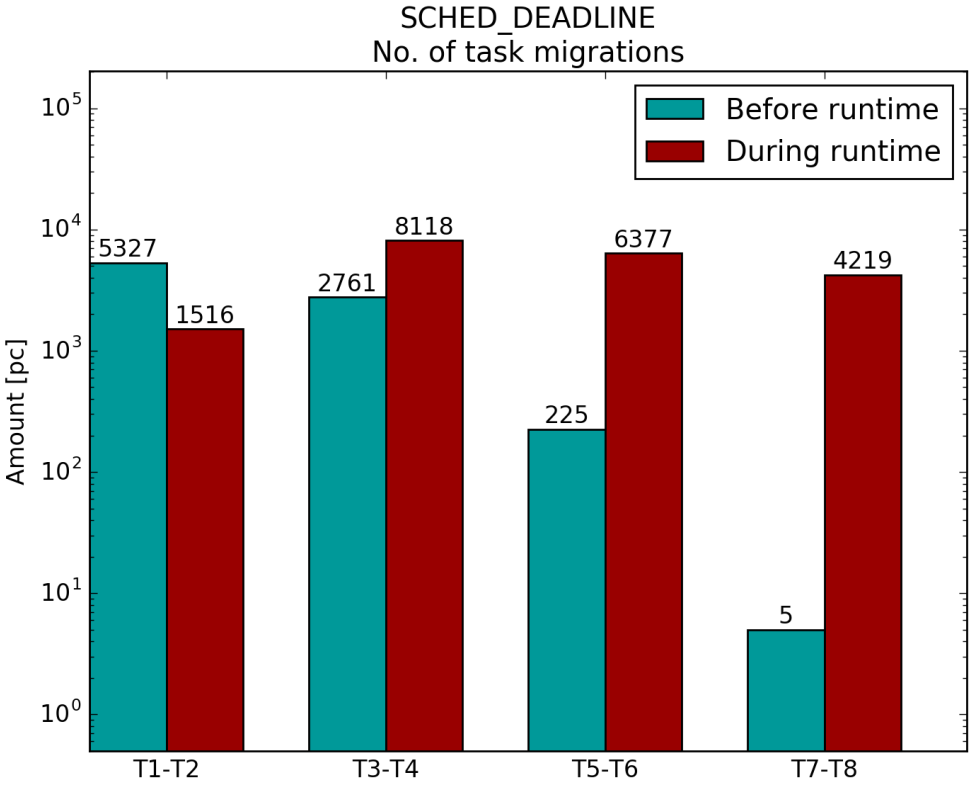


*Figure 4.68: Test case C, migrations, SCHED_DEADLINE median*

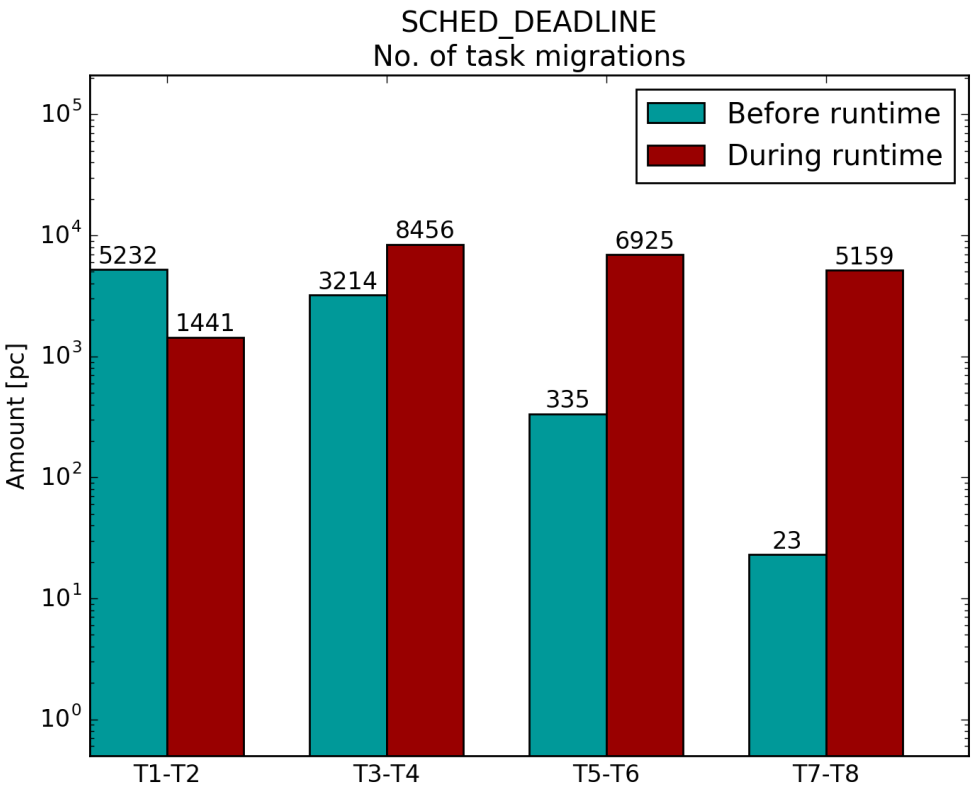


*Figure 4.69: Test case C, migrations, SCHED_DEADLINE median plus*

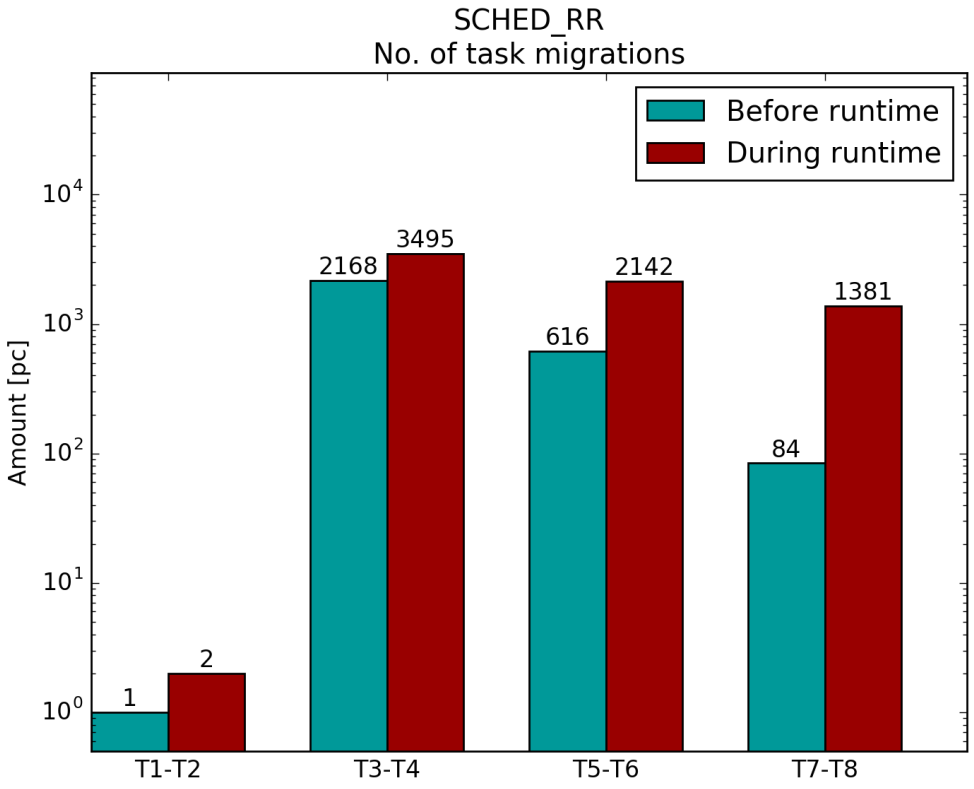


*Figure 4.70: Test case C, migrations, RMS implemented with SCHED_RR*

## 4.2 Ericsson's Application

The results from the test runs with the application provided by Ericsson are presented in this chapter. They have been collected from the test platform called monster machine. This means that they have been performed on a high performance workstation, with access to eight hardware threads.

The response time can be found in Figure 4.71 and 4.72, the computation time in Figure 4.73 and 4.74, the thread utilization in Figure 4.75 and 4.76, and the number of migrations in Figure 4.77 and 4.78. There are no figures showing results from the tests performed with the SCHED_DEADLINE policy. This is because there was no way to set different thread parameters (runtime, deadline and period) to different types of threads generated by the application. More about this in the discussion chapter 5.4.5 – *SCHED_DEADLINE with Ericsson's application*.

The different thread types in the figures below have been named dispatcher and worker threads to leave out their real names, since this is confidential information. There is one dispatcher thread and many worker threads.

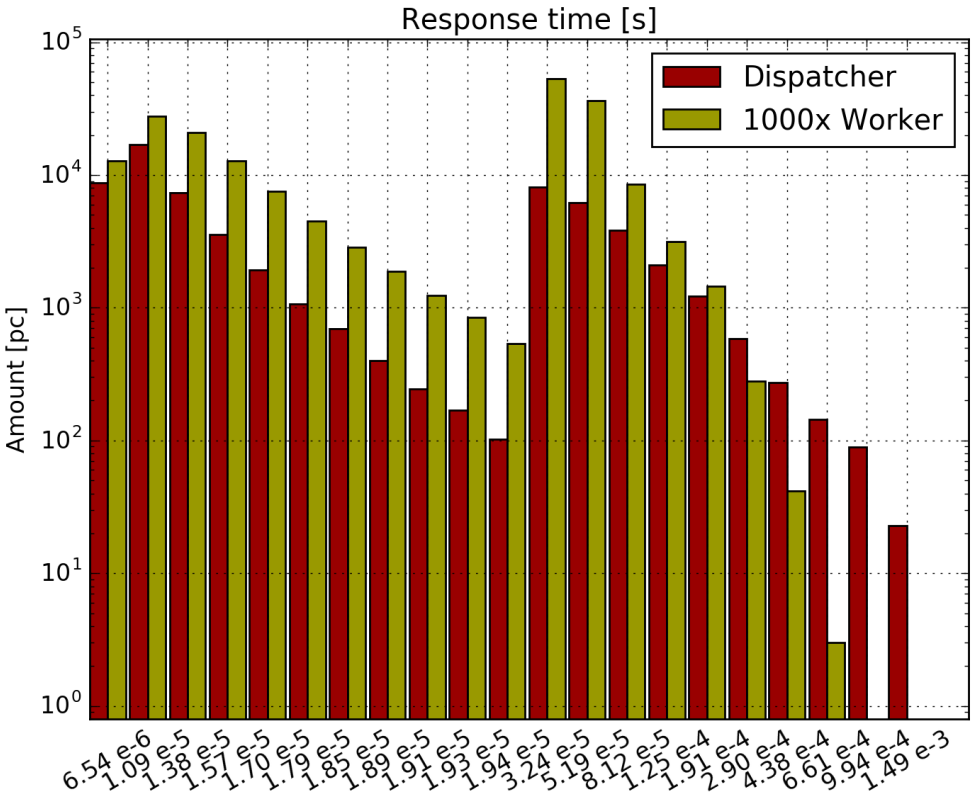


*Figure 4.71: Ericsson's application, response time, SCHED_FIFO*

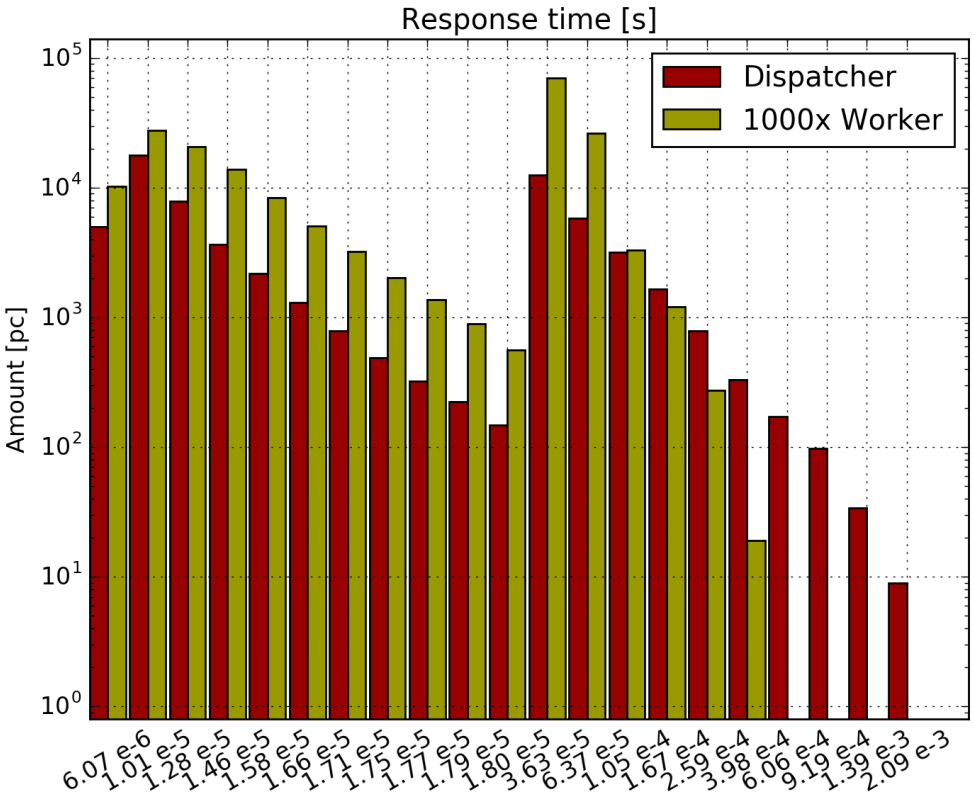


*Figure 4.72: Ericsson's application, response time, SCHED_RR*

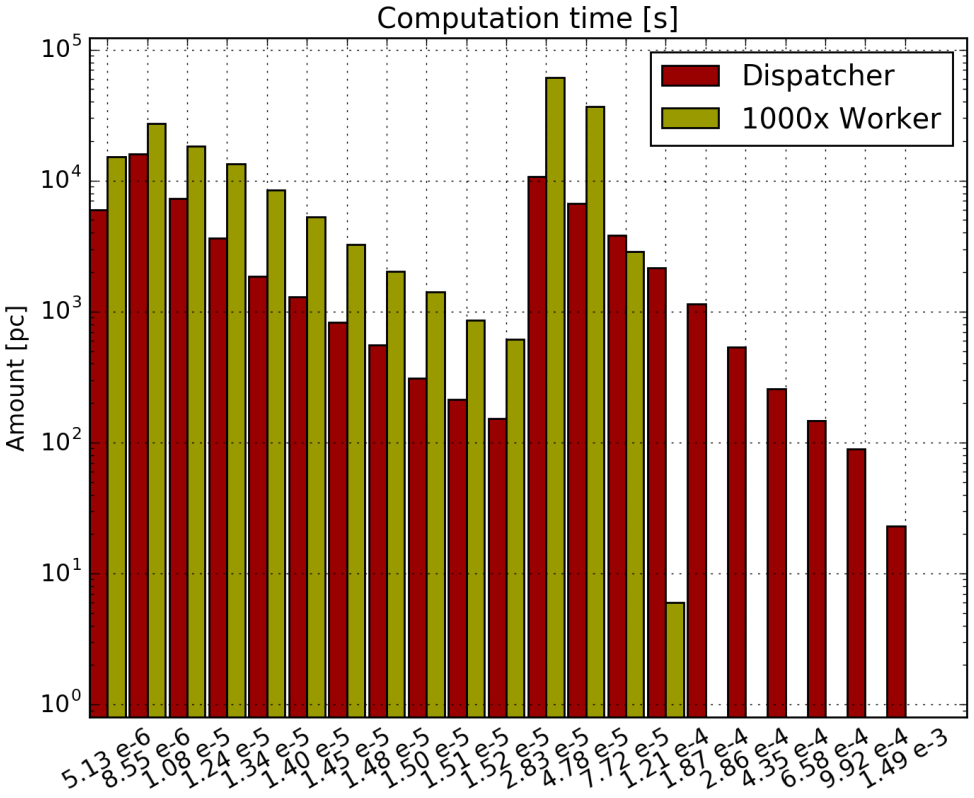


*Figure 4.73: Ericsson's application, computation time, SCHED_FIFO*

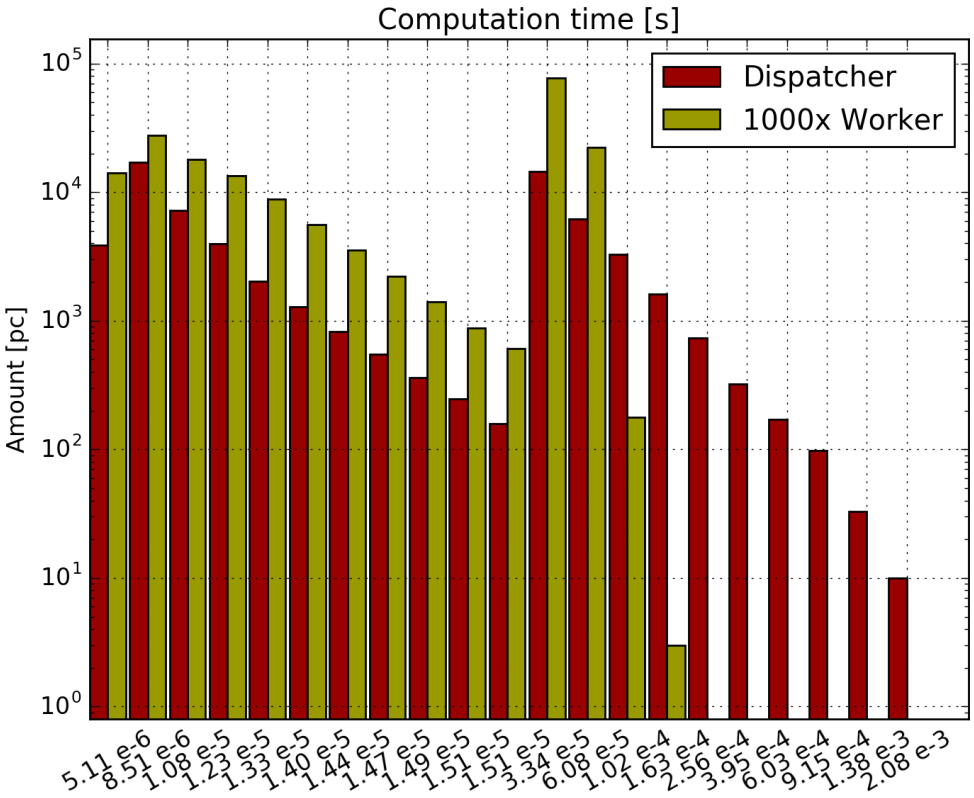


*Figure 4.74: Ericsson's application, computation time, SCHED_RR*

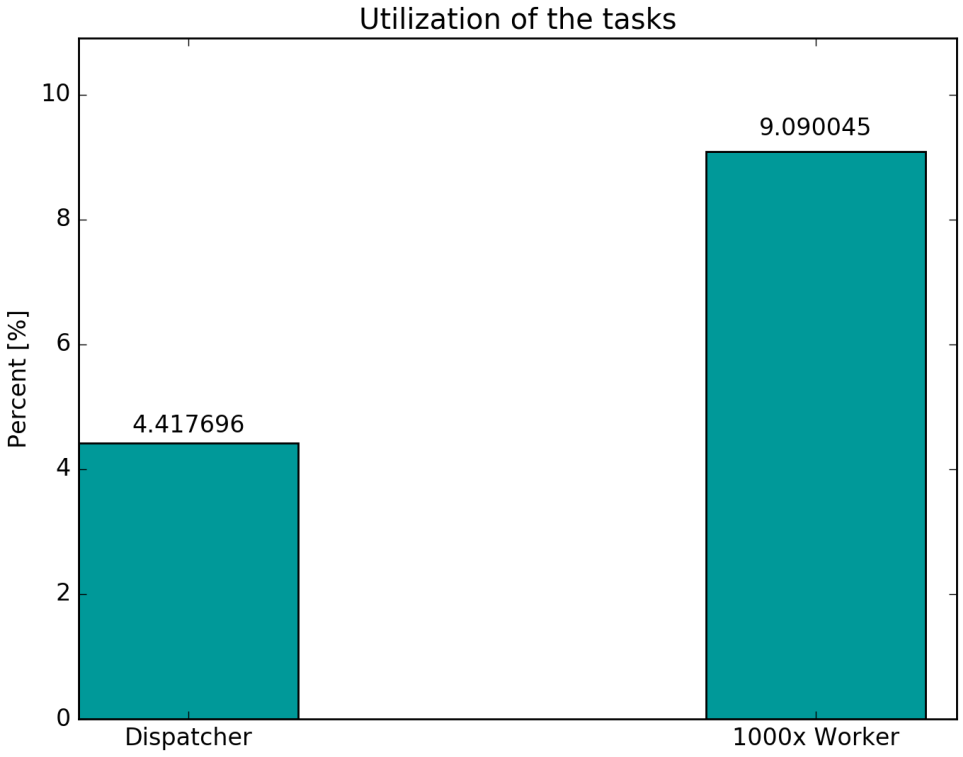


*Figure 4.75: Ericsson's application, utilization, SCHED_FIFO*

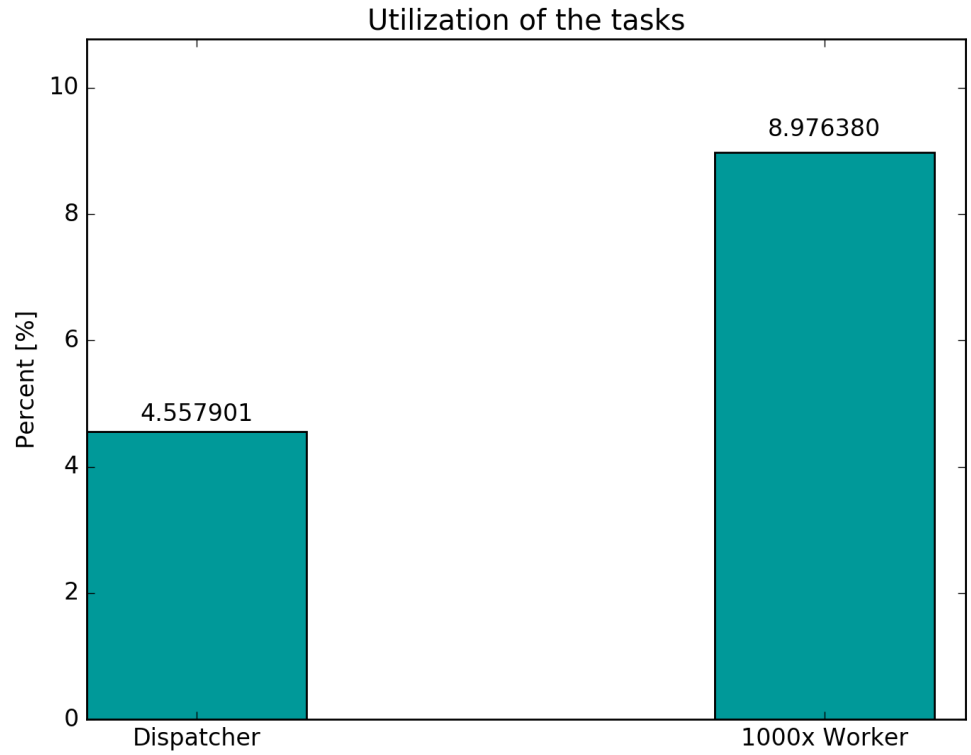


*Figure 4.76: Ericsson's application, utilization, SCHED_RR*

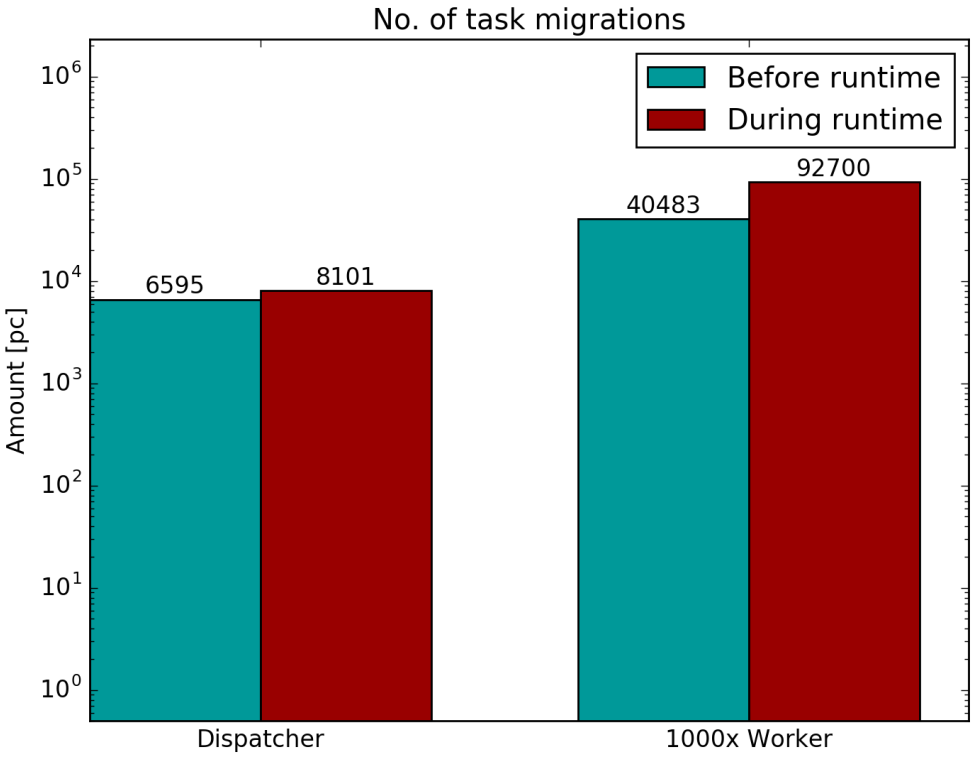


*Figure 4.77: Ericsson's application, migrations, SCHED_FIFO*

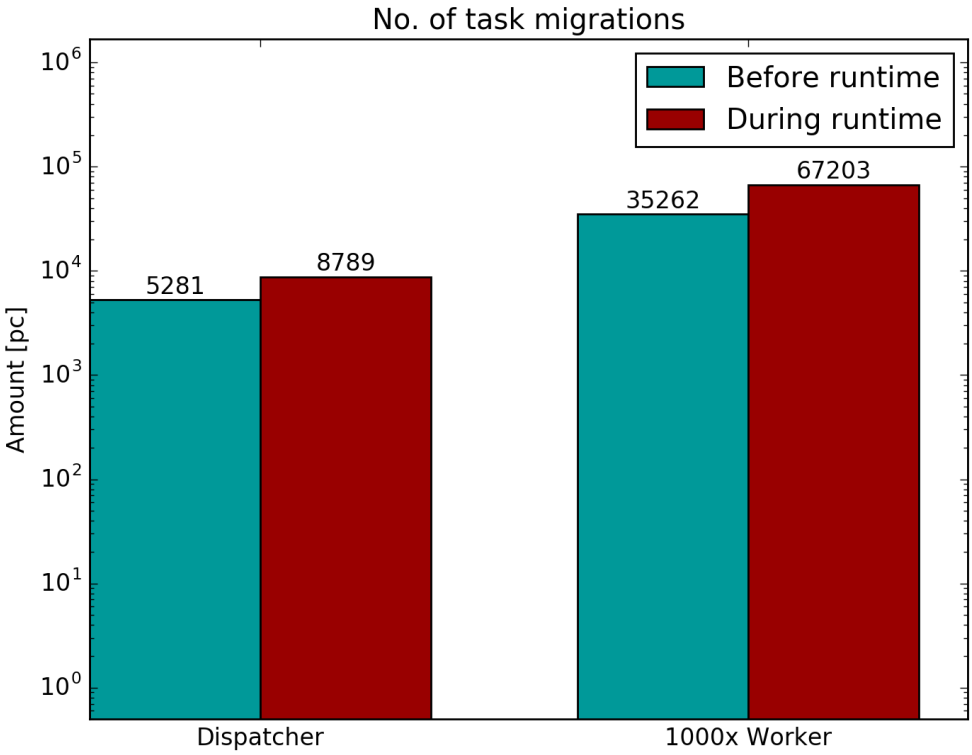


*Figure 4.78: Ericsson's application, migrations, SCHED_RR*

# Chapter 5

# Discussion

The results in chapter four will be discussed in the first three subchapters below. They include the test program, the application provided by Ericsson and results in general. Then a discussion about the project methodology will follow. Finally, a subchapter about ethical and societal aspects will be presented.

## 5.1 Results – The Test Program

There is no big difference between *FIFO* and *RR* in the three test cases. Response time, computation time and utilization are all similar for these two scheduling policies. In the histogram plots, the values in the middle are most interesting since it is difficult to know the exact value of an occurrence in a wide bin. The bins at the edges are larger than those in the middle. This is why the middle values are used for comparison between any pair of policies. The varying ranges of the bins is a feature of the equations described in the method chapter above. The large result values may not be fully correct either. An explanation of this is given in chapter 5.4.3 – *Ignoring events*. An alternative explanation is given in 5.1.3 – *Test case C*.

Regarding the computation time, almost all measurements are smaller than 1 ms. Only tasks T5-T8 in Figure 4.51 and 4.52 (thread types three and four in the test case C) have a longer computation time. If the computation time is smaller than the time slice, there is only one runtime segment. Since 1 ms is the value of the time slice used for SCHED_RR in this project, *FIFO* and *RR* should in theory (see chapter 2.5) behave the same way because their implementations are similar. The number of missed deadlines and migrations are however not the same. Deadline misses vary between the three different test cases. In test case A, *FIFO* has a lot more misses than *RR*. In test case B, *RR* misses more instead, and in test case C they have almost the same total amount of misses. The number of migrations for the SCHED_FIFO policy are significantly larger than with SCHED_RR. This is an observation for all three test cases that cannot be explained. Neither can the varying number of deadline misses be explained. They should behave the same according to the theory. Due to thesis timing constraints, this discrepancy was not investigated further.

Not only are the results from *FIFO* and *RR* similar, but the results from the tests *SCHED_DEADLINE median* and *SCHED_DEADLINE median plus* are also similar to each other. The response time and the computation time are close, but *SCHED_DEADLINE median plus* is always a little better. The bins for the minimal, median and maximal bin are in general somewhat smaller for the *SCHED_DEADLINE median plus*. This indicates that the jobs for *median plus* manage to finish during its period. This is further supported by the fact that *median plus* has fewer deadline misses. The utilization of the tasks and the migration

numbers differs between the test cases and will be therefore explained in the subchapters below.

An observation common for all test cases is that the processor utilization, the sum of all thread utilization in a test case, is significantly higher for SCHED_RR and SCHED_FIFO than for SCHED_DEADLINE. Why SCHED_FIFO and SCHED_RR get higher utilization is not so strange given that when a new task wakes up, it will place itself in a queue; the only real idle time for those policies will then be when the ready queue is empty, or when they are throttled, which per Linux default will be about 5 % of the processor time. For test cases B and C, the utilization for *RMS* is even higher than that of SCHED_RR and SCHED_FIFO with a single static priority. It is because *RMS* is better at placing threads in the ready queue for each of the runs of the threads; while normal SCHED_FIFO or SCHED_RR will sometimes have a thread in the ready queue for long enough to miss an entire period (for tasks with short periods relative to the other tasks), effectively causing it to have one period of work worth of less utilization. This is why *RMS* has better utilization than the two *RT* policies.

Why SCHED_DEADLINE gets a lower utilization is more unclear since it, like *RMS*, will wake all threads up often enough to be able to start their work before their period ends. There are a few possibilities here. First, there is lot of overhead in SCHED_DEADLINE outside of the actual runtime of the scheduled tasks, such as work for the scheduling decisions being done in a background program. Another possibility is that once a task reaches its runtime for a period, it is suspended by *CBS*, leaving the background programs more time to run.

The calibration of the test cases was based on the median value of the runtime. The figures showing the computation time reveals that this value can be found near the median in each figure. Both versions of SCHED_DEADLINE in test case C are an exception though. This is explained in chapter 5.1.3 – *Test case C*.

### 5.1.1 Test case A

The key feature of this task set was that all thread types shared the same thread parameters, except for the delay. Each individual figure in chapter 4.1.1 – *Test case A* has an expected result where the three different thread types are behaving the same. Their performance is almost identical to each other. In the test with *SCHED_DEADLINE median plus*, one can see the effect of the thread that has not been allowed to schedule. The first thread type has always a lower amount or percentage than the other two. Since the first thread type has one task less, these results are expected too.

Before comparing and discussing SCHED_DEADLINE (both *median* and *median plus*) with *FIFO* and *RR*, the test case specific results (not mentioned above) will be walked through. When the two SCHED_DEADLINE versions are compared to each other, the utilization and the number of migrations must be discussed. In this test case the *SCHED_DEADLINE median plus* migrates less than *SCHED_DEADLINE median* during runtime. It has therefore less overhead than *median*, assuming migrations during runtime are more expensive, which they should be in most architectures. The total number of migrations is almost the same. The processor utilization is the same, but the utilization for the thread types are distributed differently due to the different amount of running threads.

Now, to the comparison of SCHED_DEADLINE with *FIFO* and *RR*. SCHED_DEADLINE has a strictly better (smaller) response time, see Figure 4.1-4.4. It has also many occurrences under 1 ms, which neither *FIFO* nor *RR* have. The computation time is similar for the tested policies. Regarding the utilization, SCHED_DEADLINE has a somewhat smaller utilization per thread type on average. It also has a significantly higher amount of migrations compared to *FIFO* and *RR*. As described in chapter 2.4 – *Earliest Deadline First Scheduling, GEDF* checks the new or changed task against all running tasks in the system. This means that SCHED_DEADLINE will have more preemptions and therefore more migrations. *FIFO* and *RR* only take the next task when a task is finished. The amount of missed deadlines is a bit different between the policies. *SCHED_DEADLINE median* has more misses than *FIFO*, which has more misses than *RR*. *CBS* may be responsible for this. The lower amount of misses for *SCHED_DEADLINE median plus* compared to *median* may be caused by the lower amount of running threads.

SCHED_DEADLINE therefore has a much better response time, similar runtime, lower utilization, more missed deadlines and more migrations than *FIFO* and *RR* for test case A.

### 5.1.2 Test case B

In this test case, the effect that can be seen due to the threads (for *SCHED_DEADLINE median*) that have failed to schedule is as expected, as in test case A. The third thread type always has a lower percentage than the second thread type, where both are slave tasks with the same parameters except the delay. Four threads of type three have not been allowed to schedule in this test case. This is why T18-T32 in Figure 4.33 has a lower utilization than in Figure 4.34. This is also why thread type three has a lower amount of migrations in *SCHED_DEADLINE median plus* than in *median*. The total number of migrations are although higher for *median plus*.

Test case B was designed to behave like a more typical application. It was expected that SCHED_RR and SCHED_FIFO would perform okay in theory, and that *RMS* and SCHED_DEADLINE should perform well, if not flawlessly. That *RMS* would always run the master task first is a typical behaviour for priority based scheduling in general, and that SCHED_DEADLINE would act more dynamically. *RMS* should show fast response time on the master task, and a spread out response time for the slave tasks. It is also expected that SCHED_RR and SCHED_FIFO would have a varied response time based a lot on random chance and that SCHED_DEADLINE would show a random response time for all tasks as well, but always finish tasks in time.

Looking at the response time for test case B, both SCHED_FIFO and SCHED_RR are bad at reaching the deadline for the thread of type one. SCHED_DEADLINE (both *median* and *median plus*) performs slightly better, but it has some tasks that finish in about two periods. This clearly displays the behaviour of *CBS*, when a thread overshoots its runtime. The task set for *SCHED_DEADLINE median plus* show that more tasks manage to finish within its first period, but it still has some of the tasks running at around a period plus a little extra. The extra time is because it did not finish before it was suspended by *CBS* and had to wait for a new period to get runtime. *RMS* shows a clear spike at the start of the graph, but since the bins in the figure (4.25) are so large here, the first bin alone encapsulates the majority of the runs of thread type one. It even contains values associated with missed deadlines. This makes it hard

to know if these master threads have a better response time than with the other policies. The slave tasks have a longer response time however, so SCHED_DEADLINE has a better response time than *RMS* overall.

Figure 4.40 shows that the number of missed deadlines for the master tasks is quite well handled in the *RMS* test run. The missed deadlines of SCHED_FIFO and SCHED_RR can be misleading, displaying that they miss fewer times than SCHED_DEADLINE. The response time graphs show the opposite, the number of occurrences within the deadline for thread type one are more in the case of SCHED_DEADLINE. This indicates that SCHED_FIFO and SCHED_RR misses several periods, probably due to ready queues containing several instances of the slave threads between each run of the master thread. When the slave tasks are considered, the SCHED_DEADLINE misses a lot more deadlines than *FIFO* and *RR*. Their slave tasks almost never miss a single deadline. SCHED_DEADLINE also misses more than *RMS*.

One cannot see any significant overhead differences between the different scheduling policies by looking at the computation time. The utilization figures reveal that SCHED_RR and SCHED_FIFO have a lower utilization for the master tasks. This is because they do not get to run as often as the slave tasks which results in that they sometimes miss their entire periods. Otherwise, the total utilization is higher for the *RT* policies than for SCHED_DEADLINE, as described above. Note that *RMS* has a total utilization of 95.38 %. This is interesting since it is higher than the *RT* throttling should allow, though this is probably due to numerical errors.

Looking at migrations one can see that the SCHED_RR and SCHED_FIFO migrate quite a bit, probably due to the large number of threads. SCHED_DEADLINE migrate a lot more, especially during runtime where the *RT* policies primarily migrated tasks before running them. Worth noting is that *RMS* hardly migrate tasks at all in comparison to the others. Perhaps due to that *RMS* (with its static priority on the threads) cause the system to settle on an almost static schedule on each core.

To sum up, the SCHED_DEADLINE policy has a better response time, similar computation time, lower utilization, a lot more (not a lot for *RMS*) missed deadlines and a lot more migrations than *FIFO*, *RR* and *RMS* for test case B.

### 5.1.3 Test case C

First, it is important to remember that none of the threads in any of the two versions of SCHED_DEADLINE have failed to schedule. The metrics that differs between *SCHED_DEADLINE median* and *SCHED_DEADLINE median plus* will therefore not be caused by this. In this test case the utilization and the migration numbers differs. *Median plus* has higher utilization than *median*, which is expected since it has reserved more runtime. This also explains that *median plus* have more migrations.

In this test case, the large bin problem is clearly illustrated in the response time and computation time figures. The difference now compared with before is that the wide bins are not only in the edges of the figures, they can also be found in the middle. This makes it hard to say if the computation time and the response time are better or worse for the SCHED_DEADLINE policy. It seems that the threads of type one and two are better in both

cases, but worse for type three and four. With *RMS*, it is clear that one thread type at the time is allowed to run. This is expected.

Another thing that was expected to see in test case C was that *RMS* and SCHED_DEADLINE would perform significantly better than SCHED_RR and SCHED_FIFO. Since the task set was designed to punish non-preemptive behaviour of the tasks, it was expected that especially SCHED_FIFO would put one of thread type four on each core and effectively locking out the other threads. What happened instead was that it places both of these threads on the same core, allowing thread of type three to run well at least.

Looking at the threads of type four in the computation time figures for SCHED_DEADLINE, one can see that the majority of the occurrences are at least twice as large as the value for the calibrated runtime. This is not what is desired, so it is bad, especially for *SCHED_DEADLINE median*. Since this is an exception (as mentioned above), *FIFO* and *RR* have runtimes according to the calibrated values for thread type four. This indicates that something is causing SCHED_DEADLINE to suffer an overhead. The two SCHED_DEADLINE task sets also have more migrations than all other task sets so there might be some correlation there. Threads of type three and four both have long deadlines so they will be preempted a lot, probably causing them to spend big parts of their runtime loading their data from memory. Why this behaviour is not shown as strongly in *RMS* is harder to explain however. The assumption that SCHED_DEADLINE has a much larger overhead to begin with is not represented within the rest of the test data. Only the long tasks with many migrations show this overhead. There are however, much fewer runs of threads of type four in both the SCHED_DEADLINE task sets, indicating that maybe they were preempted so much that they almost never got to run.

Another possible and perhaps more likely explanation for this phenomenon occurring only in that thread type in that task set for that policy is that the syscall_entry_sched_yield command called by the test program is not triggering a following sched_wakeup event. This causes several runs of the test program being counted as only a single slow one. The yield event is traced by the python program (used for the analysis) as a deadline miss, so the number of misses actually exceed the number of samples in the computed data, see Figure 4.63 and Figure 4.53. This is a weakness in the analysis program. Why it does not show up as clearly in the other thread types in this task set is because they do not get as many missed deadlines relative the number of runs as the threads of type four. For the faster threads there are enough samples, so the median value will be correct.

Regarding the utilization of the tasks, which thread types get most processor time differs, but the behaviour with SCHED_DEALINE getting significantly lower processor utilization can be seen in this test case too. It is around 60 % compared with around 74.5 % for *FIFO* and *RR*. One thing that stands out is that SCHED_DEADLINE (both *median* and *median plus*) have a utilization factor per thread well under 20 %, which was the goal for this test case. Still they fail to meet the deadlines. This is possibly caused by *CBS*. The tasks simply get starved when they fail to meet their deadlines, which is odd according to chapter 2.5.3 – *SCHED_DEADLINE*. If a task takes less time than the reserved time, the spare time should be added to the next period. On average over a long runtime it should do fine, which we do not see in this case. In the case of SCHED_RR and SCHED_FIFO, the utilization is at around the 20 % for thread type three and for thread type four, but less for thread one and two. This is

probably because a single run for a thread of type four covers several runs of type one and two, so they simply miss a lot of their periods altogether. *RMS* has a utilization higher than 20 % for all tasks. This is probably due to the cost associated with context switches, which is not well accounted for in the calibration. Task type four will be preempted a lot for example. This policy also performs the best regarding deadline misses, having a lower amount of misses on average than the others do.

Looking at the migration behaviour of the task sets, there are a few interesting things. Compared to the other policies, SCHED_DEADLINE seem to do more migrations during runtime, whereas the *RT* policies seem to be more prone to migrate before the task starts to run. Tasks scheduled using *RMS* sticks out. Task type one barely migrates at all, probably because this task has the highest priority. It will therefore preempt any task currently running on its *CPU*, that task in turn might have to migrate to another core. The SCHED_DEADLINE migrations seem to be more spread out, a lot like the *RMS* migrations if the tasks of type one are disregarded. For the *RT* policies, the first thread type migrates more than the second, which migrates more than the third thread type and so on. This indicates that these scheduling policies seem to evaluate the need to migrate upon a task wakeup, where SCHED_DEADLINE does so more often.

To sum up, one cannot say if the SCHED_DEADLINE policy has a better or worse response time and computation time than *FIFO*, *RR* and *RMS* for test case C. However, it does have more missed deadlines, lower utilization and more migrations than the other three for this test case.

## 5.2 Results – Ericsson's Application

As with the test runs of the test program, there are no big difference between *FIFO* and *RR*. The response time and the computation time are similar and the utilization is almost the same. When it comes to the number of migrations, *FIFO* continues to migrate more than *RR*. The worker threads migrate a lot more, simply because they are so many. For the dispatcher thread, the number of migrations is greater with *FIFO* than with *RR*, if the total number is considered. This cannot be explained, as in the test cases with the test program. It is important to notice that the computation time is much lower than the time slice used for SCHED_RR, which implies that these two scheduling algorithm should behave the same in this test too.

The missing results from the SCHED_DEADLINE policy is a discussion about the method used for assigning the policy and the thread parameters to the application. This discussion will therefore be found in 5.4.5 – *SCHED_DEADLINE with Ericsson's application*.

There are a few reasons why SCHED_DEADLINE would probably not be useful for the Ericsson *LTE* application. First, it has this built in admission control in the *CBS*. If any form of admission control is already being performed in the system this is just redundant, sometimes throwing out threads that could possibly fit in the application due to it being somewhat pessimistic.

It is also difficult to set the parameters well for several reasons. The runtime parameters are code and platform specific. This will require a recalculation for every time the code on each different platform (on which it is run) is updated, if the desire is to fully utilize the system. In

addition, for *CBS* not to interrupt tasks, the system will have to be scheduled using the *WCET* which can be costly if the runtime varies a lot. Setting the period can also be hard for tasks that are not periodic in nature. If there is an unknown time between two runs of an application, which is the case when the tasks are sporadic, the system would have to be scheduled using the shortest time between two runs that is possible. This will cause the utilization in the admission control to go up and it results in a lower system utilization than otherwise would be possible.

Finally, it is unpractical to set *CPU* affinity for SCHED_DEADLINE. It requires the use of cpusets in Linux as using an ordinary *CPU* mask in Linux will cause the scheduler to fail. One of the checks performed in SCHED_DEADLINE is that the *CPU* mask includes all of the system's available cores. This means that a cpuset needs to be created, which fools the SCHED_DEADLINE policy to believe that the available cores are the only existing threads in the system.

SCHED_DEADLINE does however show great promise with regard to response time, so if updated, it could be really useful, i.e. if the *CBS* could be disabled and setting of the *CPU* affinity could be more flexible. In its currents state, SCHED_DEALINE is not designed with the Ericsson *LTE* application in mind.

## 5.3 Results – Overall

*EDF* seems to be a great scheduling algorithm – according to the theory. An algorithm that can guarantee that all tasks in the system will meet their respective deadline, given that the Equations 2.2 and 2.3 are fulfilled. An algorithm that can have a utilization factor up to 100 % in some cases, since it is dynamic. However, as with many things, it sounds good in theory but do not work as well in practice. The problem is that SCHED_DEADLINE includes *CBS* as well as *EDF*. The fact that not all processes may be scheduled is bad. This happens when the admission control fails. Since it is performed by the *CBS*, this feature is the problem. If SCHED_DEADLINE would not include *CBS*, the policy might work better. In this case, it would not reserve the task's runtime, never perform the admission control and the tasks get to run until they are finished, if not preempted by a task with earlier deadline. In this case, the scheduler would not be dependent on the utilization of the system or the task's bandwidth for scheduling tasks. This means that if SCHED_DEADLINE worked in this way; it would be able to stack all desired processes in the ready queue. This would probably work better for the firm tasks. The SCHED_DEADLINE policy implemented in the Linux kernel today is not suitable for firm tasks. It may be suitable for hard tasks, but this is not tested in this project.

## 5.4 Method

There are various topics regarding the project methodology that will be discussed and criticized in this chapter. First out is the two different platforms used during the testing phase. Then follows a discussion about the test cases and the scheduling policies. It contains for example a motivation to the extra test runs with SCHED_DEALDINE and *RMS*. After this, a problem with the evaluation phase is gone through together with the discussion of how the

solution affects the results. The next subchapter contains mistakes and parts we would like to change if we would re-do the work, or if we had more time. Then follows a subchapter discussing the method to assign the SCHED_DEADLINE policy to Ericsson's application and why it did not work. At last, the sources used in this report are discussed.

Notable is that the methodology is chosen by us. With other test cases, platforms, solutions etc. the result may be different.

### 5.4.1 Different platforms and Ubuntu versions

The two platforms used in this thesis contained different Ubuntu versions and most importantly, two different versions of the Linux kernel. This may affect the results, since the kernel may be updated between the two versions. To check the impact of this, a virtual machine with Ubuntu 14.04.4 LTS (64-bit) was created to compare the results between U14 LTS and U16 LTS. A test was performed and it turned out that these platforms gave similar results, so there was no further investigation in this matter.

The test program was run on the platform called virtual machine due to it being more available than the platform called monster machine. The reason Ericsson's application was run on the monster machine was because it needed at least four cores, otherwise it would not be a multicore test. Two cores for the test driver and two cores for the application, using affinity.

There were two hardware threads assigned to the virtual machine. The way Windows assigns cores made it so that these two hardware threads were located on the same physical core. This has a few implications. First, both of these hardware threads will share the same L1 cache, causing migrations to often be significantly cheaper than they otherwise should be. However, since the threads are independent from each other, there should not be any cache thrashing anyway. The way that the test program is built, every other assembler instruction for most of the running program should be a *NOP* instruction. This means that the program should not be staggered more than at most a few clock cycles due to the start or the end of a period. This could account for some of the variance in the computation time of the program.

Since the two cores are virtual, some resources are shared, creating a resource constraint where ordinary cores would not have one. In a classical *CPU* architecture, one of these constraints would be having only a single ALU. However, the modern Intel processor used have several. Some resources are a constrained however, such as system calls to get the system clock.

### 5.4.2 Test cases and scheduling policies

The test cases have been performed with the three relevant scheduling policies and in some cases with *RMS* implemented with SCHED_RR. There was also an extra test run called *SCHED_DEADLINE median plus*, which used a longer runtime than the value calculated from the calibration run used in *SCHED_DEADLINE median*. Since it is usual to set the runtime larger than the *WCET* in a hard real-time system, we wanted to see how it affected the results when the runtime was set to be equal the *WCET*. This was not possible though; the *WCET* was longer than the period in some cases. The choice of giving the *SCHED_DEADLINE median plus* a 10 % longer runtime than the *SCHED_DEADLINE*

*median* was made after several tests of different percent values. 10 % was suitable since it was on the border between when the threads did and did not fail to schedule. Only a few threads did fail to schedule with that percentage.

*RMS* was used in order to see how the static priority affected the characteristics of the task set. It was also used since it is a common method to assign static priority to threads. This have been discovered during the pre-study where it commonly was used in journal papers for comparison purposes. In this project, *RMS* has been implemented using SCHED_RR rather than SCHED_FIFO, since it is the policy Ericsson are using at the moment.

It is not trivial to create the test cases. One cannot know the computation time without measuring it. This means that one cannot know the runtime, deadline and period, which is needed in order to schedule tasks with SCHED_DEADLINE. Without the python program implemented in this project this would not be possible to create the test cases, at least not easily. The same applies to the static priority of *RMS*. It cannot be set without knowledge about the threads' periods. One may know which thread is most important and in this way give it the highest priority, but this is not *RMS* scheduling.

### 5.4.3 Ignoring events

During the evaluation phase, problems with the traced Linux kernel event occurred. Sometimes there were wakeup calls occurring on already active tasks in the event list generated by Babeltrace. This means that a task first may wakeup, switch in, wakeup again before it switched out. We tried to contact the people responsible for LTTng to ask about this problem since the significance of this is not known, but without results.

There were three different solutions to this problem:

1. Ignore the second wakeup call
2. Force a switch out and a switch in when the second wakeup call occurred
3. Discard the whole set when the wakeup occurred

The first alternative was chosen. The choice was partly based on that we did not want to add data/samples and partly because much data may have been removed with the second alternative. It would most likely affect the results the least. The choice resulted in that period, computation time, response time etc. may become very large. This is the reason why the *WCET* could not be used as runtime in the test cases with SCHED_DEADLINE. Wakeup calls have been ignored when the *WCET* is greater than the period. It is also the reason to why the largest values in the histogram plots may not be fully correct. The remaining data should still be correct, especially since most occurrences are in the middle of the histogram plots. The same alternative is used for all test cases, so the comparison between the scheduling algorithms will still be valid.

### 5.4.4 Project oversights

The used time slice value for *RR* in this report was 1 ms. It was used since it was the lowest possible value that could be set in the testing platforms. According to the results, almost none of the scheduling policies in any test case have a runtime larger than the time slice. Only

thread type three and four in test case C (task T5-T8 in Figure 4.51 and 4.52) have a longer runtime. This means that *RR* never gets its characterizing behaviour. Therefore, the test runs in this project do not give a fair assessment of the *RR* scheduling algorithm. This also applies to the application provided by Ericsson. To evaluate *RR* properly a task set with longer tasks would be required, alternatively an implementation of *RR* that allows shorter time slices. This task set was not done due to lack of time. However, we should have done this if we could re-do the work.

In this paragraph, the migrations will be discussed in regards to overhead. In the platform called virtual machine it does not give a fair assessment of the overhead. This is because the hardware threads share the same L1 cache and the same hardware since they are located on the same physical core. This means that the migrations are significantly cheaper than they otherwise should be, if the hardware threads had different L1 caches. It takes almost no time at all to access a L1 cache. The results from the computation time will probably be a better metric to look at regarding overhead for the test program. If a test case using one policy have a longer computation time than another, it will have more overhead caused by scheduling decisions. With another testing platform where the hardware threads are located on different physical cores, more overhead will be generated by migrations. Overhead is nevertheless generated by more metrics than only migrations and scheduling decisions. It may therefore not be sufficient to only look at these two metrics. This is something we would have done if we had more time.

### 5.4.5 SCHED_DEADLINE with Ericsson's application

Since the application was given in a runnable file, the only way to set affinity and the task parameters were to set them to the whole application. This worked fine in the cases with SCHED_FIFO and SCHED_RR. The problem arose with SCHED_DEADLINE, since there was no way to set different task parameters (when the application was started) to different types of threads generated by the program. The idea was to first start the application with the SCHED_RR policy and then get the thread *ID* for the dispatcher thread. The second step was to manually change all threads to SCHED_DEADLINE and to the calibrated values for the worker threads in one command, before changing the dispatcher thread to its calibrated values. This method works for a small number of threads when the parameters for the dispatcher thread were set in time before the test driver returned an error message. However, when using a larger number of threads together with the calibrated parameter values it does not work to run the application with SCHED_DEADLINE. This is because the application does not receive the traffic from the test driver in time. The dispatcher thread was starved when it had the same parameters as the worker threads. This means that its jobs do not manage to finish before their deadlines and that the program crashed. This problem could not be foreseen before actually running the tests. It is a precedence relation between the dispatcher and the workers. The dispatcher needs to receive the threads before the workers can run.

To run the application with the lower amount of threads should not give a fair assessment of the program. The policy and the thread characteristics could be set for each thread in the program, but it is not a part of this project to change the code. This is why there are no results from the SCHED_DEADLINE policy.

### 5.4.6 Source criticism

There are numerous sources used in this report. Almost two-thirds of the sources are scientific sources. The remaining consist of web pages and one press release. The press release and one of the web pages was used in order to give a presentation of Ericsson, the company this thesis cooperates with. Another source (web link) is only used as reference to the open source framework called LTTng, the program used in this project for recording the events in the Linux kernel. The remaining web pages are mainly sources from GitHub. These are used in chapter 2.5 – *Linux Implementation*. There are also two other web link sources used in this chapter. The first is IEEE and the Open Group, which contains the *POSIX* standard and the second is the Linux programmer's manual maintained by Michael Kerrisk. The later should be trustworthy since there are a lot of information available about both the project and Kerrisk in person. He has even been paid by the Linux foundation, to work full time on the project.

Since the Linux operating system is an open-source software, there is no better way than to read the code and its comments when it comes to understanding the implementations of the relevant scheduling algorithms. However, it may be difficult to understand all details by only reading code, so the *POSIX* standard and the Linux programmer's manual have been used as complements. This is why the web links have been selected as sources.

Regarding the scientific sources, there are only a few secondary ones (books). The primary sources, the papers, are all from large well-known organizations.

## 5.5 The Work in a Wider Perspective

We cannot think of any ethical and societal aspects regarding the characteristic evaluation between the *EDF*, *FIFO* and *RR* scheduling algorithms.

# Chapter 6

# Conclusions

In the first chapter the purpose and the problem statement of this thesis work are stated. Throughout this report an effort has been made to see if SCHED_DEADLINE (henceforth referred to as *EDF*) scheduling has any positive aspects compared to *FIFO* and *RR*. With the results from the tests, the problem statements can be answered. The response time for the *EDF* is in general better than the current *RT* policies, but this is the only aspect in which *EDF* is better than the other two. Overhead does not seem to be significantly different for any of the three policies due to similar computation time and the fact that migrations are fast. *EDF* is therefore more suitable than the current *RT* policies regarding response time, but not regarding utilization and met deadlines. Regarding overhead, no conclusion can be made. Notably the performance of the *RT* policies can be even better with a smart way to set the static priority, for example using *RMS*. These conclusions are derived from the test program due to incomplete results from Ericsson's application with *EDF*.

One cannot say that *EDF* is more suitable than the other scheduling algorithms without knowledge about the requirements of the system. This means that we cannot say if *EDF* is more suitable than the others by only comparing their performance. A firm real-time system with high requirements on response time will benefit from the *EDF* policy even though it might miss more deadlines. However, the fact that not all processes may be scheduled with *EDF* is not good, if considering the *LTE* application. Neither is the fact that the processes are preempted if they do not manage to finish in time. These factors combined with the fact that *EDF* is not user friendly/flexible, lead us to the conclusion that *EDF* scheduling is not suitable for the Ericsson *LTE* application. Since the *CBS* feature is the main problem, the *EDF* could be suitable in the future, if *CBS* was an option that could be disabled.

# References

**Books**

Baruah, Sanjoy, Bertogna, Marko and Buttazzo, Giorgio. 2015. *Multiprocessor Scheduling for Real-Time Systems*. Switzerland: Springer.

Buttazzo, Giorgio C. 2011. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd ed. New York: Springer.

Dahlman, Erik, Parkvall, Stefan and Sköld, Johan. 2011. *4G LTE/LTE-Advanced for Mobile Broadband*. Oxford: Oxford Academic Press.

Kopetz, Hermann. 2011. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd ed. New York: Springer.

**Electronic Journal Papers**

Abeni, Luca, Lipari, Giuseppe and Lelli, Juri. 2014. Constant bandwidth server revisited. *ACM SIGBED Review* 11 (4): 19-24. doi: 10.1145/2724942.2724945.

Brun, Adrien, Guo, Chunhui and Ren, Shangping. 2015. A Note on the EDF Preemption Behavior in "Rate Monotonic Versus EDF: Judgment Day". *IEEE Embedded Systems Letters* 7(3): 89-91. doi: 10.1109/LES.2015.2452226

Dellinger, Matthew, Lindsay, Aaron, and Ravindran, Binoy. 2012. An experimental evaluation of the scalability of real-time scheduling algorithms on large-scale multicore platforms. *Journal of Experimental Algorithmics (JEA)* 17(4.3):4.3.1-4.3.22. doi: 10.1145/2133803.2345677.

Lelli, Juri, Faggioli, Dario, Cucinotta, Tommaso and Lipari, Giuseppe. 2012. An experimental comparison of different real-time schedulers on multicore systems. *Journal of System and Software* 85 (10): 2405-2416. doi: 10.1016/j.jss.2012.05.048.

Liu, Chang L and Layland, James W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)* 20 (1): 46-61. doi: 10.1145/321738.321743.

Yuan, Xin and Duan, Zhenhai. 2009. Fair Round Robin: A Low Complexity Packet Scheduler with Proportional and Worst-Case Fairness. *IEEE Transactions on Computers* 85 (3): 365-379. doi: 10.1109/TC.2008.176.

**Electronic Conference Papers**

Dellinger, Matthew, Garyali, Piyush and Ravindran, Binoy. 2011, ChronOS Linux: A Best-Effort Real-Time Multiprocessor Linux Kernel. In Leon Stok (ed.) *2011 48th*

*ACM/EDAC/IEEE Design Automation Conference (DAC)*, 474-479. DAC, 2011, New York, NY, USA. IEEE. doi: 10.1145/2024724.2024836.

Milic, Luka and Jelenkovic, Leonardo. 2014. Improving thread scheduling by thread grouping in heavily loaded many-core processor systems. In Petar Biljanovic et al. (ed.) *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 1009-1012. MIPRO, 2014, Opatija, Croatia. IEEE. doi: 10.1109/MIPRO.2014.6859716.

Stahlhofen, Andreas and Zöbel, Dieter. 2015. Linux SCHED_DEADLINE vs. MARTOP-EDF. In Eli Bozorgzadeh et al. (ed.) *2015 IEEE/IFIP 13th International Conference on Embedded and Ubiquitous Computing (EUC)*, 168-172. EUC, 2015, Porto, Portugal. IEEE. doi: 10.1109/EUC.2015.28.

**Press Releases**

Ericsson. 2016a. *Ericssons årsstämma 2016* [Ericsson's annual general meeting 2016]. http://hugin.info/1061/R/2003370/739445.pdf (Accessed 2016-04-20)

**Web Pages**

Ericsson. 2016b. *Our Portfolio*. Ericsson. http://www.ericsson.com/ourportfolio/ (Accessed 2016-04-19)

GitHub. 2013. *torvalds/linux/Documentation/scheduler/sched-design-CFS.txt*. GitHub, Inc. https://github.com/torvalds/linux/blob/097f70b3c4d84ffccca15195bdfde3a37c0a7c0f/Documentation/scheduler/sched-design-CFS.txt (Accessed 2016-02-18).

GitHub. 2015a. *torvalds/linux/kernel/sched/stop_task.c*. GitHub, Inc. https://github.com/torvalds/linux/blob/097f70b3c4d84ffccca15195bdfde3a37c0a7c0f/kernel/sched/stop_task.c (Accessed 2016-02-18).

GitHub. 2015b. *torvalds/include/linux/sched/rt.h*. GitHub, Inc. https://github.com/torvalds/linux/blob/097f70b3c4d84ffccca15195bdfde3a37c0a7c0f/include/linux/sched/rt.h (Accessed 2016-02-22).

GitHub. 2015c. *torvalds/linux/Documentation/scheduler/sched-deadline.txt*. GitHub, Inc. https://github.com/torvalds/linux/blob/097f70b3c4d84ffccca15195bdfde3a37c0a7c0f/Documentation/scheduler/sched-deadline.txt (Accessed 2016-03-22).

GitHub. 2016. *torvalds/linux/kernel/sched/deadline.c*. GitHub, Inc. https://github.com/torvalds/linux/blob/af345201ea948d0976d775958d8aa22fe5e5ba58/kernel/sched/deadline.c (Accessed 2016-02-18).

IEEE and the Open Group. 2013. *2.8.4 Process Scheduling*. The Open Group Base Specifications Issue 7; IEEE 1003.1, 2013 Edition; Copyright © 2001-2013 The IEEE and The Open Group. http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html (Accessed 2016-04-04).

Kerrisk, Michael. 2015. *Linux Programmer's Manual: SCHED (7)*. The Linux man-pages project. http://man7.org/linux/man-pages/man7/sched.7.html (Accessed 2016-03-10).

LTTng. 2014. *LTTng is an open source tracing framework for Linux*. The LTTng Project. http://lttng.org/ (Accessed 2016-03-15)

# Appendix A

# Output File from Babeltrace

```
[15:48:39.554657917] (+0.000006429) an syscall_entry_clone: { cpu_id = 0 }, { vtid
= 2194, procname = "test" }, { clone_flags = 0x3D0F00, newsp = 0x7F0C30EB6FF0,
parent_tid = 0x7F0C30EB79D0, child_tid = 0x7F0C30EB79D0 }

[15:48:39.554683910] (+0.000008385) an sched_process_fork: { cpu_id = 0 }, { vtid =
2194, procname = "test" }, { parent_comm = "test", parent_tid = 2194, parent_pid =
2194, parent_ns_inum = 4026531836, child_comm = "test", child_tid = 2196,
_vtids_length = 1, vtids = [ [0] = 2196 ], child_pid = 2194, child_ns_inum =
4026531836 }

[15:48:39.554688103] (+0.000004193) an sched_migrate_task: { cpu_id = 0 }, { vtid =
2194, procname = "test" }, { comm = "test", tid = 2196, prio = 20, orig_cpu = 0,
dest_cpu = 1 }

[15:48:39.554695091] (+0.000006988) an sched_wakeup_new: { cpu_id = 0 }, { vtid =
2194, procname = "test" }, { comm = "test", tid = 2196, prio = 20, target_cpu = 1 }

[15:48:39.555184224] (+0.000489133) an syscall_exit_clone: { cpu_id = 0 }, { vtid =
2194, procname = "test" }, { ret = 2196 }

[15:48:39.555192330] (+0.000008106) an syscall_entry_mmap: { cpu_id = 0 }, { vtid =
2194, procname = "test" }, { addr = 0x0, len = 8392704, prot = 3, flags = 131106,
fd = -1, offset = 0 }

[15:48:39.555199317] (+0.000006987) an sched_switch: { cpu_id = 1 }, { vtid = 1456,
procname = "compiz" }, { prev_comm = "compiz", prev_tid = 1456, prev_prio = 20,
prev_state = 0, next_comm = "test", next_tid = 2195, next_prio = 20 }

[15:48:39.555200435] (+0.000001118) an syscall_exit_mmap: { cpu_id = 0 }, { vtid =
2194, procname = "test" }, { ret = 139690320289792 }
```

The text lines above are a small part of a Babeltrace output file. Each segment corresponds to one line in this file. The line begins with the timestamp from when the kernel event occurred followed by the data inside the parenthesis, which is not used. After that, the name of the recorded kernel event is stated. Next, the current *CPU* is specified and then thread *ID* and process name of the current running thread are printed. Last follows a curly bracket with event specific information.

In the example above, one can see that thread 2194 receives an instruction to clone itself. Then it performs a fork to create the child thread 2196. This thread migrates from *CPU* zero to *CPU* one before it wakes up for its first time. At this point, the clone is finished and thread 2194 enters some system call. The next kernel event shows that a process named `compiz` with thread *ID* 1456 have been running on *CPU* one until this moment, while thread 2195 will be running henceforward. The system call at *CPU* zero exit at last.

# Appendix B

# Glossary

| Word | Explanation |
|---|---|
| Aperiodic | If something is aperiodic, or sporadic, it will perform a task with no known interval in between. Often triggered by a real world event. |
| Cache | A cache is a small memory placed between the main memory and the processor in order to speed up memory accesses. |
| Context switch | A context switch is performed when a processor core switches from one task to another. |
| Deadline | The deadline of a task is the time by which the task in question has to be completed in order to not waste work or causing the system to fail. |
| Dynamic | Something that is dynamic will adopt to what is happening and adopts its behaviour thereafter. |
| Non-preemptive | A non-preemptive behaviour means that a task cannot interrupt another task. |
| Periodic | If something is periodic in nature, then it will repeat itself with a specific interval in between. |
| Preemptive | A preemptive behaviour means that a task can interrupt another task. |
| Priority | The priority of a task indicates how highly the system should prioritize that specific task, a higher priority means that a task gets to run sooner. |
| Pthread | Pthreads is the way to handle threads in the *POSIX* standard. |
| SCHED_DEADLINE | SCHED_DEADLINE is the Linux implementation of *EDF*, earliest deadline first scheduling. |
| SCHED_FIFO | SCHED_FIFO is the Linux implementation of *FIFO*, first in, first out scheduling. |
| SCHED_RR | SCHED_RR is the Linux implementation of *RR*, round robin scheduling. |
| Static | Something that is static is set before hand and will behave the same every time something happens. |
| Task set | A set of tasks that has to be performed and scheduled. |

| | |
|---|---|
| Thread | A thread is a separate execution path in a program. Every independent program on a computer has its own thread. A program can also create more threads. |
| Utilization | A measurement of how well a resource is used. A high utilization means that not much of the resource is wasted. |