Universiteit Leiden
Opleiding Informatica

A Framework for Scheduling and Analysis of Real-Time Applications without the use of Worst-Case Execution Times

Name:	Frank van Smeden
Date:	17/07/2015
1st supervisor:	Todor Stefanov
2nd supervisor:	Walter Kosters

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Acknowledgments

First of all, I would like to thank Jezus Christ my savior for all He has given to me and enabling me to do this. Also I would like to thank my parents for their support during my study and their interest in my work. Also I would like to thank my friends for being there for and also for their opinions and suggestions. Of course I also want to thank my supervisors Todor Stefanov, Teddy Zhai and Mohamed Bamakhrama for giving me the opportunity and the chance to do my master thesis in the LERC group and provide me with the right amount of guidance throughout the project. I'm sure that all the things I have learned during my time in the LERC group will proof their value. I also want to thank my study advisor Jeannette de Graaf for all her advise during my study at Leiden Institute of Advanced Computer Science (LIACS). Also I would like to thank Emanuele Cannella, Jelena Spasic, Di Liu and Tsvetan Shoshkov of the LERC group for always having time for my questions and also for a good laugh (also thanks for the free coffee). Last but not least I would also like to thank my wife for trying to understand what I have been doing for the last couple of months and listening to me for hours and hours explaining about what she refers to as "real-time blah blah".

Abstract

In today's modern real-time embedded systems, there is a huge demand for high performance. Especially in the world of multimedia systems and streaming applications. While we see that multiprocessor architectures are becoming more common, we also see that scheduling these hard real-time tasks is hard. Often these schedules are not optimized for high throughput due to the guarantee of system stability, which means the system should always respond correctly in the correct time frame.

In this thesis we explore the effect of optimizing the schedule of a hard real-time system by reducing estimations based on the worst possible execution time. We also present a framework for: A) optimizing throughput for a multiple predefined types of applications and B) analysis of schedules running on both a simulator and a hardware platform.

In the end we conclude that our proposed optimization could be used to potentially increase throughput for various types of applications.

Samenvatting

Vandaag de dag is er een grote vraag naar moderne real-time embedded systems met hoge prestaties. Vooral in de wereld van multimedia systemen en streaming applicaties. Naast dat processor architecturen met meerdere kernen meer en meer gebruikelijk zijn, zien we ook dat het schedulen van deze processen in real-time applicaties een moeilijke taak is. Vaak zijn deze schedules niet geoptimaliseerd voor een hoge doorvoer binnen het systeem doordat de stabiliteit moet kunnen worden gegarandeerd. Wat betekend dat er geen deadlines gemist mogen worden.

In deze thesis verkennen we het effect van optimalisatie van een hard real-time schedule door het reduceren van de voorspelde executie tijd die gebaseerd is op de slechts mogelijk executie tijd. Ook presenteren we een framework voor het: A) Optimaliseren van de doorvoertijd voor een aantal voorgedefinieerde applicatie typen en B) Analyse van een schedule op een simulator en een hardware platform.

Aan het eind concluderen we dat onze voorgestelde optimalisatie gebruikt kan worden om de doorvoertijd van meerdere soorten type applicaties te vergroten.

Abbreviations

BCET
Best-case execution time

CPU
Central processing unit

CSDF
Cyclo-static dataflow

EDF
Earliest Deadline First

ET
Execution time

ETC
Execution time compensation

FP
Fixed priority

RM
Rate monotonic

WCET
Worst-case execution time

Contents

Chapter 1

Introduction

Modern operating systems are often complex pieces of software that consist of multiple components. One handles interrupts, another one handles drivers, networking, process management and so on. In order for modern operating systems to be able to correctly execute more then a single process simultaneously, these processes have to be managed. This is done by a component called *scheduler*. The scheduler is responsible for a variety of tasks. First of all it has to keep track of all running processes and their properties. Secondly, it has to inform the *central processing unit* (*CPU*) which process to execute. If there are multiple CPUs present, the scheduler also has to decide which process gets executed by which CPU.

Scheduling processes can be done in many ways, using different algorithms. An efficient scheduler maximizes the utilization of the available CPUs in a system to increase the throughput of the system as a whole. However, some applications demand CPU execution time by a certain frequency in order to produce correct results. For example, an audio playing application. If the application is not assigned proper CPU execution time, the playback of the audio will be interrupted and wrong/no output is produced. Such applications where time is critical to the correct functionality are called *real-time applications*. Real-time applications must be executed on a real-time system in order to be able to guarantee certain throughput of an application. Basically, a system could be categorized in two ways:

- *General purpose system*. Such a system is able to execute many different types of tasks and is flexible to perform a lot of different tasks.

- *Embedded system*. Such a system is designed for a specific purpose and can be found in devices like TVs, cars, airplanes, etc.

The majority of real-time systems are implemented on embedded devices. This is because the task of a real-time system is often very specific and does not require the flexibility introduced by a general purpose system. However, a real-time operating system (operating system that is adjusted to meet real-time requirements) could be implemented on either general purpose or embedded system.

A scheduler that takes care of scheduling tasks within a real-time system is called a real-time scheduler. A real-time scheduler can be either hard or soft. A *hard real-time scheduler* has the property that if the scheduler fails to meet the real-time requirements at even a single point in time, the system has failed. A practical example of a hard real-time scheduler is the one used in the airbag in a car. The airbag has to perform only once but if the airbag pops out too late, the system failure has lead to catastrophic consequences. *Soft real-time schedulers* are different in the sense that a single failure does not make the system as a whole a failure. A practical example is a video-streaming application. If the system fails to meet its requirement of decoding video frame at a single time, the video would be scrambled but if this happens only once in a while, someone watching the video would probably not even notice it and the system can recover which results in normal playback of the video being resumed.

Hard real-time scheduling has been an extensive research topic for the last couple of decades. A lot of different algorithms have been proposed over the years, like the *EDF* (Earliest-Deadline-First) [LL73] algorithm and the *server-based* [DB11] approach. All these hard real-time scheduling algorithms share the same goals, which are to:

- Maximize CPU utilization to create higher throughput;
- Generate as less overhead as possible;
- Keep the algorithm easy to implement;
- Keep the algorithm deterministic (for debugging purposes);

These goals have become even more challenging with today's heterogeneous Multiprocessor Systems-on-Chip architectures. For a long time, Fixed Priority Rate Monotonic scheduling has proven to be very dominant in the research field of hard real-time scheduling as stated in [But05]. This is because Rate monotonic is considered easy to implement, and easier to debug then other proposed algorithms.

1.1 Problem Description

One of the major drawbacks of scheduling hard real-time systems in general is the fact that one has to estimate the execution time of every tasks within an application. *Execution time* (ET) of a task is how long it will take to execute the task on a specific hardware platform. Hard real-time systems have to guarantee task completion with a given amount of time, otherwise it is considered a system failure. Modern hard real-time systems overestimate the execution time of the tasks within a system to make sure the system does not fail. Due to this overestimation, the throughput of task decrease which influences the throughput of the application negatively.

Since there is no single ET for a task (tasks do not always take the same amount of time, due to caches or interrupts for example) finding the worst possible ET is key for creating a Hard real-time system. the worst ET for a task is called the WCET (*worst-case execution time*) as will further described in Chapter 2. WCETs are derived mainly using static or non-static methods. The drawback that both of these approaches have in common is that they often overestimate the the actual execution time a task may have. This overestimation results in slower throughput of the system as a whole.

For example, a hard real-time system may have a task that, in 80% of all the cases, needs 20 time units to complete. 10% of all the cases it needs 25 units and in the last 10% it needs 15 units. In order to guarantee the schedulability of the task, hard real-time systems always have to assume the worst-case and must thereforee schedule the task in such a way that the task will always need 25 time units available for execution. This means that in 80% of all cases, 5 time units are overestimated and in 10% there are 10 units overestimated. The amount of overestimation is very application dependent and can differ highly among different applications. Why the ET of a task can differ for each firing can have multiple reasons. It could be due to the fact that the CPU is interrupted to handle different tasks, but it could also be normal application specific behavior.

1.2 Contribution

In this thesis we will present a way to overcome some of the issues stated in Section 1.1. We will introduce a method to analyze an application to come up with higher throughput on a real-time Fixed Priority rate monotonic scheduler. We obtain this higher throughput by trying to reduce the overestimation. This overestimation is introduced by guaranteeing no deadline will be missed by using ETs that are based on the WCET. Significant reduction of the overestimation in a hard real-time system could optimize the utilization and could therefore increase the throughput.

Also, we contribute a framework that monitors and analyses the execution of a predefined schedule and generate statistics based on the execution of that schedule. For this thesis we created both a real-time system implementation on the STM32F4 Discovery board hardware [dis] and a software simulator which involves an extension of the $Daedalus^{RT}$ framework [dae14]. The hardware implementation is based on Erika Enterprise operating system [eri14]. In our framework we have chosen for Erika Enterprise because it is an operating system with much flexibility and supports multiple scheduling algorithms and it supports the STM32F4 discovery board.

1.3 Scope of Work

In this section, we list all the assumptions and restriction to our work presented in this thesis.

- Multimedia applications
  We mainly focus on multimedia applications. This means that our applications are data-flow dominated and could be modelled using CSDF graphs.

- Scheduling algorithm
  We use Rate Monotonic as a scheduling algorithm both on the hardware platform and in the simulator. However, our proposed framework is capable of also handling different scheduling algorithm. Any limitations for using a different scheduler are based on the availability of the algorithm on the operating system used (Erika Enterprise) and the $Daedalus^{RT}$ framework.

- Single processor
  In this thesis we focus on using a single processor for our applications.

1.4 Related Work

Optimizing the utilization by trying to reduce the estimated time for a task has been tried before. Zhao et al. [ZKW+04] presented their method in 2004. They proposed a method for optimizing the estimated WCET and tune it to become more like the actual WCET. They present the first compiler that interacts with a timing analyzer. This timing analyzer calculates WCET predictions during the compilation of applications to come up with better estimations. This timing analyzer uses static analysis of the code by calculating the WCET for each function and loop in the program. The outcome of this paper is that their timing analyzer, together with an interactive compiler, is capable of estimating the WCET more accurately. Our approach is different from this in the sense that we take into account that there could be multiple ETs per task. This allows us to create a more fine grained optimization. Also, their proposed method is compile time and our method uses run-time analysis.

Like us, Sung-Soo Lim et al. present in [LKM98] a technique to analyze the WCETs. They use a technique called "extended timing schema" and try to apply these schemas to optimized machine code in order to more accurately estimate the WCET. One of the main issues that they try to overcome with their compiler-based approach, is the lack of correspondences in the control structure between the optimized machine code to be analyzed and the original source program written in a high-level programming language. However their approach suffers from what they call the "unfeasible path problem". Their compiler is unable to eliminate some of the paths of the application logic that are unfeasible. This results in a overestimation for applications with multiple executions paths. Since our approach is not compile time but run-time, we do not have these issues. However our approach can still benefit from approaches like these to generate more accurate WCETs at forehand as described in Section 2.1

In [BCPt02] Bernat et al. present another approach to reduce the WCET by doing probabilistic static analysis. According to their work, their approach is capable of strongly reduce the overestimation produced by traditional approaches. Basically they estimate the ET by profiling how long a certain code-block takes taking into account the probability of having cache misses or hits. Opposite of what Zhao et al. did, Bernat et al. do take into account multiple possible ETs for same blocks of code.

Other research is done by M. Petters et al. They focus in their paper [PF99] more on measurements and less on modeling to make proper WCET estimations. They use a compiler to generate an intermediate flow control graph and run measurements based on this graph. Our approach is similar in the sense that we also make our predictions mainly on measurements. However, the intermediate graphs they use are a simplification of all the code branches within a system. Therefore they do not consider the fact that some branches are shortest in terms of ET then others. This will give one a good estimation of the WCET, but does not consider overestimation in case the short branch is being executed. Our approach takes multiple ET per code branch into account and uses this to improve the throughput of the system as a whole.

1.5 Structure of the Thesis

The first Chapter of the thesis is an introduction and covers the problem description in Section 1.1 and the contribution of my thesis can be found in Section 1.2 Section 1.3 presents the scope of this thesis and covers the boundaries of the work. Section 1.4 describes other work related to this thesis and in which way this thesis covers a different approach. Chapter 2 contains sections about different background topics like WCET, repetition vector, and periods. In Chapter 3, our framework is proposed. One will also find a brief description about how to use the framework in Section 3.8. In Chapter 4 we present the experiments done during this project and the outcome. Chapter 5 discusses the findings based on the results and remaining issues related to this thesis.

Chapter 2

Background

In this section we explain more about the context of the proposed framework and the basic concepts of (hard) real-time systems. The following subsections are necessary to understand the subsequent chapters. They cover definitions like *CSDF graphs*, *worst-case execution time*, *periods* and more hard real-time related topics.

2.1 Worst-case execution time

While designing a real time application, time is an important aspect. In order to make proper scheduling calculations, the time a task takes to execute a certain amount of code should be known. Two definitions that relate to the time a task takes within a system are the *WCET* and the *BCET*, being the *worst-case execution time* and *best-case execution time*, respectively.

Definition 1. *(Worst-case execution time [WKRP05]) the longest time it takes to execute a given program code.*

Definition 2. *(Best-case execution time [WEE+08]) the shortest time it takes to execute a given program code.*

As explained by Reinhard Wilhelm et al. [WEE+08] , Figure 2.1 shows a basic notion of timing analysis of systems. Here one can see that the distribution of different ETs can differ among different executions of the application. Please note that this figure is an example and the real ETs depend highly on the application. The lower curve represents a subset of measured executions. Its minimum and maximum are the minimal observed ETs and maximal observed ETs, respectively. The darker curve represents the times of all executions. Its minimum and maximum are the best-case and worst-case execution times, respectively.

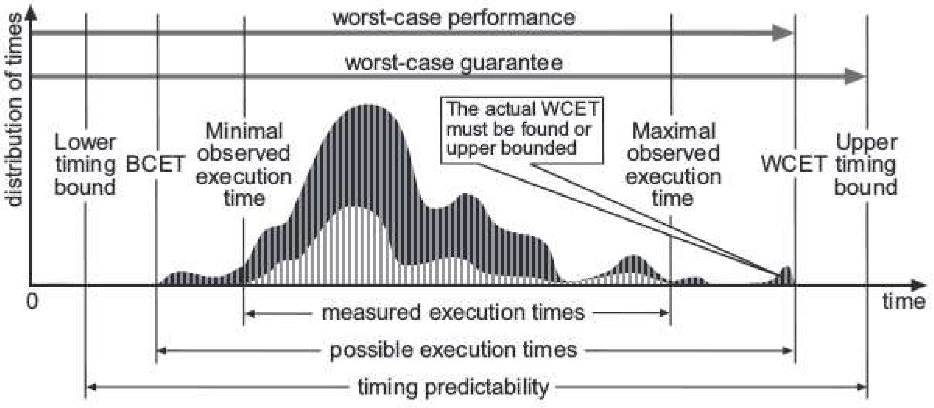

Figure 2.1: Timing analysis of systems.(Figure by Reinhard Wilhelm et al. [WEE+08]).

A general way to extract the exact WCET of a task has proven to be impossible since it is equivalent to the *halting problem* proposed by A. Turing [Tur36]. In his proposal, Turing tries to decide whether the given program will ever halt on a particular input. Unfortunately, at the time of writing, this problem remains undecidable. Therefore, despite frameworks on the market today try to not derive the exact WCET of a task but an approximation of the WCET instead.

There are basically two approaches to derive the WCET of an application or task within an application. In the first approach, a *static analysis* of the code implementing a task or application is performed. This means one has to count the assembler instructions for each function, loop etc. and combining them to compute the ET of that particular piece of code. One drawback of this method is that it can not account for other factors that only occur at run-time, like interrupts or the use of caches. Interrupts and the use of caches are extremely hard to predict at design time. Static analysis overcomes these issues by assuming the worst possible execution time, whether this will happen or not.

The second approach is *hardware profiling*, also called "end to end" measuring. One has to run the application for a long period of time and measure the actual ETs performed by the hardware platform. This method as stated by [WKRP05] tends to be very error-prone and has the drawback of begin unable to keep up with the rapidly increasing complexity of modern systems. As stated by [WEE+08] this method "will in general overestimate the BCET and underestimate the WCET". This type of measuring can also be done by simulation. But as stated by [DBK01] this is not very reliable since not all simulators use clock-cycle accurate models which can lead to very inaccurate results.

2.2 CSDF

In our framework we use Cyclo-Static Dataflow (CSDF) graphs [BELP96]. These graphs are an extension of the Synchronous Dataflow (SDF) graphs presented by [LM87].

CSDF allows us to model an application as a set of actors. A CSDF graph as defined by [BELP96] as $G = (A, E)$ where A is a set of actors that are represented in the graph by the nodes and $E \subseteq A \times A$ is a set of communication channels represented by the graph's edges.

An actor represents one or multiple statements in the program that have to be completed to transform incoming data stream(s) into outgoing data stream(s). In a CSDF graph the information on an edge indicates the production and consumption rate of the adjacent actors. The input and output produced or consumed by an actor is related to the execution sequence of the actor. For example, if the production rate of an actors is $[0, 1]$, the actors will produce the amount of output indicated by the first item (being 0) during the first execution of the actor. The next execution the actor will produce the amount of output presented in the second item of the sequence and so on. The same holds for the consumption rate.

Communication channels use the FIFO mechanism and are bounded in size so they are limited in the amount of output data from the node they can hold. A single instance of output created by an actor is referred to as a *token*. A communication channel is defined as $E_u = (A_i, A_j)$ where A_i is called *source actor* and A_j is called *destination actor*.

A CSDF graph can be either *connected* or *unconnected*. If there is a path from any actor in the graph to any other actor in the graph, it is called *connected* otherwise it is called *unconnected*.

We define the successor actor and predecessor actor as follows:

$$Succ(A_i) = A_j \in A : \exists E_u = (A_i, A_j \in E) \tag{2.1}$$

$$Prec(A_i) = A_j \in A : \exists E_u = (A_j, A_i \in E) \tag{2.2}$$

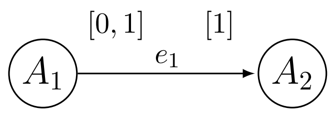

Figure 2.2: CSDF graph example, Actor A_1 produces a single token after two firings, actors A_2 consumes a token each firing.

In Figure 2.2, an example of a CSDF graph is given. It consists of two actors, A_1 and A_2. In this graph, actor A_2 is the successor of actor A_1 indicated by the arrow. The production rate of actor A_1 is $[0, 1]$, meaning it will create a single token on the second firing. No output will be produced the first time the actor is fired. Also, actor A_1 has a phase of 2, this means that the production rate has 2 possibilities (being 0 and 1) before it will start repeating the same sequences from the start.

The consumption rate of actor A_2 is $[1]$, meaning it will consume a single token from the FIFO each time this actor fires. An actor is called *sink-actor* or *sink-node* if the actor is the last actor in a chain of actors and generates output for the application instead of creating input for other actors.

2.3 Repetition Vector

One of the most important aspects of a real-time system is the schedule. A schedule is being generated by the scheduler based on a scheduling algorithm and a predefined taskset. In order to make sure a schedule of a hard real-time system will not end up in a situation where deadlines will be missed, one has to make sure a*valid static schedule* will be generated.

Definition 3. *(Valid static schedule [BELP96]) Given a connected CSDF graph G, a valid static schedule for G is a finite sequence of actors invocations that can be repeated infinitely on the incoming sample stream while the amount of data in the buffers remains bounded.*

Within a hard real-time system where no deadline misses should occur, the system does not have to handle deadline misses since it assumes it can derive a valid static schedule based on the given taskset. If the scheduler is not able to derive a valid static schedule the behavior of the system is undeterministic but will likely end up in an unrecoverable state. In a soft real-time system, deadline misses can occur so the system should be able to handle them and should not become unrecoverable after a deadline miss.

To determine a valid static schedule for a CSDF graph one has to derive the amount of invocations of each actor and the correct sequence of these invocations. This is done by computing the *repetition vector*:

Definition 4. *(Repetition Vector [BS14]) A vector $\vec{q} = [q_1, q_2, ..., q_n]^T$ where $q_j > 0$ and n is the amount of actors in the CSDF graph G. This vector is a repetition vector if each q_j represents the number of invocations of an actor A_j in a valid static schedule for G*

According to [BS14], the repetition vector $\vec{q}$ can be computed by

$$\vec{q} = \Theta \cdot \vec{r} \tag{2.3}$$

Where:

$$\Theta_{jk} = \begin{cases} N_j & \text{if j = k} \\ 0 & \text{otherwise} \end{cases}$$

and where $\vec{r}$ is calculated according to the following balance equation:

$$\Gamma \cdot \vec{r} = \vec{0} \tag{2.4}$$

and where Γ is the topology matrix of G representing the production and consumption rate of the communication channels:

$$\Gamma_{uj} = \begin{cases} X_j^u(N_j) & \text{if actor } A_j \text{ products from channel } E_u \\ -Y_j^u(N_j) & \text{if actor } A_j \text{ consumes from channel } E_u \\ 0 & \text{otherwise} \end{cases}$$

Where N_j, is the length of the production/consumption rate vector.

Taking the graph presented in figure 2.2 as an example G, the needed invocations of actor A_1 and A_2 are 2 and 1 respectively.

$$\Gamma = \begin{bmatrix} 1 & -1 \end{bmatrix}, \vec{r} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \Theta = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \vec{q} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \tag{2.5}$$

2.4 Periods

One of the first steps in our approach is to calculate the *period*. Here follows a definition.

Definition 5. *[BS14] (Period) For a CSDF graph G, a period vector $\vec{P}$, where $\vec{P} \in \mathbb{N}^{|\mathbb{A}|}$, represents the periods, measured in time-units, of the actors in G. $P_j \in \vec{P}$ is the period of actor $A_j \in A$. $\vec{P}$ is given by the solution to both:*

$$q_1P_1 = q_2P_2 = ... = q_{n-1}P_{n-1} = q_nP_n \tag{2.6}$$

and

$$\vec{P} - \vec{C} \geq \vec{0} \tag{2.7}$$

Where $\vec{C}$ is called the *execution time vector* as defined by the following definition:

Definition 6. *[BS14] (execution time vector) For a graph $G = (V, E)$, an execution time vector $\vec{C}$, where $\vec{C} \in \mathbb{N}^{|\mathbb{A}|}$, represents the worst-case execution times measured in time units of the actors in G.*

From (2.6) and (2.7), one can derive the period vector ($\breve{P}$) of a CSDF graph by the following formula:

$$\breve{P}_i = \frac{lcm(\vec{q})}{q_i} \lceil \frac{\hat{W}}{lcm(\vec{q})} \rceil \forall A_i \in A \tag{2.8}$$

where $lcm(\vec{q})$ is the *least common multiple* of the repetition vector, and $\hat{W}$ is the maximum workload defined as $\hat{W} = max_{A_i \in A}\{W_i\}$. The workload of an actor is $W_i = q_iC_i$ where C_i is the WCET and q_i the entry in the repetition vector.

2.5 Start Times

Each actor within a real-time system has a start time. For the start time of an actor, we use S_i to refer to the time at which an actor A_i can start executing. The definition is as follows:

Definition 7. *(Start Time [BS14]) Let $E_u = (A_i, A_j)$ be a communication channel in graph G. Under a periodic schedule, a valid start time of A_j, denoted by S_j,guarantees that A_j finds enough data in E_u to fire at time instants $S_j + kP_j$ for all $k \in \mathbb{N}_0$*

Definition 7 implies that A_j will never block while reading from E_u i.e. no *buffer underflows* will occur. To enforce this, actors that depend on incoming data from their predecessor, are scheduled after the start time of their predecessor plus their predecessors period.

Once the start times are computed they are passed to the hard real-time scheduler together with the periods (see chapter 2.7 for more details) The scheduler will create release times based on the start times and periods for each actor/task. A release time is the exact point in time a task is ready to be schedulable but this does not mean that it will be scheduled at that exact time. It could be that some other task will be scheduled first because of prioritization of the scheduler. After the first release time of a task, the period will indicate the point in time the tasks will be schedulable again for execution. More on prioritization on a fixed priority schedule algorithm in Section 3.7.2.

2.6 Buffer Sizes

As described in Section 2.2, communication channels define the production rate of the output of an actor. Buffers are the actual containers of the data once the output is produced. The size of these buffers is important because it allows the scheduler to fire an actor multiple times without the need for the successor actor to be fired directly after.

Definition 8. *(Valid Buffer Size [BS14]) Let $E_u = (A_i, A_j)$ be a communication channel in graph G. Under a periodic schedule, a valid buffer size of E_u, denoted by b_u, guarantees that A_i can store tokens to E_u at time instants $S_i + kP_i \forall k \in \mathbb{N}_0$.*

When calculating the buffer sizes, one should assume that when an actor is executed that is dependent on input from another actor, the corresponding buffer contains the amount of tokens that the actor must consume. Otherwise, the actor may not be able to consume the tokens and the system will run into a buffer underflow. Also, one should also assume that an actor produces output at the very end of its execution so that other actors depending on its output will not be released and try to consume data from the buffer too quickly.

2.7 Scheduling

In our proposed framework we use the *Rate Monotonic* (RM [LL73]) scheduling algorithm. RM is categorized as a *fixed priority* (FP) scheduling algorithm which means that priorities for all the tasks have to be known at the start of the scheduling and do not change throughout the execution of the algorithm. The assignment of the priorities is based on the length of the periods. The smaller the period, the higher the priority.

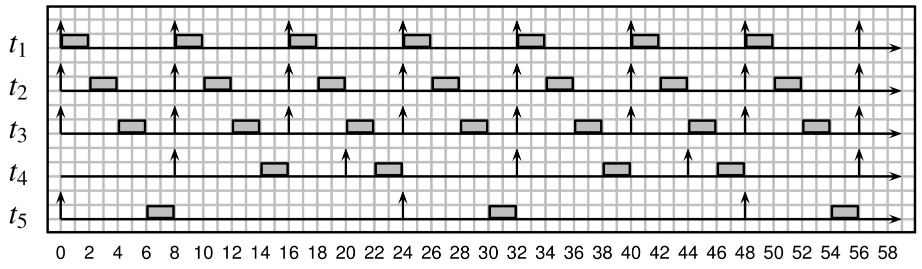

Figure 2.3: Example of a schedule.

Figure 2.3 shows an example of a RM schedule. In this schedule there are 5 tasks denotes $t_1, t_2 ... t_5$. All these tasks combined are called a *taskset*. Each gray block represents the WCET assigned to a particular *job* which is an instance of that particular task. In this example, the priorities of the tasks are assigned according to the number of the task with t_1 having the highest priority. The arrows indicate the task's *release time*. The release time of a task is the exact point in time a job of that task is ready to be scheduled by the operating system. Each job in a schedule has a *deadline*. The deadline is the point in time when the job must have finished execution. In our scheduling implementation, the deadline of a job is the next release time of the task. If a job has not finished executing by its deadline, a *deadline miss* occurs. For a hard real-time system, this results in a failure of the system. The space between two release times of a task (indicated in Figure 2.3 with the arrows) is the period of a task (see section: 2.4).

One of the properties of the RM scheduling is that when a new job is released that has a higher priority then the job that is currently being executed, the current job, will be postponed and the job with the higher priority will start execution. Postponing a job in favor of another job is called *preemption*. Preemption will cause a context-switch in the operating systems because the execution context for different tasks are different. Context-switching within operating systems are time-consuming and should be done as less as possible to reduce the overhead of the system. Preemption is more likely to occur on systems were the processor is more utilized (See section 2.7.1 for more about utilization).

Figure 2.4 shows an example of preemption. Task t_5 is preempted twice in favor of t_1 because the priority of t_5 is lower. Also one can see that no deadline is missed even though t_5 was preempted, indicated by the two white blocks of task t_5. This is because the period of t_5 is long enough to cope with the preemption. Also, as shown in Figure 2.4, the release time (indicated with the arrow for each task) is the same for t_1, t_2 and t_3. The FP scheduling algorithm makes sure that in this case the task with the highest priority is executed.

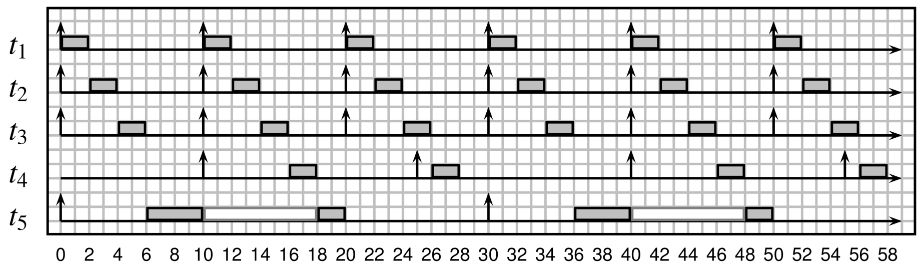

Figure 2.4: Example of a schedule with preemption.

2.7.1 Utilization

One other important aspect of scheduling is the utilization of the available CPUs in a system. The utilization reflects the time a CPU is execution a task. to define the utilization of a CPU we use a range from 0...1 where 0 means the CPU is not being used at all and 1 means the CPU is working at its full capacity. The utilization of a real-time system depends on multiple factors. First there are the amount of processors. On a system with 2 CPUs the maximum utilization is 2. This is equal to the amount of processors in a system and is called the *upper bound utilization*. Secondly, the utilization also depends on the paralyzability of the taskset. For example, if a taskset contains 4 tasks which are all independent on an other task, each task can be executed on a separate CPU. This means that the utilization of the system is 4. However if the tasks are dependent of each other, the utilization could become less since some task(s) have to wait for other task(s) to finish execution first.

In order to improve the speed of execution on a system, Schedulers generally try to keep the utilization as high as possible. This means one has to remove as much *slack* as possible. Slack is the amount of time in a schedule that the CPU is in an idle state and not executing any job.

Chapter 3

Hard Real-time Testing Framework

In this chapter, we will present a real-time testing framework. The framework's main feature is measuring the impact on the scheduler while running different schedules. Our framework tries to find an optimal period for all tasks in the system by running the scheduler and monitor the amount of deadline misses that occur during the execution of A taskset. Our approach is to reduce the periods of the tasks within a taskset. We accomplish this by not using the WCET while deriving the period but a lower value. The impact of this reduction is measures in the amount of deadlines misses of the sink-node in our CSDF graph. This is because the application as a whole only generates output at the time the sink-node has been executed. So if deadlines are missed within the system because of the reduction, but the sink node is still not missing any deadlines, the throughput is still being generated at a higher rate then if the system uses the WCET. However, in these conditions we cannot say it is still a hard real-time system anymore because of the missed deadlines. But since we can still guarantee certain throughput for the system as a whole we can still call it a real-time system.

3.1 Functionality

The framework is divided into two parts, one is for running on hardware and the other part is the simulator. To conduct our experiments we have used a hardware platform based on an ARM processor called the STM32F4 Discovery board. On this board, we use the Erika Enterprise operating system [eri14] to run a fixed priority rate monotonic schedule.

Next to running on a real hardware platform we also use a simulator to simulate the FP RM algorithm. This is because hard real-time operating systems often only assume a valid static schedule (see Section 2.4). But in our experiments this will not always be the case because reduction of the value taken as the WCET will decrease the size of the periods. This can cause the schedule to become invalid. If the total amount of time needed for a task to finish a single execution is more than the period for that task, the task will result in a deadline miss and according to the theory of hard real-time systems, the schedule will not invalid. Hard real-time systems that are supplied with an invalid static schedule can become unstable due to deadline misses or may not run at all. Because deadline misses should never happen, hard real-time schedulers often cannot recover with a deadline miss. The hard real-time operating system, we used on our hardware platform (Erika Enterprise), could also not handle a deadline miss. This is why we decided to use a simulator to be able to adjust this behavior and be able to define how the system should cope with deadline misses. More details about how we ensured proper execution after deadline misses is given in Section 3.7.2. The simulator we used is a modified version of the simulator in the $Daedalus^{RT}$ framework [dae14]. In Figure 3.1 one can see a schematic overview of the proposed framework. The input for the framework consists of two parts. The framework we propose needs a CSDF graph as input. The format of this graph is specified by the$Daedalus^{RT}$ framework. The needed dataset is explained later in Section 3.3.

The framework consists of a pipeline of features that are executed after each other. The next items shows each of those stages:

- CSDF Graph
  This a representation of a graph in text file. It contains information like the names of the nodes, the production and consumption rate of each of the the actors and the WCET of all the actors.

- Dataset
  This is a file that presents all execution times and how often they happen for each actor that is specified in the CSDF graph.

- Computing scheduling data
  Based on the supplied CSDF graph and the dataset scheduling data generated. The scheduling data is contains the following information;

  - Periods for each of the tasks in the CSDF graph;
  - Start times for each of the tasks in the CSDF graph;

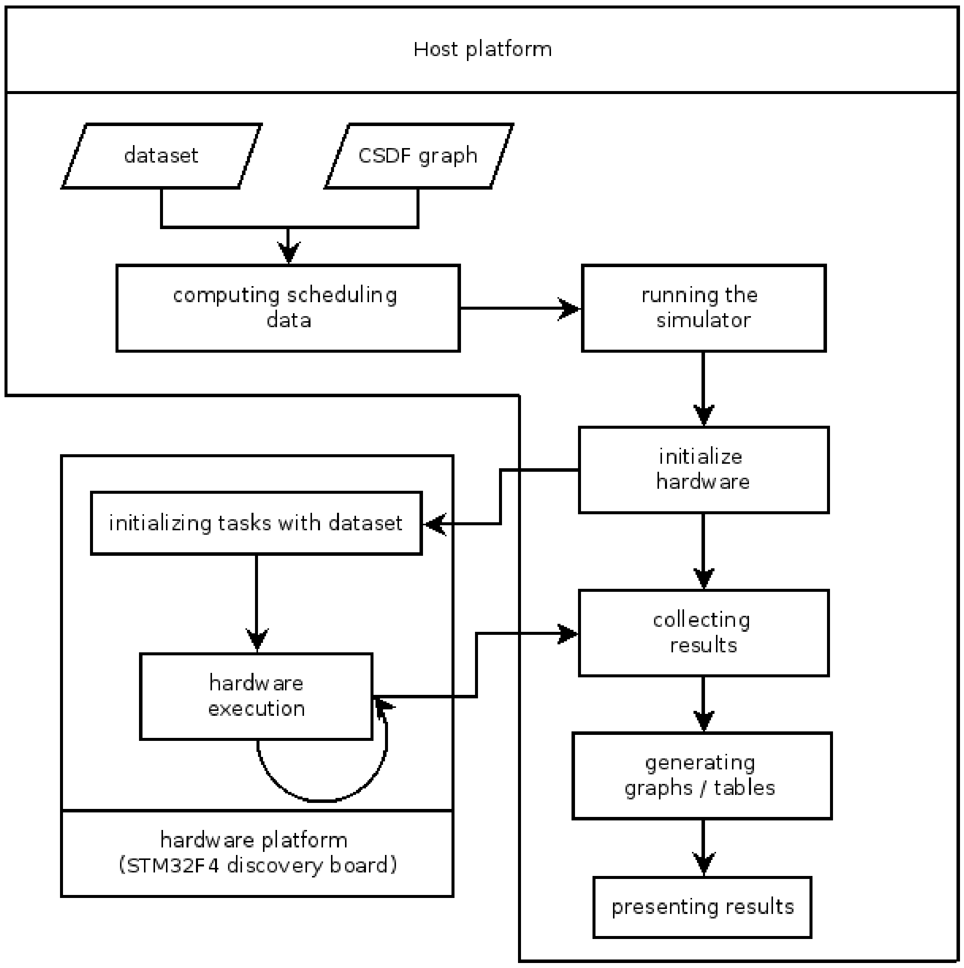
Host platform
dataset
CSDF graph
computing scheduling data
running the simulator
initialize hardware
initializing tasks with dataset
hardware execution
collecting results
generating graphs / tables
presenting results
hardware platform (STM32F4 discovery board)

Figure 3.1: Schematic overview of the framework, each box represent a separate stage within the framework.

  - Buffer sizes for each of the communication channels in the CSDF graph

  This scheduling data is generated multiple times for each dataset. Once by taking the WCET to compute the scheduling data and after that again while we gradually decrease the value we take for the WCET by 5%.

- Running the simulator
  After we have generated the scheduling data, we run the simulation

based on the scheduling data. The simulator is a python framework that simulates a hard real-time operating system and contains a FP RM scheduler. While running, the simulator generates output that is saved for analysis later on.

- Initialize hardware
  After the simulation we initialize a hardware platform to run with the same datasets as the simulator.

- Initializing tasks
  As with the simulation, we supply the hard real-time operating system with the scheduling data and set the periods, start times and buffer sizes to the correct values.

- Hardware execution
  We let the hardware platform run. While running the hardware platform generates output that is collected by the host platform.

- Collecting result
  After both running on the hardware platform and simulation are done, we parse the result and save them in databases.

- Generating graphs / tables
  Based on the results we generate new tables and generate graphs based on the generated databases

- Presenting results.
  We store the tables and derived results in a report.

3.2 Deriving Execution Times

The first step in our approach is to construct a CSDF graph modeling an application and give this CSDF as input to our framework. This is part of the "computing scheduling data" stage shown in Figure 3.1 For our testing framework we use the CSDF graph G showed in Figure 3.2 as an example. This graph represents a simple image manipulation application and it contains 5 actors and 5 edges.

The next step is to obtain the ETs for each of the actors within G. Each actor may have multiple ETs. All the ETs for a single actor are represented by vector $\vec{D}$, being the *actor execution time vector*. $\vec{D}$ always has to be ordered from high to low. In our framework we assume one obtains $\vec{D}$ by

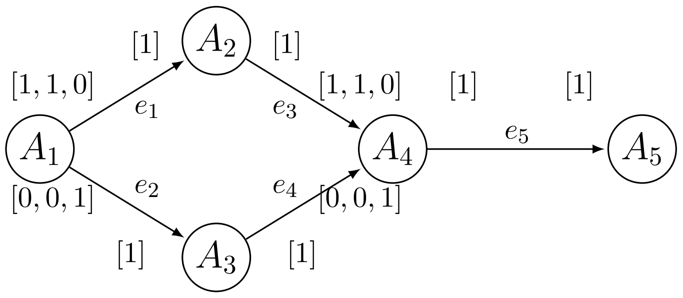

Figure 3.2: CSDF graph containing 5 actors and 5 edges.

either one of the two methods described in Section 2.1 being static analysis or hardware profiling.

As an example, Figure 3.3 shows the occurrence of each of the ETs of an actor A_1. As one can see, there are 8 different ETs all in the range from 10 to 40 time units for a total of 50 firings for actor A_1.

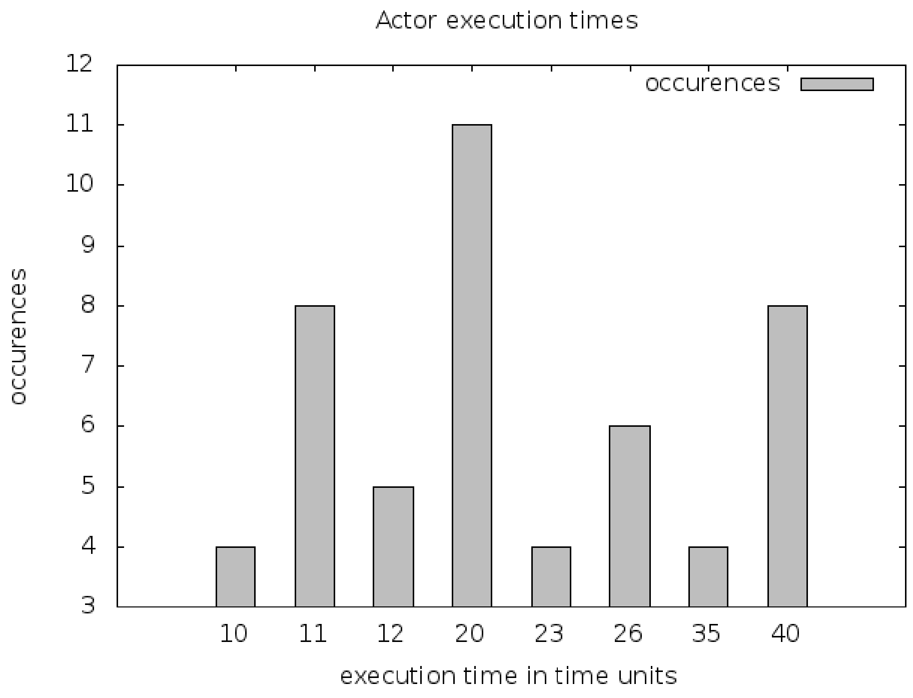

Figure 3.3: Example actor execution times.

execution time	occurrences	% occurrences out of 50
10	4	$4/50 = 8\%$
11	8	$8/50 = 16\%$
20	$11 + 5$	$16/50 = 32\%$
26	$6 + 4$	$10/50 = 20\%$
40	$8 + 4$	$12/50 = 24\%$

Table 3.1: Dataset of an actor. In this table one can see 3 occurrences of execution-time mapping

Like the WCET, the amount of different ETs for an actor is hard to predict. Mechanisms like caches and interrupts can also result in different execution times.

Our approach assumes that $\vec{D}$ contains all possible ETs that were the result of either static analysis or hardware profiling. However, because this can be a very large vector, we introduce a shorter fixed vector representing the *actor execution time vector* $\vec{D}$ as $\vec{D}'$. This shorter fixed vector contains the WCET, BCET and the n-most occurring execution times. The larger value one will take for n, the more accurate to the $\vec{D}$ it will be. In our approach we take $n = 3$ for convenience reasons. Regarding Figure 3.3, one can represent vector $\vec{D} = (10, 11, 12, 20, 23, 26, 35, 40)$ as a shorter fixed vector $(40, 26, 20, 11, 10)$ where 40 is the WCET and 10 is the BCET.

The next step is to obtain the corresponding occurrences for each ET in the shorter fixed representation of $\vec{D}$. To do this, we have to round each ET in $\vec{D}$ up to the nearest ET present in $\vec{D}'$.

For example, as one can see in Figure 3.3, there are 5 occurrences of ET 12. But because 12 is not present in our short execution-time-vector we map these 5 occurrences to another execution time that is within the exection-time-vector but only to one which has a higher execution time then 12 which is in this example 20. We call this *execution-time mapping*.

Considering the results from Figure 3.3 and the approach stated above we construct Table 3.1. The first column shows the execution times present in $\vec{D}'$. The second column shows the occurrences of a particular ET. If this column contains a summation of multiple integers, they indicate the event of *execution-time mapping*. The third column shows the occurrences out of the total amount of executions of the task. In order to obtain the execution times for all actors within G, one has to repeat this method for all actors within the CSDF graph.

3.3 Creating Datasets

As stated earlier in Section 3.2, the ET of a tasks differs for each application.

Some applications can have a large amount of values for $\vec{D}$ and other application can have a very small value for $\vec{D}$. This influences the the amount of possible deadlines misses and how much reduction of the periods is possible. To analyze how our approach is affected by multiple types of application, our framework mimics various application characteristics by using different amounts of occurrences of ETs for an actor.

A *dataset* represents ETs and occurrences for an actor in a CSDF graph. It holds the information present in column 1 and 3 in Figure 3.1 for each task. Also it contains the period, start times and buffer sizes for each actor.

Because we mimic application to test our framework on different types of applications, we have to determine the ET. In a non-experimental setting, this would be done by static analysis or hardware profiling as described in Section 2.1. For our application we assume $BCET = \frac{WCET}{2}$, and all the other execution times have to be mapped evenly in between by using execution-time mapping. We define Δ as the difference between two adjacent execution times in $\vec{D}'$

$$\Delta = \frac{\left(\frac{WCET}{2}\right)}{|\vec{D}'| - 1} \tag{3.1}$$

Our framework uses 9 datasets. These are represented in Table 3.2 and Table 3.3 In these tables, one can see the occurrences of ETs available in the short execution time vector.

Each of these datasets represent a certain characteristic behavior that a task can have.

- Dataset 1
  Here all the times this task gets executed, the task will have the same ET. We took 20 as the ET here.

- Dataset 2
  Tasks that often execute between the WCET and BCET are represented in this dataset.

- Dataset 3
  A more extreme case of dataset 2, the chances of executing x2 are more likely here.

- Dataset 4
  Tasks that more often execute the BCET evenly distributed chance of executing any other ET.
- Dataset 5
  A more extreme case of dataset 4.
- Dataset 6
  Tasks that more often execute the WCET evenly distributed chance of executing any other ET. This is exactly the opposite of dataset 4
- Dataset 7
  A more extreme case of dataset 6.
- Dataset 8
  Tasks that are either likely to execute the WCET or BCET but less likely to execute something in between.
- Dataset 9
  A more extreme case of dataset 8.

One can see in dataset 2 of Table 3.2 that in 10% of the cases, the WCET is used as ET. Also, in 60% of all the cases the x2 is used, which for a WCET of 80, means $2 \cdot \Delta$. Equation 3.2 shows how to derive Δ based using $|\vec{D}'| = 5$ and $WCET = 80$.

$$2 \cdot \Delta = \cdot \frac{\left(\frac{80}{2}\right)}{|5| - 1} = 20 \tag{3.2}$$

In Table 3.4 One can see the actual values used in dataset 2.

For the WCETs of our actors in our presented CSDF graph G we take $[50, 80, 240, 40, 40]$ for actors A_1 till A_5 respectively. These are similar to the WCETs proposed in [BS14]. Based on Equation 3.1, we can construct all the datasets for all actors in G. Details of dataset 2 are presented in Table 3.4.

3.4 Calculating Periods

Recall from Section 2.4 how periods of actors are computed using the $Daedalus^{RT}$ framework. Our framework recalculates the periods by gradually decreasing the value of the WCET by 5%. This will decrease the length of the periods since these are based on the this value. Decreasing the length of the periods

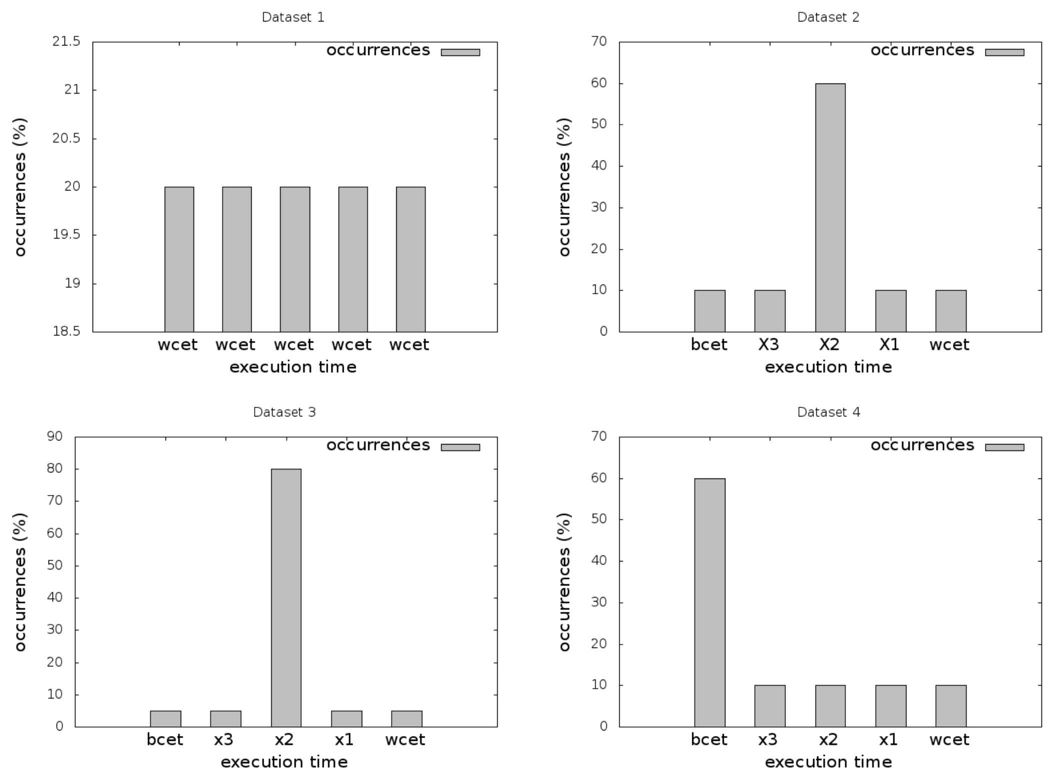

Table 3.2: Datasets containing $\vec{D}'$ and the percentage of occurrences for single actor within CSDF graph G (part A)

within a taskset will increase the throughput since the CPU will have less time assigned for each task to finish execution. However this can also result in creating more deadline misses if the periods are becoming too small to finish execution of a task.

In our framework we use the following steps to calculate the periods for a single dataset:

1. Take the WCETs from a dataset.
2. Calculate the periods of all actors in the CSDF graph with the $Daedalus^{RT}$ framework based on the WCET.
3. Reduce the period of each of the actors in the CSDF graph by 5%
4. Recalculate the period and start times of the tasks using the $Daedalus^{RT}$

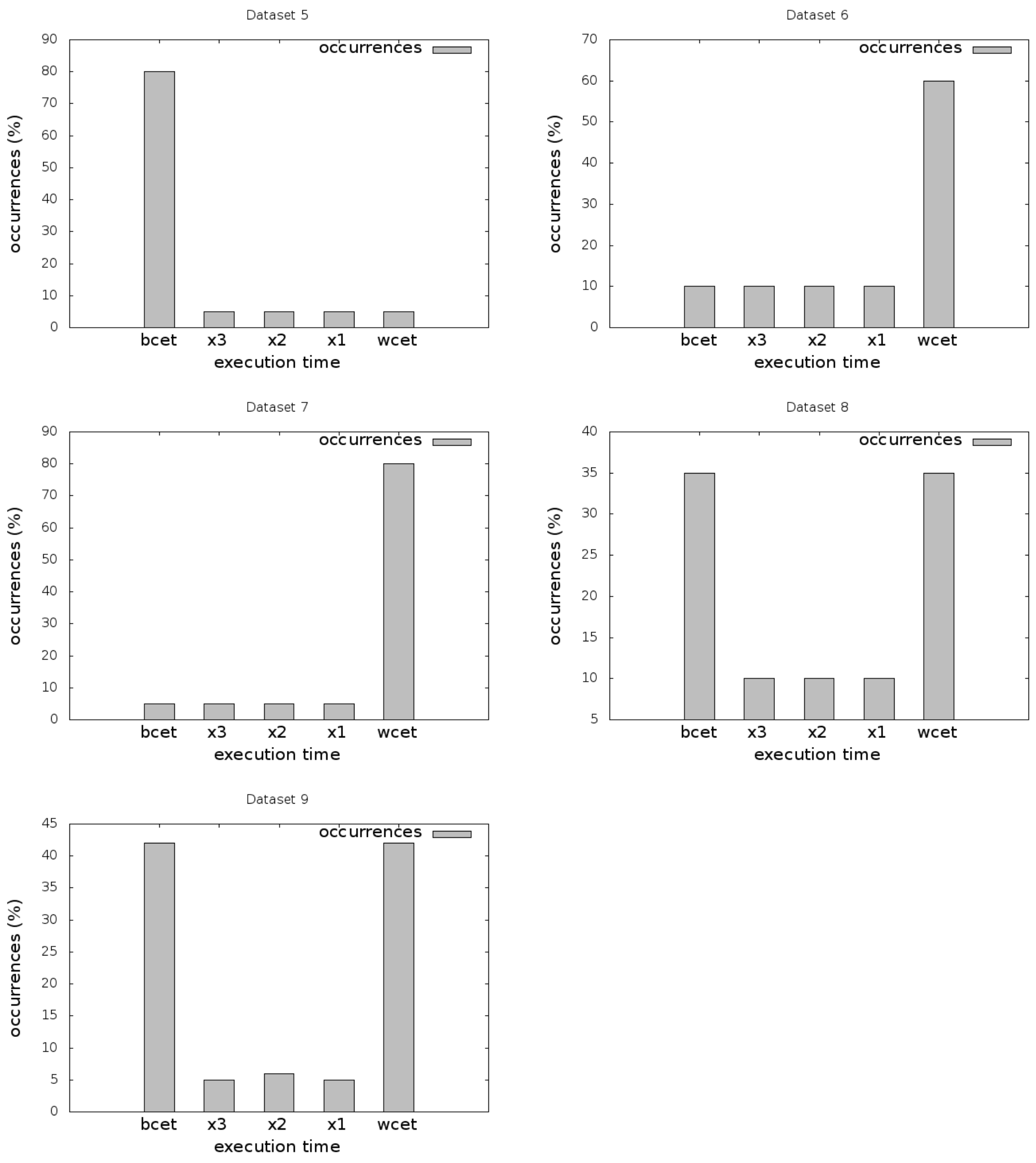

Table 3.3: Datasets containing $\vec{D}'$ and the percentage of occurrences for single actor within CSDF graph G (part B)

framework

5. Redo step 3 and 4 until a total of 20 datasets are produced containing

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	10	29	10	36	60	43	10	50	10	320	0	2
A_2	40	10	50	10	60	60	70	10	80	10	480	320	2
A_3	120	10	150	10	180	60	210	10	240	10	960	960	3
A_4	20	10	25	10	30	60	35	10	40	10	320	1280	2
A_5	20	10	25	10	30	60	35	10	40	10	320	1600	2

Table 3.4: Dataset 2 for all actors within CSDF graph G

data for a range 0% period reduction to 95% period reduction with steps of 5%.

As an example, we take the dataset presented in Table 3.4. For actor A_2 the WCET is 80. The calculated period for that actor given CSDF graph G is 480 and the start times is 320. The next step is to reduce the value taken as the WCET of each actor in the CSDF graph by 5%. We reduce 80 to 76 and recalculate the periods and the start times. The reason for also recalculating the start times is because the period of the predecessor actor could have changed by the recalculation of the period of that actor. For example, if the value we take for the WCET of Actor A_1 is reduced by 5%, the period has been changed as well. As a result of that, the successor actor of actor A_1 is released earlier. In the case of our CSDF graph G, where actor A_2 has multiple successors, the successor with the highest priority can have an earlier start time.

One can see the 5% reduced version of dataset 2 in Table 3.5. Also note that $\vec{D}'$ is the same in Table 3.4 as in Table 3.5

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	10	29	10	36	60	43	10	50	10	304	0	2
A_2	40	10	50	10	60	60	70	10	80	10	456	304	2
A_3	120	10	150	10	180	60	210	10	240	10	912	912	3
A_4	20	10	25	10	30	60	35	10	40	10	304	1216	2
A_5	20	10	25	10	30	60	35	10	40	10	304	1520	2

Table 3.5: Dataset 2 with 5% reduction of the WCET

It could happen that while taking a percentage of the WCET, the result is not an integer value, in such a case we round the value up to its nearest integer value. Also, recalculating the buffer sizes is not necessary in our framework, the buffer sizes that are calculated with the WCET of an actor are minimal. Reduction of the periods and the changing of the start times should not change the order of releases in the schedule.

3.5 Calculating Start Times

Recall from Section 2.5 the method to obtain the start time of a task. For calculating the start time of a task we use the $Daedalus^{RT}$ framework [dae14]. To extract the start time of an actor $A_j \in A$, denoted by S_j, of graph G under a periodic schedule is given by:

$$S_j = \begin{cases} 0 & \text{if } prec(A_j) = \varnothing \\ max_{A_i \in prec(A_j)}\{S_{i \to j}\} & \text{if } prec(A_j) \neq \varnothing \end{cases} \tag{3.3}$$

where $S_{i \to j} =$

$$min\{t \forall k - 0, 1, \cdots, \alpha : prd^S(A_i, E_u) \geq cns^S(A_j, E_u)\} \tag{3.4}$$

where k are the actors within the CSDF graph.

In Equation 3.4, $prd^S(A_i, E_u)$ represents the cumulative production and $cns^S(A_j, E_u)$ represents the cumulative consumption of the corresponding actors. This results in $S_{i \to j}$ being the minimal point in time that, with every firing of the actor, the the cumulative production is equal or greater then the cumulative consumption. Meaning that the actor will always be able to fire because there are tokens in the input buffer available.

As briefly described in Section 3.4, we recalculate the start time at the same time we recalculate the periods. The re-calculated start times are therefore also based on the new reduced value of the WCET.

The re-calculation of the start times is necessary because if the periods are reduced and the start times are not, there will be additional slack added to the schedule. Since the tasks are able to start execution earlier.

3.6 Calculating Buffer Sizes

For calculating the buffer sizes of a communication channel we use the $Daedalus^{RT}$ framework [dae14]. Equation 3.5 presents the minimum bounded buffer size b_u of a communication channel $E_u = (A_i, A_j)$ under a periodic schedule.

$$b_u = max_{k \in [0,1,\cdots,\alpha]} \{prd^B(A_i, E_u) - cns^B(A_j, E_u)\} \tag{3.5}$$

Equation 3.5 ensures that at any point in time, b_u will not receive a token that will cause the buffer to overflow. Also, within a single iteration of the CSDF graph there is at least a single point in time where the buffer is filled completely.

These buffer sizes are minimum so reducing these will likely result in buffer overflows during run-time of the system. While calculating the buffer sizes one has to make sure two constrains are met. The first one is to make sure they are sufficient. This means that given a schedule, an actor is never blocked because it could not write to a specific buffer. In a general hard-real time system, this may never happen. Secondly, an actor should never be blocked at reading from a buffer at any point in time. Section 3.7 will explain that using our approach, the periods will change and therefore the these two constrains are not met. How our framework copes with this is explain in Section 3.7.

3.7 Scheduling

Recall from Section 2.7 the different scheduling algorithms. Our framework uses the FP RM scheduling algorithm. Further in this section we will explain the discussion made regarding the utilization and starvation, also we introduce a technique called *Yielding*.

3.7.1 Utilization

To compute the utilization of the the CSDF graph G used in our framework, we used the $Daedalus^{RT}$ framework. The utilization of our graph ≈ 3.3. However in our framework we use the STM32F4 Discovery board which contains only a single CPU. This means that our hardware is having a upper bound utilization of 1. Therefore we have to multiply the start times and periods by $\lceil 3.3 \rceil = 4$ to be able to schedule G on a single processor. This will increase the amount of slack in our schedule a bit.

The reason why we do not have to recompute the buffer sizes is because we do not change the order of the execution of the task by increasing the periods and start times. Therefore buffer sizes will not change.

3.7.2 Yielding

Sometimes in our approach it could happen that a task is unable to start. The reason for this could be because there is no input data (token) available for that job to consume. Another reason could be that the output buffer is already full. For these situation we introduce *Yielding.* Yielding means that the scheduler will postpone a task and schedule a different task because the initial task that should be scheduled is unable to start execution. The reason that yielding has to be implemented in our approach is because if the scheduler will let the job would start anyway, it would not be able to finish because it will either have to wait for the successor actor to consume data from the buffer first, or wait for the predecessor to create data. In a general hard real-time system, this behavior cannot occur because the periods are based on the WCET of a task. So, once a task fires, the schedule is design in that way, that there will always be input data available for that job. However, this assumption only holds if the periods are based on the WCET and because in our framework the periods are decreased in length, this assumption does not hold. Yielding keeps the scheduler from entering a deadlock state where buffer overflows and buffer underflows occur. In Figure 3.4 one can see a representation of the life cycle of a job, the dotted line represents the situation without yielding. The dark gray shapes represent default hard real-time system behavior, the light gray shapes represent parts that were introduced in order to make the system able to handle deadline misses.

As specified in Figure 3.4, one can see there are three queued a task can be in.

- Ready queue
  This is the queue where every task starts, it holds the tasks that cannot to be fired at that particular moment in time.

- Release queue
  This queue holds all the tasks that are ready to be fired, so in a general hard real-time system, this is where the tasks go at the point in time that there are released.

- Active queue
  This queue holds all the tasks that are currently being executed. The size of this queue is equal to the amount of processors in the system.

For our approach to be able to be operational, there were multiple adjustments made to the standard hard real-time system behavior in the

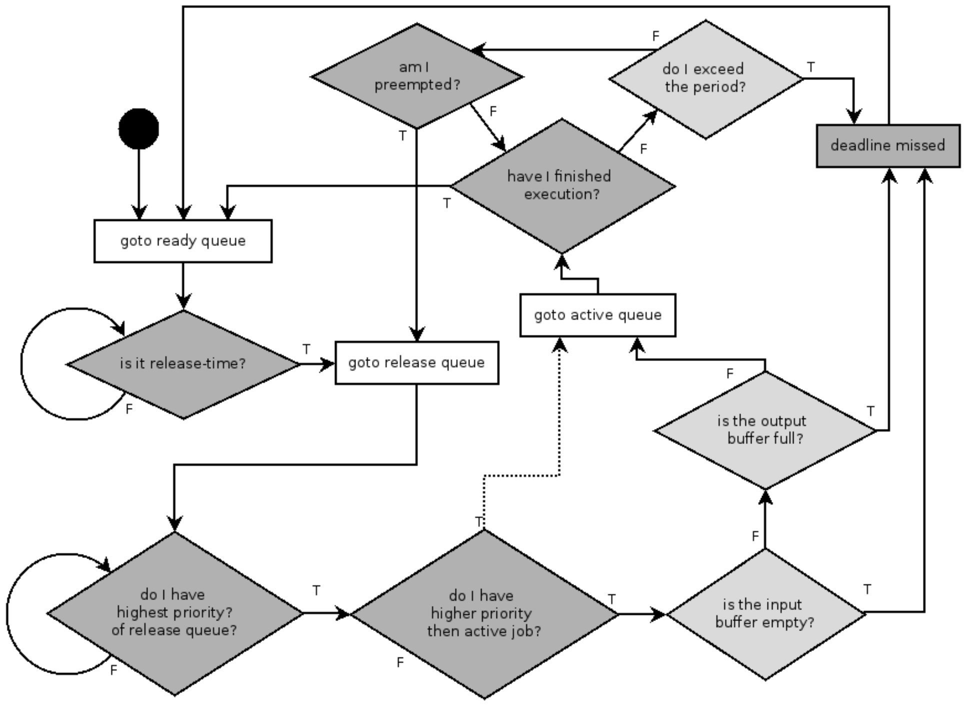
am I preempted?
do I exceed the period?
deadline missed
have I finished execution?
goto ready queue
goto active queue
is it release-time?
goto release queue
is the output buffer full?
do I have highest priority? of release queue?
do I have higher priority then active job?
is the input buffer empty?
T
F

Figure 3.4: Lifecycle of a task within a system

Daedalus^{RT} framework in order to cope with previously described situations. Also, the changes should still ensure proper execution of the tasks. Commonly, a hard-real time operating system assumes that the schedule provided will not have any deadline miss and therefore yielding is not necessary. In a system were the schedule is bound to have deadline misses, yielding should be implemented to make sure that a task is not executed if there is nothing to consume from its input buffer or no room to store data in the output buffer. In both cases the system will run into a buffer overflow or underflow state and cannot be recovered. Also, the checking of deadlines is added to the *Daedulus*^{RT} framework. By default, there is no need for hard real-time systems to check for deadline misses since the schedule is assumed to not introduce these. In our approach this is a necessity to check for deadline misses to be able to measure the influence of the reduction of the periods on the system.

Because the datasets contain different occurrences of ETs for each task, it could be that a task misses its deadline for one job and that the next job has been given a lower execution time from $\vec{D}'$ so both the ETs combined is still less then both periods combined. This will prevent the system from accumulating the deadlines over time and is called *execution time compensation* or *ETC*. This can only occur if the dataset has multiple ETs, so that if the deadline is missed, one of the next releases of that task will make up for the time that was taken the time it missed its deadline.

For example, consider the CSDF graph in Figure 3.5 containing 3 actors and 2 edges. $\vec{D_1'} = (10, 9, 8, 7, 6)$ and $\vec{D_2'} = (10, 9, 8, 7, 6)$.

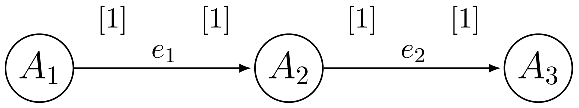

Figure 3.5: CSDF graph with three nodes

Also note that our estimated period for actor A_1 and A_2 are 8 and 8 for our current series of executions. Now consider the scenario that actor t_1 is fired and will execute its WCET being 10. This means that A_1 is missing its deadline because $10 > 8$. If actor A_2 is also selecting its WCET (being also 10), the sink-node A_3 is also missing its deadline. This is because actor A_3 will start firing after the predecessors. This scenario will result in lower throughput of the system because the sink-node was not able to create output within the correct time. However, if actor A_2 is selecting 6 as execution time instead of 10, the total ET for both actor A_1 and A_2 is reduced to 16. This results in the sink-node being fired after both the periods of node A_1 and A_2 (which is $8+8 = 16$). The output is still produced at the correct point in time, even though a deadline was missed. Given our dataset from Figure 3.2, dataset 1 can have no execution time compensation because all the execution times are of a single value.

3.7.3 Starvation

Another phenomenon that is introduced while reducing the value taken as the WCET, is *starvation* of a task. If the periods become smaller, the utilization of the hardware platform increases. If the utilization becomes too much for the hardware platform to handle, deadlines will be missed and some tasks may not be scheduled at all. This is because according to the FP scheduling algorithm, the task with the highest priority should be scheduled. This means that at some point, one or multiple tasks will not executed at all and will stay in the release queue all the time. The task that will "starve" first under a FP rate monotonic schedule is the one with the lowest priority. This is because all other tasks are more likely to be schedule due to their priority. Also, if the task with the lowest priority does get scheduled, it is likely to get preempted later on.

3.8 Framework Usage

In this section, we describe how to use the framework and how to setup everything needed in order to get it working on a Linux based system. We assume basic knowledge about how to use Linux's CLI and how to flash external hardware.

Please note that the package 'libusb-1.0' is needed as well as libusb-devel. These are the package names for the Linux distribution Fedora, For Ubuntu the packages are libusb-1.0-0 and libusb-1.0-0-dev.

3.8.1 Installation

First one has to download the framework from `cvs.liacs.nl:/cvs/lerc/docs/students/frankvansmeden/` and extract it to a desired place. Please make sure to use a file system with support for symbolic links like EXT2 EXT3 or EXT4. 64 and 32 bit systems are both supported to run the framework. Within this framework, part of the *Daedalus*^{RT} framework is already included in order to precompute things like periods start times and buffer sizes.

3.8.2 Running the Framework

Now that the hardware platform is ready, we can start using the framework. Go to the root directory of the downloaded framework and execute the following code:

Listing 3.1: command line interface start script setup

```
sh framework_start.sh <dataset> <case> [percentage]
```

The framework_start.sh script takes three arguments.

- The first argument is the dataset to use. Within the directory of the framework there are 9 datasets present that all have a separate folder in the root directory of the framework. Every dataset consists out of separate cases. Cases are variations within the same dataset. For example, currently, all the actors within our CSDF graph use the same equation for reduction defined in Equation 3.1. Cases could be used to create variations on this and reduce periods more/less for a specific actors within the CSDF graph.

- The second argument is the case the use. This could be used for making small variations within a certain dataset. At this point its not used within the framework and its value will always be 1.

- The third argument is the percentage. If the value 0 is used, the framework will generate datasets and take the WCET to derive the periods start times and buffer sizes. If 1 is used, 95% of the WCET will be used and so on.

Listing 3.2: command line interface start script

```
sh framework_start.sh 2 1 0
```

In listing 3.2 one can see an example of the start arguments of the framework. In this case dataset 2 is being generated. After the dataset generation the framework will automatically start execution on both the hardware platform and simulator. This corresponds to the "running the simulator" and "initialize hardware" stages in Figure 3.1.

Also, if the optional percentage argument is not supplied, the framework will run all the percentages (from 100% to 0% with a step of 5% as explained in section 3.4).

3.8.3 Hardware Platform

By default, the framework uses only the simulator, but there is support to run the simulator and the external hardware platform. In order to setup the hardware platform, one has to download the package from `https://github.com/texane/stlink`. Compile and install this package. Then download the software for the hardware platform from `cvs.liacs.nl:/cvs/lerc/docs/students/frankvansmeden/`. This package contains the Erika Enterprise based operating system is capable of mimicking various applications as explained in Section 3.3. In order to compile the operating system, one has to use the Erika Enterprise IDE that can be downloaded from `http://erika.tuxfamily.org/drupal/download.html` (used version in development: 2.1). This is an Eclipse based IDE that contains plugins to compile the operating system. Once started, one can select the workspace for the IDE to work in. Point the workspace towards the downloaded operating system. Once the IDE has finished loading, press ctrl-b to build the project and generate a BIN-file. This BIN file is the executable needed on the hardware platform and contains the operating system as well as the first stage bootloader. Then navigate to the place where one has extracted the operating system and run the script in the file called "flash". Make sure the hardware platform is connected with a USB cable. Also a few changes have to be made to a file called "start.sh". Comment out line 377 and uncomment lines 373, 376 and 772 to support running on the hardware platform.

3.8.4 Extended Usages

The framework is setup in a flexible way, one can easily adjust pieces of code to add/change functionality. One can easily change the CSDF graph that we have used in our framework or add datasets that have different characteristics.

If one wants to add a dataset, one has to execute the following steps:

1. Create a new folder in the root directory of the framework that starts with "dataset";
2. Within this new folder create another folder that starts with "case";
3. Within this new folder create a file called "raw_input.txt";
4. Fill this file with the content that can be found in Table 3.4 specifying the ETs and occurrences.

Chapter 4

Experiments and Results

In this section, we present the experiments and the results that were obtained using the proposed framework. For these experiments we use the datasets that are presented in Appendix A and also the CSDF graph presented in Section 3.2 as G. The results of the tests are based on the simulation.

In our CSDF graph G there are 5 actors present. The prioritization of the actors is 1 for actor A_1,2 for actor A_4, 3 for actor A_5, 4 for actor A_2 and 5 for actor A_3. In this case, a lower number means a higher priority. Actor A_1 has the highest priority and is able to fire at any given point in time because the actor does not depend on any input. During execution, we monitor which actor gets scheduled and if it misses its deadline. For our experiment, we are mainly interested in actor A_5. This actor is called the *sink-actor*. As long as the sink-actor does not miss deadlines, the throughput of the application as a whole is as to be expected. It could happen that other actors do miss deadlines and fill the buffer for their successor node at a slower rate while the sink-actor does not miss deadlines.

With every firing of an actor within CSDF graph G, the actor randomly picks one of the 5 possible ETs available from a provided dataset. This being either the WCET, BCET or one of the three other possible values within $\vec{D}'$. The dataset also contains the amount of occurrences for each of the ETs in $\vec{D}'$ as explained in Section 3.2. As shown in Table 3.1 all the occurrences have a percentage on how many times they are being executed. To mimic the behavior of executing certain ETs with respect to the percentage, we keep track which ET was executed and count the occurrences. For this we execute the actor 100 times and subtract 1 of the occurrences for each time that ET was randomly selected.

For example, consider actor A with $\vec{D}' = [10, 20, 30, 40, 50]$ and the corresponding occurrences are $[5\%, 10\%, 30\%, 40\%, 15\%]$. When actor A is executed, it randomly selects one of the entries available in $\vec{D}'$, for example 20. D_{20} has a corresponding occurrence value of 10%, we adjust this value to 9 so this ET can only be selected for another 9 times. After 100 executions of actor A, all occurrences are exhausted. When this happens the occurrences reset to their initial value and the sequence starts again.

This reset is done for each actor as many times as the corresponding value in the repetition vector. For our example CSDF graph G is the repetition vector: $[3, 2, 1, 3, 3]$. So, actor A_1 in our CSDF graph will use and reset the dataset 3 times, actor A_2 2 times etcetera. This is done so that we can be sure that every dataset is exhausted at least once, mimicking the distribution of ETs for our specific dataset. We also do this to ensure 1 full execution iteration of the CSDF graph.

Because we cannot be sure about the amount of time a dataset will take given a certain dataset, we wait for 300 executions of the sink-actor. This way we are sure that every actor has exhausted the their dataset at least once. Since we are reducing the periods with 5% for each simulation as described in Section 3.1, there are datasets being generated from which a valid static schedule cannot be derived. We call these datasets invalid datasets.

Invalid datasets are unable to finish execution because the sink-actor is unable to execute 300 times. This is due to the prioritization of the actors. When reducing the periods but not the ET, the actors with the lowest priority will be preempted in favor of actors with a higher priority until there is no room to schedule the actor with the lower priority in any way. This will eventually stop the production of output of the application.

For example, for our CSDF graph G, the actor with the lowest priority (being actor A_3) will eventually not be able to execute at all. This is because it will never be scheduled in favor of any other actor within G. This will cause actor A_4 to stop executing because actor A_3 is not able to create output needed for the execution of actor A_4. So, in order to keep executing the sink-node, every actor has to be able to finish execution. The reason why an increase of delay is possible by reducing the periods is because it could happen that actor A_3 is not able to finished execution and it is being preempted. This means that the output A_3 should have produced will be produced the next time A_3 get scheduled. This is initially preventing actor A_4 from being able to execute but next time A_4 is scheduled it will be able to execute because the input buffer of A_4 has been filled by the second execution of actor A_3.

Consider Figure 4.1. This Figure shows the amount of deadline misses of the sink-actor for each dataset for different percentages of the WCET used

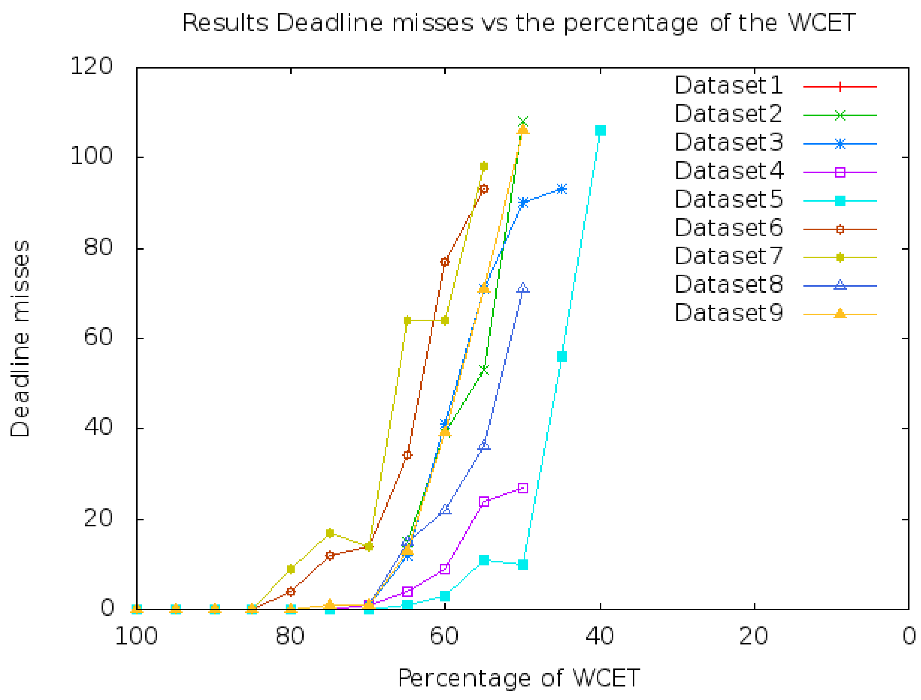

Figure 4.1: Deadline misses versus percentage of the WCET for all datasets

to calculate the period. As one can see. dataset 6 and 7 go up to 55% of the WCET. At 55% of the WCET the lines stop. This indicates that at 50% of the WCET, the sink-actor did not execute 300 times.

During the experiments we let the system run for the estimated time of one full iteration of the CSDF times 25. We assume this is enough time to finish execution even with missed deadlines. If the iteration did not finish completely we assume the dataset is invalid and a no valid static schedule could be derived from it. We can see this in Figure 4.1 onces the lines stop. This indicates that the sink-node was unable to execute a full iteration.

Consider Figure 4.1, as expected, dataset 1 is either schedulable or not. Once a single deadline is misses for this dataset, it accumulates and no task will be able to finish execution within its period. This is expected because all the ETs for each task is the same (8 as described in Table A.1). This means that there is no room for compensation (ETC) as described in Section 3.7.

Figure 4.2 represents the execution time of the application measured in time units in order to finish 300 execution of the sink-actor. As one can see,

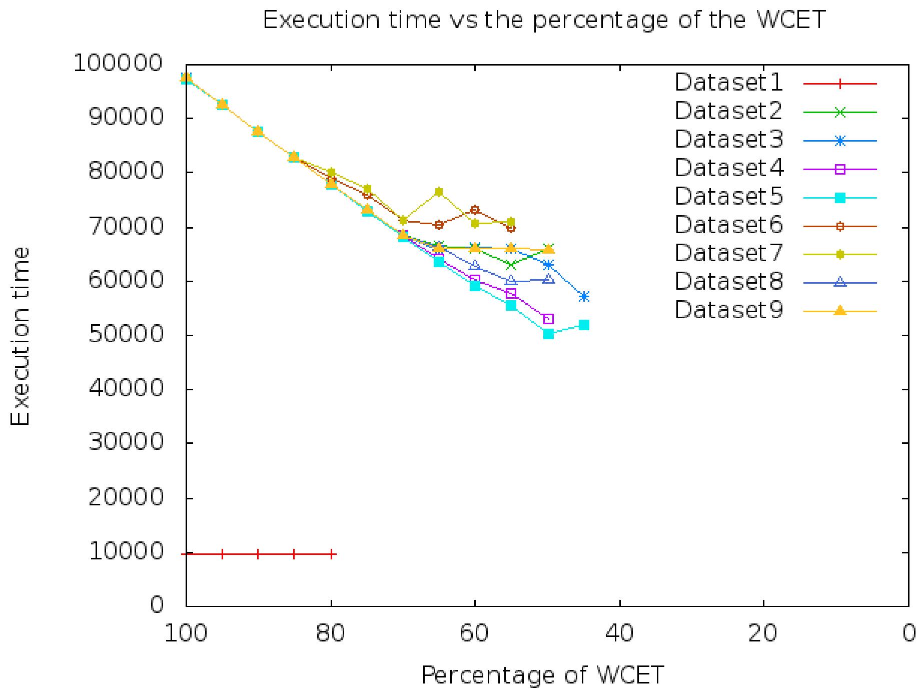

Figure 4.2: Troughput versus percentage of the WCET for all datasets

most of the datasets become more stable in the amount of execution time before being invalid. We call this *execution time leveling* This is probably due to ETC described in Section 3.7. This means that when the periods are reduced, there is a percentage of the WCET where reducing it further will not increase the throughput of the system anymore but only increase the amount of deadline misses.

One can also see that dataset 6 and 7 are becoming unfeasible prior to the other datasets. Dataset 1 is becoming invalid at 75%, the reason for the straight line is because the dataset uses very small ET values, the periods are not changed for the 100%, 95%, 90%, 85% and 80%. Because the new reduced period is always rounded up to the nearest possible integer. This means that when the new reduced period is being calculated for 75% the value is lower then the WCET for the first time. This results in an invalid schedule.

Basically, one can extract three stages for reducing the periods by looking at Figure 4.1 and Figure 4.2. The first stage is where the reduction leads to an increase of the throughput of the system. In this stage, deadlines are being missed already but not that much compared to the second stage. The second stage, is where the the reduction of the periods does not lead to an increase of the throughput but mainly to an increase of the amount of deadline misses. In Figure 4.2 this is very well visible with dataset 9. This dataset levels at 65% and from there on the ET does not change significantly but the amount of deadline misses increases at a much higher rate then before the $70\% - 100\%$ as can be seen in Figure 4.1. The third stage is when the periods are reduced too much and the schedule becomes invalid. The first stage is the most beneficial, some deadlines are missed so the reduction has a direct effect on the throughput of the system. The benefit of the second stage depends on the dataset that is being used. Datasets that have the characteristics of dataset 6 are more likely to level earlier and become unfeasible then datasets like dataset 5.

Comparing Figure 4.1 and Figure 4.2, one can see that there is a correlation between the amount of deadlines being missed and the ET. As soon as the ET of a task does not decrease much more, the amount of deadline misses increases. This is because the utilization of the processor is approximately 1.0, meaning that reduction will not increase the utilization but only stress the system and generating more deadline misses. Reducing the periods even more will make the schedule invalid and the system will enter a deadlock state. As one can see in Figure 4.2, dataset 9 will level at 65%, so for this dataset is does not matter in terms of throughput of the system to reduce the periods by 65% or 50% because the only difference will be the amount of deadlines but the ET will not change much. At this point we can only determine where this leveling point is by reducing the periods further and see the results.

One benefit of scheduling dataset 9 at 65% instead of 50% is the fact that because less deadlines are missed, the sink-node is producing output regularly then it would at 50% reduction. Depending on the type of application this could be more desirable.Note that, like in Figure 4.1, deadlines are started to be missed at 80%. This is because of the periods and start times are scaled by 4.0, as described in 3.7, introducing slack and not utilizing the processor a 100% at $WCET = 100\%$. So when we reduce the length of a period, our analyzed deadlines of the sink-node are not missed immediately. This is because the accumulated periods could be still lower then the start time of the sink-node (as described in Section 3.7.2 as ETC). Dataset 1 is different from the other datasets since this dataset is either valid or not. This is also expected since there is no room in the dataset for any ETC.

Chapter 5

Conclusion and Open Issues

In this thesis, we proposed a framework for exploration and reduction of ETs calculated by the value taken as the WCET of a task within an application. Nowadays the demand for high performance real-time embedded systems keeps growing. However, due to undecidable problems, the throughput on these systems is not optimal due to overestimation. Our framework is capable of simulating applications and possibly show better estimations for the periods given the single-actor-execution-time-vector. These better estimates depend upon the distribution of ETs and therefore are different for each application. However, in this thesis we showed that there is a correlation between characteristics of applications and the amount of ETC. These characteristics are defined by the single-actor-execution-times of an actor. This means that with proper testing, applications can benefit more from having shorter periods and more deadline misses but a higher throughput of the system. Especially for datasets with the same characteristics as datasets 4 and 5. These could benefit more from our reduction approach then datasets like 6 and 7. Datasets 4 and 5 are able to cope with more reduction compared to other datasets. Also, datasets that are extreme versions (datasets 3, 5, 7 and 9) of other datasets but have the same characteristics seem not very much affected by the difference between them. This is probably because the difference between extreme and non-extreme datasets is too small compared to having a different characteristics.

Another benefit of our research is the fact that with our framework, one can make a trade-off between the throughput of the system versus the regularity of the production of the output tokens. This is because the less deadlines are missed, the more regular the tokens will be produced. Depending on the system this could be desirable.

Further research could be done on the following open issues:

5.1 Open issue 1 - Leveling of ET / ETC

Further research could be investigating the "leveling" of ET and trying to calculate at which point this occurs for an application. Applications can benefit very much from this if this could be calculated based on the dataset instead of having to run experiments to find out when this happens.

5.2 Open issue 2 - Different Scheduling Algorithm

Another topic would be to change the scheduling algorithm. The proposed framework uses Rate Monotonic scheduling rules but changing this to EDF and measuring the effect of ETC. Since EDF will always schedule the task with the nearest deadline, the yielding will not be an issue anymore. This is because the task with the lowest priority will not longer be preempted in favor of higher priority tasks if the deadline of the lowest priority task is ealier. This also means that effects like starvation will have different consequences because the execution of tasks will not depend anymore on the priority of the task but on the deadlines of the tasks.

Appendix A

Datasets

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	8	20	8	20	8	20	8	20	8	20	32	0	2
A_2	8	20	8	20	8	20	8	20	8	20	48	32	2
A_3	8	20	8	20	8	20	8	20	8	20	96	96	3
A_4	8	20	8	20	8	20	8	20	8	20	32	128	2
A_5	8	20	8	20	8	20	8	20	8	20	32	160	2

Table A.1: Dataset 1 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	10	29	10	36	60	43	10	50	10	320	0	2
A_2	40	10	50	10	60	60	70	10	80	10	480	320	2
A_3	120	10	150	10	180	60	210	10	240	10	960	960	3
A_4	20	10	25	10	30	60	35	10	40	10	320	1280	2
A_5	20	10	25	10	30	60	35	10	40	10	320	1600	2

Table A.2: Dataset 2 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	5	29	5	36	80	43	5	50	5	320	0	2
A_2	40	5	50	5	60	80	70	5	80	5	480	320	2
A_3	120	5	150	5	180	80	210	5	240	5	960	960	3
A_4	20	5	25	5	30	80	35	5	40	5	320	1280	2
A_5	20	5	25	5	30	80	35	5	50	5	320	1600	2

Table A.3: Dataset 3 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	60	29	10	36	10	43	10	50	10	320	0	2
A_2	40	60	50	10	60	10	70	10	80	10	480	320	2
A_3	120	60	150	10	180	10	210	10	240	10	960	960	3
A_4	20	60	25	10	30	10	35	10	40	10	320	1280	2
A_5	20	60	25	10	30	10	35	10	40	10	320	1600	2

Table A.4: Dataset 4 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	80	29	5	36	5	43	5	50	5	320	0	2
A_2	40	80	50	5	60	5	70	5	80	5	480	320	2
A_3	120	80	150	5	180	5	210	5	240	5	960	960	3
A_4	20	80	25	5	30	5	35	5	40	5	320	1280	2
A_5	20	80	25	5	30	5	35	5	40	5	320	1600	2

Table A.5: Dataset 5 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	10	29	10	36	10	43	10	50	60	320	0	2
A_2	40	10	50	10	60	10	70	10	80	60	480	320	2
A_3	120	10	150	10	180	10	210	10	240	60	960	960	3
A_4	20	10	25	10	30	10	35	10	40	60	320	1280	2
A_5	20	10	25	10	30	10	35	10	40	60	320	1600	2

Table A.6: Dataset 6 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	5	29	5	36	5	43	5	50	80	320	0	2
A_2	40	5	50	5	60	5	70	5	80	80	480	320	2
A_3	120	5	150	5	180	10	210	5	240	80	960	960	3
A_4	20	5	25	5	30	5	35	5	40	80	320	1280	2
A_5	20	5	25	5	30	5	35	5	40	80	320	1600	2

Table A.7: Dataset 7 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	35	29	10	36	10	43	10	50	35	320	0	2
A_2	40	35	50	10	60	10	70	10	80	35	480	320	2
A_3	120	35	150	10	180	10	210	10	240	35	960	960	3
A_4	20	35	25	10	30	10	35	10	40	35	320	1280	2
A_5	20	35	25	10	30	10	35	10	40	35	320	1600	2

Table A.8: Dataset 8 with 100% WCET used as the period

	executions and occurrences												
	BCET		X_1		X_2		X_3		WCET		others		
	ET	%	ET	%	ET	%	ET	%	ET	%	period	start time	buffer
A_1	22	42	29	5	36	6	43	5	50	42	320	0	2
A_2	40	42	50	5	60	6	70	5	80	42	480	320	2
A_3	120	42	150	5	180	6	210	5	240	42	960	960	3
A_4	20	42	25	5	30	6	35	5	40	42	320	1280	2
A_5	20	42	25	5	30	6	35	5	40	42	320	1600	2

Table A.9: Dataset 9 with 100% WCET used as the period

Bibliography

[Ada79] Douglas Adams. *The Hitchikers Guide To The Galaxy.* Pan Books, 1979.

[BCPt02] Guillem Bernat, Antoine Colin, and Stefan M. Pettersreal-time. Wcet analysis of probabilistic hard real-time systems. In *In Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, 2002.

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *Trans. Sig. Proc.*, 44(2):397–408, February 1996.

[BS14] Mohamed Bamakhrama and Todor Stefanov. *On Hard Real-Time Scheduling of Cyclo-Static Dataflowand its Application in System-Level Design.* PhD thesis, Leiden University, 2014.

[But05] Giorgio C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Syst.*, 29(1):5–26, January 2005.

[dae14] Daedalusrt framework. `http://daedalus.liacs.nl`, 2014.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[DBK01] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 266–277, New York, NY, USA, 2001. ACM.

[dis] Stm32f4 discovery. `http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419`.

[eri14] Erika enterprise rtos. `http://www.evidence.eu.com`, 2014.

[LKM98] Sung-Soo Lim, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for optimized programs. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 151–157, Oct 1998.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[PF99] S.M. Petters and G. Farber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 442–449, 1999.

[Tur36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[WKRP05] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, SEUS '05, pages 7–10, Washington, DC, USA, 2005. IEEE Computer Society.

[ZKW$^+$04] Wankang Zhao, Prasad Kulkarni, David Whalley, Christopher Healy, Frank Mueller, and Gang ryung Uh. Tuning the wcet of embedded applications. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '04, pages 472–, Washington, DC, USA, 2004. IEEE Computer Society.

List of Figures

List of Tables