Abstract

For real-time streaming applications such as video decoding, the rate of the application is very important. To fully use the available resources of a multiprocessor platform, thus achieve a high rate, the applications have to be scheduled efficiently. We assume a composable multiprocessor, which means that applications on the platform are independent both in functional and temporal behaviour. Prior work provides composability for the processor utilising two levels of scheduling: composable application-level scheduling with Time Division Multiplexing (TDM) and task-level scheduling with an arbitrary scheduler. In this thesis we are concerned with efficient task-level scheduling. We compare two existing schedulers: Time Division Multiplexing (TDM) and Credit-Controlled Static-Priority (CCSP). TDM is simple and couples latency and rate; CCSP originates from memory controllers and decouples latency and rate. To compare these schedulers we create a dataflow graph of the scheduled application and calculate the parameters that result in the maximum rate. Then these parameters are assessed in experiments on a MPSoC implemented on an FPGA.

From the results of our investigation we can draw two important conclusions. The first is that the formal models of the schedulers lack accuracy in predicting the latency of a task. This is visible in our experiments with buffer bounded applications. The second conclusion is that CCSP schedules tasks more efficiently than TDM. When the period for TDM is exactly proportional to the execution times of the tasks and the application is not buffer bound, then TDM can equal the performance of CCSP.
Scheduling Streaming Applications on a Composable Multi-Processor System
TDM vs. CCSP

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Aster Leegwater

born on 12 August 1986
in Haarlem, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
For real-time streaming applications such as video decoding, the rate of the application is very important. To fully use the available resources of an multiprocessor platform, thus achieve a high rate, the applications have to be scheduled efficiently. We assume a composable multiprocessor, which means that applications on the platform are independent both in functional and temporal behaviour. Prior work provides composability for the processor utilising two levels of scheduling: composable application-level scheduling with Time Division Multiplexing (TDM) and task-level scheduling with an arbitrary scheduler. In this thesis we are concerned with efficient task-level scheduling. We compare two existing schedulers: Time Division Multiplexing (TDM) and Credit-Controlled Static-Priority (CCSP). TDM is simple and couples latency and rate; CCSP originates from memory controllers and decouples latency and rate. To compare these schedulers we create a dataflow graph of the scheduled application and calculate the parameters that result in the maximum rate. Then these parameters are assessed in experiments on a MPSoC implemented on an FPGA.

From the results of our investigation we can draw two important conclusions. The first is that the formal models of the schedulers lack accuracy in predicting the latency of a task. This is visible in our experiments with buffer bounded applications. The second conclusion is that CCSP schedules tasks more efficiently than TDM. When the period for TDM is exactly proportional to the execution times of the tasks and the application is not buffer bound, then TDM can equal the performance of CCSP.

Laboratory: Computer Engineering
Codename: CE-MS-2010-30

Committee Members:

Chairperson: Sorin Cotofana, Computer Engineering, TU Delft
Member: Kees Goossens, Electronic Systems, TU Eindhoven
Member: Said Hamdioui, Computer Engineering, TU Delft
Member: Anca Molnos, Computer Engineering, TU Delft
# Contents

- List of Figures v
- List of Tables vii

## 1 Introduction 1

## 2 Related work 5

## 3 Background 7
- 3.1 Real-time streaming applications 7
- 3.2 Architecture 8
- 3.3 Operating system 8
  - 3.3.1 Time Division Multiplexing (TDM) 9
  - 3.3.2 Credit-Controlled Static-Priority (CCSP) 10

## 4 Formal model 13
- 4.1 Dataflow model 13
- 4.2 Application model 14
- 4.3 Model of scheduled applications 15
- 4.4 Dataflow model of TDM 16
- 4.5 Formal model of CCSP 17
  - 4.5.1 Latency-Rate (LR) model 18
  - 4.5.2 Serial bi-rate model 19
  - 4.5.3 Non-serial bi-rate model 20

## 5 TDM vs. CCSP 21
- 5.1 Exclusive application processor 25
  - 5.1.1 Buffer size 26
  - 5.1.2 Task execution time distribution 28
  - 5.1.3 Task execution rate 30
  - 5.1.4 Slot size 31
  - 5.1.5 Number of tasks 32
  - 5.1.6 Different topologies 34
- 5.2 Shared application processor 35
  - 5.2.1 Task execution time distribution 36
- 5.3 Conclusion 38

## 6 Conclusion and future work 39
- 6.1 Conclusion 39
- 6.2 Future work 42
Bibliography 45

A List of abbreviations 49

B List of symbols 51

C CCSP implementation 53
### List of Figures

3.1 Two levels of scheduling ........................................... 9  
3.2 TDM time wheel ...................................................... 9  
3.3 State diagram of the two parts of a CCSP arbiter .............. 11  
4.1 SRDF graph of an application with two tasks and a connecting FIFO buffer 14  
4.2 Dataflow graph of an application scheduled with a LR task-level scheduler 15  
4.3 WCRT $r_x$ for a task $x$ with time slice $s_x$ in period $p$ .......... 17  
4.4 Latency-Rate model for CCSP ..................................... 18  
4.5 Serial bi-rate model for CCSP ..................................... 19  
4.6 Non-serial bi-rate model for CCSP ................................. 20  
5.1 Worst-Case Response Times (WCRTs) for independent tasks scheduled with TDM and CCSP ........................................... 22  
5.2 Overview of investigated parameters .............................. 23  
5.3 Default application for investigations .............................. 26  
5.4 Application used for buffer size investigation .................... 26  
5.5 Throughput of pipeline with different sizes of FIFO buffers .... 27  
5.6 Application used for execution time distribution investigation ..... 28  
5.7 Throughput of an application where Task 1 has various execution times 29  
5.8 Application used for execution rate investigation .............. 30  
5.9 Throughput for an application with different consumption rate .... 30  
5.10 Application used in slot size investigation ..................... 31  
5.11 Throughput with different time slot sizes ....................... 32  
5.12 Application used for number of tasks investigation ............ 33  
5.13 Influence on throughput for multiple tasks on a processor .... 33  
5.14 Three different task graph topologies with three tasks per processor .. 34  
5.15 Three different configurations. 1 is pipeline, 2 is split and three independent tasks ........................................... 35  
5.16 Application consisting of three tasks with unequal execution times .. 36  
5.17 Fixed application scheduler period and different sized tasks ........ 37  
5.18 Application consisting of ten tasks with equal execution times .... 37  
5.19 Fixed application scheduler period and equal sized tasks ........ 38
List of Tables

B.1  List of symbols .................................................. 51
Acknowledgments

First of all I want to thank Anca Molnos for her guidance, advice, discussions, ideas, reviews of this thesis and much more. Without her this thesis would have been very different. I also want to thank Kees Goossens, sometimes hard to meet because of his busy schedule, but always helpful. From the University of Eindhoven I owe thanks to Benny Åkesson and Firew Siyoum for creating CCSP and its dataflow models. They helped me by explaining CCSP and its effects to me, developing dataflow graphs and discussing my results. Furthermore I want to thank Michiel Leegwater for his constructive review of this thesis.

I want to thank my parents and friends, who forced me to explain things simple, which gave me more insight in my own work. They also helped me with relaxing when I needed it most. Last but not least I want to thank René den Ouden for his trust, help, Matlab support and our many lunches.
As the number of applications on embedded systems grows, the systems become more and more complex. For example, the newest generation of mobile phones can not just make phone calls, but also browse the Internet, take pictures, broadcast music, and navigate on roads. Typically on such a system each application has its own performance demands. Most applications do not only need to execute correctly, but also complete within a timing bound. An application with strict timing demands is denoted as real-time. A real-time application may have various timing demands, e.g. maximum latency and minimum rate. The latency is the interval between availability of input-datum and production of output-datum, and the rate is the time between the production of two consecutive outputs by the application.

An important subset of the real-time applications are streaming applications, such as video or audio encoding and decoding. Streaming applications perform the same set of computations on an infinitely long input stream. This implies that the system starts up once and then runs ‘forever’, which makes the rate of the application more important than the latency.

To ensure that the timing demands of the application are met, timing analysis has to be performed. Timing analysis is only possible in a predictable system where the timing behaviour of an application [36] can be bounded.

Applications share the multi-processor’s resources. If one application were to dynamically take hold of the resources of another application, the second application might not fulfil its timing requirements. Such dependencies between the applications make the timing analysis hard, because all applications must be analysed together. The timing analysis effort is (generally) exponential on the number of applications.

In a composable system, the functional and temporal behaviours of an application are independent of other applications [15]. When one application in a composable system is modified, only the analysis for that application has to be repeated. This enables incremental design and independent testing. As a result, the integration and verification complexity of the system is only linear with the number of applications.

For the complete system to be composable, the hardware, interconnects, and software, all have to be composable. To obtain a composable processor, the application scheduler has to enforce fixed, application-independent, time-slots in which tasks can execute. Otherwise, if an application can influence the duration of a time slot, it could therefore postpone or advance the start of another application. The fixed time slots have to be assigned to the applications by a composable scheduler. An example of a composable scheduler that is easy to implement is Time Division Multiplexing (TDM). TDM is used as the application scheduler in this thesis.
A streaming application consists of multiple tasks with data dependencies between them. These tasks have to be scheduled efficiently in the time slots the application scheduler has assigned to the application. For this, a second level of scheduling is introduced: the task-level scheduler that assigns one task to each time slot. The task-level scheduler does not have to be composable since tasks are not independent; it may be any scheduler that is predictable, because this makes timing analysis possible. The rate of an application is dependent on the latencies of its component tasks, also denoted as the task response time [24, 29, 39].

In this thesis two task-level schedulers are compared. The first scheduler is TDM, since it is also used as application-level scheduler and therefore is already available on the system. TDM assigns each task a time slice (i.e., a number of time slots) out of a fixed period. In a TDM scheduler the latency and rate of a task are coupled, since the latency can not be changed while the rate of a task is kept fixed. Reducing the latency of a task can be achieved by increasing the time slice of the task, which also increases the rate of the task. If one task has a large time slice, potentially other tasks have smaller time slices and lower task rates. The best achievable application rate can be reached when all tasks have the same latency and rate.

The second task-level scheduler is the Credit-Controlled Static-Priority (CCSP) scheduler. During each time slot, all tasks that are ready to execute receive credits and the CCSP scheduler decides, considering the credits and priorities, which task is scheduled. CCSP is a recent proposal [1] to schedule memory requests and is more flexible than TDM, hence it has the potential to deliver higher application rates than TDM. CCSP is very useful for memory requests, because a low latency can be combined with a high rate, something that is not possible with TDM. The results for memory requests [1, 2, 3], do not automatically apply to task scheduling, because tasks are not independent and can execute during more than one time slot. These differences motivate the need to investigate the behaviour of CCSP as a task-level scheduler.

If one is to make an informed choice between TDM and CCSP as task-level schedulers, the performance of a scheduled application has to be known by the system designer. In this research the differences between CCSP and TDM are investigated. Different characteristics of an application, or settings of the platform, influence the performance of the scheduler. We investigate the effects of changing the following six characteristics:

1. The size of the FIFO buffers as tasks communicate data through these FIFO buffers;
2. The distribution of the execution times of the tasks;
3. The amount of data tokens produced and consumed by each task;
4. The size of the time slots of the platform, which is important because only one task can be scheduled per time slot;
5. The number of tasks in an application that are mapped onto one processor; and
6. The topology of the tasks in the application.

Initially, we assume that an application can use the full processor capacity. This is needed to assess only the effects of the task-level scheduler. Subsequently, we assume several applications share a processor, and observe the combined effects of the application-level and task-level schedulers. During the experiments with an application utilising a processor exclusively, we observed that the rate achieved by TDM is the same as the one achieved by CCSP. We have reasons to believe that the CCSP flexibility could be exploited when the tasks’ demands are diverse. Thus for the shared processor we explored only the distribution of the execution times of the tasks leaving as future work the exploration of the other characteristics.

The text is structured as follows. In Chapter 2 we discuss related work. Chapter 3 contains background information regarding the applications, platform and schedulers. We discuss the formal models we use to perform timing analysis on our applications in Chapter 4. In Chapter 5 we compare the timing analysis of the formal model with experiments on the platform. Based on our results we draw the conclusions in Chapter 6, where we also discuss future work.
Related work

A lot of research is performed to understand the effects of scheduling. A survey of many static and dynamic schedulers can be found in [32]. Highly dynamic systems have (a)periodic or streaming applications that join or leave the system at run-time. The schedulers in [18, 19, 23] can be used in such a system. The problem with these systems is that the applications are not composable, which requires reevaluation of the timing behaviour of all applications when one is added or removed from the system. Some schedulers are designed to guarantee small latency and high rate for one high priority task [33] and perform all other tasks in a fixed order. A group of schedulers called the static-priority schedulers [1, 3, 4, 5, 28] assigns a fixed priority to each task. These priorities are used when there are multiple tasks schedulable according to the algorithm. In [31] each task with a hard deadline is assigned an individual processor, so the task does not have to share it resources. In [10] work is performed that is similar to ours. On this platform two levels of scheduling are used, the inter-application scheduler is fixed and the task-level scheduler can be chosen per application. We use this theoretical platform setup, and show with experiments the effects of two-level scheduling on the timing behaviour of the application.

In our research we deal with real-time streaming applications. To guarantee that the deadlines are met, we need to be able to predict the performance of the application. This can be done by modelling the application as a dataflow graph and performing timing analysis on that graph [32, 39]. This method is used very often to calculate the minimal buffer space an application needs to be able to reach its optimal throughput [11, 14, 22, 35, 37, 38]. In these papers tasks and therefore also the dataflow graphs of the tasks assume that each task has its own resources. In our research we want to make efficient use of the resources. This can be done when applications share a resource, which creates the need for a dataflow graphs that models this resource sharing. In [6, 37, 39] the models include the effects of scheduling multiple tasks on a processor. The model from [39] is accurate for a large class of schedulers: the Latency-Rate (LR) servers [34]. LR servers are schedulers that guarantee service at a minimum rate after an initial latency. TDM and CCSP are both LR servers and therefore the dataflow model for LR servers is very useful.

In this thesis we build upon the existing work and perform a comparison of TDM and CCSP as task-level schedulers on a platform with two-levels of scheduling. We compare the effects of the two schedulers on the throughput of an application, both with formal timing analysis and experiments on a platform.
CHAPTER 2. RELATED WORK
In this chapter we describe the details of the platform and applications we utilise in this thesis. We start in Section 3.1 with describing the real-time streaming applications we target with our research. In Section 3.2 we explore the architecture of the experimental environment. The operating system Composable Operating System Extreme (CompOSE) that runs on the Multi-Processor System on Chip (MPSoC) platform is described in Section 3.3. In this section we also discuss the two-level scheduling and briefly describe the functionality of the two schedulers we compare, namely: Time Division Multiplexing (TDM) in Section 3.3.1 and Credit-Controlled Static-Priority (CCSP) in Section 3.3.2.

### 3.1 Real-time streaming applications

A streaming application is an application that performs the same computations on different input data [20]. For simplicity we assume that the input data of an application is always available without creating delay, however our work is easily extendable for the case where input data is delivered with a given rate.

We model streaming applications as a set of tasks with data dependencies between them [26]. The data between tasks is communicated through First-In First-Out (FIFO) buffers. Applications can be mapped over multiple processors, a task can run on one processor only and no task migration is supported. The start of the execution of a task is controlled by its firing rules. A task can start execution or fires, when it has data available in all input buffers and has enough space in the buffers to produce its output. Tasks have a Worst-Case Execution Time (WCET) which is the worst case time from the moment it starts computing the input data till the moment it produces output data, this time is measured in processor cycles. As mentioned in the introduction, the processor time is divided into discrete units, the time slots; in each time slot only one task executes. The number of slots a task needs to execute completely is denoted as its requested service. The Worst-Case Response Time (WCRT) of a task is the worst case time between availability of its input data and production of its output data. The WCRT is therefore a measurement for the maximum time it takes for a task to react on its input.

A real-time application is an application where its correctness depends not only on the correctness of its output, but also upon the time the output is produced [9]. There are three different types of real-time applications: soft, firm and hard real-time. In soft real-time systems it is desirable to meet the deadlines and an occasional deadline miss slightly reduces the quality of the system. In hard real-time systems a deadline miss is catastrophic for the system or the environment and is therefore unacceptable [7]. In firm real-time systems a deadline miss causes no harm to the environment or to the user, but creates severe and unacceptable quality degradation of the output [41].
In this research we target firm real-time streaming applications. Firm real-time streaming applications typically have a much stronger rate than latency constraint [25, 40]. The rate of an application is the time between production of consecutive output data of the application, and the latency of an application is the time between the moment the input of the application is available and the output of the application is produced. Typically the rate of an application is denoted as its \textit{throughput}. For streaming applications we have to guarantee a minimal throughput.

\section{Architecture}

This thesis work is aimed at a composable Multi-Processor System on Chip (MPSoC) environment. This environment consists of multiple processing units, connected with a Network on Chip (NoC) [12] through which the processors can communicate. To be able to make the applications composable, all system’s hardware resources have to be composable. We conduct our experiments using a composable, predictable platform [15, 16] available on Field Programmable Gate Array (FPGA). The multiple computing processors in our setup are each implemented as a Xilinx MicroBlaze, another MicroBlaze is used to monitor the system values.

\section{Operating system}

The operating system on our platform is Composable Operating System Extreme (CompOSe) [15]. This operating system provides the capabilities to run multiple applications on the same processor, while preserving composability of the applications. CompOSe uses fixed size task time slots in which one of the tasks of an application can be scheduled. After each task time slot, a fixed size operating system time slot is executed. The time slot lengths have to be application independent, otherwise one application can influence the moment when another application can be scheduled, and the system loses composability. A fixed size time slot is an easy way to implement application independent time slots.

CompOSe runs separately on each processor and the instances of CompOSe do not have to synchronise, thus the multiprocessor system is scalable.

During its time slot, the operating system schedules an application and task, performs the context switch, and sends the slot information (e.g. application id, task id, timing information) to the monitoring MicroBlaze.

The operating system implements two levels of scheduling as shown in Figure 3.1: application-level and task-level scheduling. The application-level scheduler decides which application uses the next time slot. For the application-level scheduler we use a TDM scheduler (see Section 3.3.1), due to its predictability and composability properties. When the application-level scheduler has decided which application runs next, the task-level scheduler of that application chooses the task. The task-level scheduler can be different for each application, it can be any scheduler as long as it is
predictable. Only one task can run per time slot, when this task finishes before the end of the time slot, the remaining time in the slot is idle. Idle time is annotated as slack; internal slack is remaining idle time within a time slot and external slack is a time slot where no task is scheduled.

In this thesis we compare TDM and CCSP as task-level schedulers. In Section 3.3.1 and 3.3.2 we describe each of the schedulers in detail.

### 3.3.1 Time Division Multiplexing (TDM)

A Time Division Multiplexing (TDM) scheduler divides the processor or application time in periods of equal length, and allocates fixed length slices of a period to an application or task, respectively. The TDM schedule of a task has two parameters: the time slice $s_x$ of the task $x$ and the period $p$ of the scheduler. In our case, both the period $p$ and the time slices $s_x$ consist of an integer number of time slots. In Figure 3.2 an example of a TDM wheel, with a period of nine time slots divided into the time slices for four tasks, is shown. In this example, task one has a time slice of one slot out of a period of nine slots, while task two has a time slice of three time slots out of nine.

A task can only be executed if it can fire, which means that it has all input data available and it has space in the output buffer to produce its result. If a task can not fire at the beginning of one of its time slots, this slot is idle (external slack). The Worst-Case Response Time (WCRT) of a task is given by: (1) the worst case time the task has to wait till it can process its already available input, (2) the execution time of the task, and (3) the size of the slice allocated to the task. The formula for the WCRT of a task scheduled with TDM is presented in Section 4.4.

The throughput of an application depends on the WCRTs of its tasks. Reducing the WCRT of a task means increasing the size of its time slice, which also results in a higher rate for the task. Potentially the time slice of other tasks is decreased and therefore their
CHAPTER 3. BACKGROUND

WCRT increases and they have lower rates. Because the throughput of the application depends on the WCRTs of all tasks, the throughput is limited.

3.3.2 Credit-Controlled Static-Priority (CCSP)

Credit-Controlled Static-Priority (CCSP) is a scheduler that is more flexible than TDM, because it does not fix the order of the tasks. CCSP is proposed for scheduling memory requests [1]. CCSP decides which task to schedule, based on its collected credits and priority. In a CCSP scheduler a task has three parameters: the priority $\phi_x$, the allocated rate $\rho'_x$ and the burstiness $\sigma_x$. Priorities are unique (1 is given the highest priority) and from the tasks that can fire and have enough credit the highest priority task is selected. The allocated rate describes how many credits a task receives per time slot, and therefore decides the execution rate of the task. The sum of the allocated rates of all tasks of an application on a processor is maximum one. If the sum of all the allocated rates is one, this means that the processor is fully used. The burstiness is the amount of credits a task receives if it fires for the first time or after a period it could not fire. These burstiness credits shorten the latency of the task. The burstiness of a task is always greater or equal to one.

A CCSP arbiter consists of two parts: a rate regulator and a static-priority scheduler [3, 1]. Figure 3.3 shows the main functions of both the rate regulator and the static-priority scheduler. These functions are described in more detail in the next paragraph, the implementation of the CCSP task-level scheduler on our platform can be found in Appendix C.

The rate regulator checks if a task can fire, updates credits and decides which tasks are eligible for scheduling. If a task can fire, it receives the number of credits defined by its allocated rate $\rho'_x$. A task is eligible for execution if it can fire and has enough credits to run at least one slot. If a task can fire for the first time or after a period it could not fire, it receives initial credits of the size of the burstiness $\sigma$. This is most useful when we have a very short task that should run immediately when it can fire, but runs rarely. Because it has to run rarely it would get a small allocated rate and without a burstiness bigger than one, the task would have a long latency. This illustrates that the latency and the rate of a task scheduled with CCSP are decoupled.

The static-priority scheduler schedules the task with the highest priority from the eligible task list. The credits of this scheduled task are decreased. If there are no eligible tasks, the static-priority scheduler can not schedule any task, and the processor is idle during that time slot (external slack).

The WCRT of a task scheduled with CCSP consists of two parts: the initial latency and the execution rate. Both the initial latency and the execution rate of the arbiter are measured in time slots. The initial latency is the time between availability of the input data and the start of execution of the task, and depends on the burstiness and allocated rate of higher priority tasks. Because the highest priority task is always selected when it is eligible, it has an initial latency of zero. The execution rate of a task
is the time between the production of two outputs, it can be calculated as the allocated rate of the task divided by the requested service of the task. The formulas for both the initial latency and execution rate of task scheduled with CCSP can be found in Section 4.5.
In this chapter we present the dataflow model of a streaming application, standalone, and mapped and scheduled on the platform. This formal model is used to analyse the timing behaviour of the application. We start in Section 4.1 with the basic theory about dataflow graphs and dataflow analysis. In Section 4.2 an application is modelled as a dataflow graph. We assume that this application is mapped on an infinite number of processors, so there are no shared resources between tasks. In Section 4.3 the application is mapped on a finite number of processors. Because of the finite number of processors, tasks share resources and task-level scheduling is necessary. This task-level scheduling influences the timing behaviour of the application. Therefore we present the dataflow model of an application scheduled with a task-level scheduler. The specific dataflow models for TDM and CCSP are shown in Section 4.4 and 4.5, respectively.

4.1 Dataflow model

Streaming applications can be modelled well with dataflow graphs. Dataflow graphs can be used for timing analysis [29]. There are many different types of dataflow graphs, for example Single-Rate Dataflow (SRDF), Multirate Dataflow (MRDF) [25], Cyclo Static Dataflow (CSDF), Dynamic Dataflow (DDF) [8], Synchronous Dataflow Graph (SDFG), and Homogeneous Synchronous Dataflow Graph (HSDFG) [13, 39]. Our streaming applications have a constant production and consumption rate and can be modelled as MRDF graphs. All MRDF graphs can be transformed into SRDF graphs [25] to perform timing analysis. We start with a description of SRDF graphs.

A SRDF graph is a directed graph described by $G = (V, E, d, r)$. $V$ is a finite set of actors and $E$ is a set of directed edges. Data is transported in discrete units called tokens, via edges, from the output of one actor to the input of another actor. The set $d$ describes the number of initial tokens for each edge in $E$, where $d_j$ is the number of initial tokens on edge $j$. An initial token on an edge is also called a delay on the edge. The set $r$ contains the delays of the actors in $V$, where $r_i$ is the delay of actor $i$ [25, 39]. In a SRDF graph an actor consumes/produces one token per incoming/outgoing edge. An actor can only be activated or fired if there is a data token available on each incoming edge and there is space to produce a data token on the outgoing edge.

A MRDF graph is not very different from an SRDF graph. The only difference is that in a MRDF graph, each actor can produce or consume more than one token from an edge. The number of tokens produced or consumed per firing is fixed and known at compile time. The MRDF graph is described by $G = (V, E, d, r, p, c)$. The set $p$ is the constant number of tokens produced on each edge in $E$, where $p_j$ is the number...
of tokens produced on edge $j$. In the same way $c$ is the set with token consumptions, where $c_j$ is the number of tokens consumed from edge $j$.

All MRDF graphs can be transformed into SRDF graphs by replicating actors as described in [17].

The SRDF model allows timing analysis. A fundamental upper bound for the throughput of an SRDF graph can be found as $1/\mu$ [24, 29, 39] where $\mu$ is the Maximum Cycle Mean (MCM) of the graph:

$$\mu = \max_{g \in C(G)} \frac{\sum_{i \in V(g)} r_i}{\sum_{j \in E(g)} d_j}$$

$g$ is a cycle in the set of all simple cycles $C(G)$ of the graph. A simple cycle is a path in the graph from an actor to itself that visits a set of actors, each of them once [27, 41]. $V(g)$ is the set of all actors traversed by cycle $g$ and $E(g)$ the set of all edges traversed by cycle $g$. $r_i$ is the delay of actor $i$ and $d_j$ is the initial number of data tokens on edge $j$. We calculate the sum of the delay of the actors divided by the sum of the delay of the edges in a simple cycle and take the maximum of all simple cycles as the MCM of the graph. The cycle that gives the MCM is the critical cycle of the graph.

4.2 Application model

We model an application mapped on an infinite number of processors. This means that the tasks each have their own resources, thus scheduling effects do not have to be taken into account.

Figure 4.1: SRDF graph of an application with two tasks and a connecting FIFO buffer. Below the task graph and up the corresponding dataflow graph

Figure 4.1 presents a SRDF model of a simple producer-consumer application, that
4.3. MODEL OF SCHEDULED APPLICATIONS

consists of two tasks with a FIFO buffer in between. Each task is modelled as one actor with the WCET of the task as delay of the actor. Each actor has a self-edge with a delay of one, to model that a task can not run concurrently with itself. The FIFO buffer between the tasks is modelled as two opposing edges. The backwards edge has a delay $d$ that represents the size of the FIFO buffer [8, 21].

4.3 Model of scheduled applications

In this section we present a dataflow model for an application scheduled on a limited number of processors. This implies that tasks of the application have to share the resources of a processor. When the tasks share resources, a task-level scheduler is required. The task-level scheduler affects the timing behaviour of the application, and has to be taken into account when modelling the application as a dataflow graph. In this model each task is represented by a set of actors.

In [39] a dataflow model is proposed that includes the effects of a Latency-Rate ($\mathcal{LR}$) server. Since both TDM [39] and CCSP [1] prove to be $\mathcal{LR}$ servers, this model suits our purposes.

The theory of $\mathcal{LR}$ servers enables the modelling of the worst-case behaviour of a broad range of scheduling algorithms. The most important concept in this theory is a busy period, a period of time during which requests arrive faster than, or at the same rate as they are handled. A $\mathcal{LR}$ server guarantees a minimal throughput during the busy period. The behaviour of a task scheduled with a $\mathcal{LR}$ scheduler is determined by its initial latency and allocated rate. The initial latency is the worst-case time between availability of the input and the start of the execution of a task. The allocated rate is the minimum rate at which a request is served during a busy period [34].

![Figure 4.2: Dataflow graph of an application scheduled with a $\mathcal{LR}$ task-level scheduler. Below the task graph and up the corresponding dataflow graph](image-url)
The formal model of a simple producer-consumer application scheduled with a LR server is presented in Figure 4.2. Each task in the application is not represented by one, but two actors in the graph. The first actor represents the initial latency $\theta_x$ of a task $x$, the time between the availability of its input and the start of its execution. The second actor represents the throughput $\frac{1}{\rho_x}$ of a task $x$, the time between two consecutive finishes of the task. The total response time $r_x$ of a task $x$ is

$$r_x = \theta_x + \frac{1}{\rho_x}$$  \hspace{1cm} (4.2)

The difference in the timing analysis between the TDM and CCSP scheduler is mainly in the formulas for the initial latency $\theta_x$ and rate $\rho_x$ of a task.

### 4.4 Dataflow model of TDM

Figure 4.2 presents the dataflow model of a simple producer-consumer application scheduled with a LR scheduler. The formulas for the initial latency and rate of a task scheduled with TDM are derived in this section.

We start with defining the parameters of the TDM scheduler. The parameters of a TDM scheduler are: the number of time slots in the wheel $p$ and the size of the time slice $s_x$ (in slots) assigned to a task $x$. We assume that the time slots of one time slice are consecutive within the time wheel. The task $x$ has a Worst-Case Execution Time (WCET) $W_x$ measured in cycles, and a requested service $w_x$ that represents the number of time slots a task needs to execute. The length of a time slot is $\lambda$, which is measured in cycles. The relation between $W_x$ and $w_x$ can now be written down as:

$$w_x = \left\lceil \frac{W_x}{\lambda} \right\rceil$$  \hspace{1cm} (4.3)

The WCET $w_x$ is rounded, since we always allocate an integer number of slots to a task in the system. The WCRT $r_x$ (measured in time slots) of a task $x$ scheduled with TDM is [6, 24]:

$$r_x = w_x + (p - s_x) \frac{w_x}{s_x}$$  \hspace{1cm} (4.4)

In Figure 4.3 we show the WCRT $r_x$ of a task with a requested service of two slots. The WCRT occurs when the input data for the task comes available just after its assigned time slice $s_x$ has past. The task has to wait $p - s_x$ time slots to start execution. When the task is too long to fit in one time slice, it has to wait another $p - s_x$ time slots for the start of its next time slice. The WCRT $r_x$ is the time from the worst case moment the input data is available, until the moment the execution of the task finishes.

From [39] we know the execution rate of a task as shown in the following equation:

$$\rho_x = \frac{s_x}{w_x \cdot p}$$  \hspace{1cm} (4.5)
4.5. FORMAL MODEL OF CCSP

Formalising CCSP's timing behaviour is hard, since there are multiple parameters that influence the scheduler's behaviour. These parameters are the priority $\phi_x$, the allocated rate $\rho'_x$ and the burstiness $\sigma_x$ as defined in Section 3.3.2. As mentioned in Section 4.3 CCSP is a LR server, and a LR model can be applied. This LR model is conservative, but not very accurate. For example it does not take into account that the burstiness parameter $\sigma_x$ of a task scheduled with CCSP causes this task to run temporarily at a higher rate than its allocated rate.

In Section 4.5.2 and 4.5.3 we show two different bi-rate models, that include the effects of burstiness [30]. Both models have restrictions on when they can be used.

When we model applications scheduled with CCSP we can use a combination of all three models. Each task is modelled with the best suitable model. In 4.5.2 we present formulas for two variables: $\rho^*$ that describes the high rate at which an application can run, and $h$ which describes how many executions can be performed at high rate. The value of these two variables decides which model we should use for the task, as follows:

- If $h > 1$ and $\rho^* > \rho'$, the task can be modelled with the non-serial bi-rate model
- If $h = 1$ and $\rho^* > \rho'$, the task can be modelled with the serial bi-rate model
- If the former two conditions do not hold, e.g. $h < 1$ or $\rho^* = \rho'$, the LR model should be used
4.5.1 Latency-Rate (LR) model

In Section 4.3 we presented a dataflow model that can be used to include the effects of a LR scheduler into the dataflow graph of an application. Since CCSP is a LR server [1] this model can be used. The model in Section 4.3 is for tasks with a requested service of one slot. In our case a task might have a requested service of more than one slot. This is where a MRDF graph is useful, the requested service of the task is modelled as the production rate of the initial latency actor $\theta_x$, as shown in Figure 4.4.

![Figure 4.4: Latency-Rate model for CCSP](image)

A low priority task may experience a long waiting time before it receives service at its allocated rate. This is because higher priority tasks have precedence when two tasks are eligible. The initial latency $\theta_x$ therefore depends on the burstiness and rate of the higher priority tasks. The initial latency can be calculated with Equation (4.9) [1]:

$$\theta_x = \sum_{\forall y \in X^+_x} \sigma_y \left(1 - \sum_{\forall y \in X^+_x} \frac{\rho_y}{\rho'_x}\right)$$

(4.9)

where $X^+_x$ is the set of all tasks with a higher priority than task $x$, $\sigma_y$ is the burstiness and $\rho'_y$ is the allocated rate of task $y \in X^+_x$. The initial latency $\theta_x$ in this equation is measured in time slots.

With the latency and rate described in Equation (4.9), we derive the response time $r_x$ for task $x$ as:

$$r_x = \frac{\sum_{\forall y \in X^+_x} \sigma_y}{1 - \sum_{\forall y \in X^+_x} \frac{\rho_y}{\rho'_x}} + \frac{w_x}{\rho'_x}$$

(4.10)

The proof of correctness for the response time formula is presented in [1]. Here we present an intuitive proof, in which we use the concept of a busy period of a LR server as described in Section 4.3.

There are two different cases, (1) the task is eligible for the first time and starts a busy period or (2) the task is already eligible for a longer time and therefore is in a busy period.

In case (1), the task starts a busy period, this implies that it is eligible and has enough credits to be scheduled at least one time slot. Thus the task has to wait for all higher priority tasks to finish, before it can be scheduled. This case is modelled by the initial
4.5. FORMAL MODEL OF CCSP

In case (2) the task is already in a busy period, for a long time. We assume that this long time is enough for all higher priority tasks to finish execution. CCSP now guarantees a task its allocated rate \( \rho'_x \) for the rest of its busy period. In the graph of Figure 4.2 this case corresponds to actors \( \frac{1}{\rho_x} \) and \( \frac{1}{\rho_y} \).

The LR model of CCSP is conservative, but not very accurate. One of the other, more accurate, CCSP models can be used, when its applicability conditions are met.

4.5.2 Serial bi-rate model

The LR model in the previous paragraph, does not include the effects of the burstiness \( \sigma_x \) of task \( x \), which makes it inaccurate. Due to its burstiness, a task could temporarily run at a higher rate than its allocated rate.

The serial bi-rate model, includes the effects of the burstiness of a task. It models a task \( x \) with three actors: the first actor represents the initial latency \( \theta_x \), the second actor the high-rate execution \( \frac{1}{\rho'_x} \), and the third actor the low-rate execution \( \frac{1}{\rho_x} - \frac{1}{\rho'_x} \) of the task \( x \). The model is a MRDF graph if the task has a requested service higher than one. The three actor model is shown in Figure 4.5.

\[
\rho'_x = 1 - \sum_{y \in X^+_x} \rho'_y \quad (4.11)
\]

\[
h_x = \left[ \theta_x + \frac{\sigma_x + \rho'_x - 1}{\frac{1}{\rho_x} - \frac{1}{\rho'_x}} \right] \quad (4.12)
\]

Figure 4.5: Serial bi-rate model for CCSP

The formula for the initial latency \( \theta_x \) can be found in Equation (4.9). New in the serial bi-rate model are \( \rho'_x \) given in Equation (4.11) and \( h_x \) given in Equation (4.12):
We call this bi-rate model "serial", because, unlike the bi-rate model in the next section it models the low rate as a serial firing of two actors.

This model is only conservative if $h_x = 1$ and $\rho^* > \rho'$. If $h_x \neq 1$ or $\rho^* = \rho'$, another model, e.g. the non-serial bi-rate model or $\mathcal{LR}$ model, is more accurate.

### 4.5.3 Non-serial bi-rate model

The non-serial bi-rate model is similar to the serial bi-rate model, but it is conservative only if $h > 1$ and $\rho^* > \rho'$. Like the serial bi-rate model it consists of three actors per task and the formulas for $\rho^*_x$ and $h_x$ are the ones found in (4.11) and (4.12). The non-serial bi-rate model is shown in Figure 4.6.

![Figure 4.6: Non-serial bi-rate model for CCSP](image)

The difference with the previous bi-rate model is in the value $\frac{1}{\rho_x}$ associated with the low-rate actor and the presence of a self-edge at the low-rate actor. The non-serial bi-rate model, models the low-rate of a task with only one actor, not with a serial execution of two actors.

The non-serial bi-rate model can only be used if $h > 1$ and $\rho^* > \rho'$, otherwise we have to use one of the other models.
In this chapter we compare TDM and CCSP as task-level schedulers. We perform a thorough investigation, that consists of a timing analysis and experiments on the FPGA platform (Section 3.2), with various application and platform settings. From the results of this investigation we can advice about the use of the schedulers.

If an application consists of independent tasks, the tasks can be modelled similar to memory requests; as a set of actors that represent the initial latency and rate of a task as described in Chapter 4. Both are served by a resource, the tasks are served by a processor and the memory requests by a memory. Because the independent tasks and memory requests are modelled similarly, the results from [1] hold for both. When scheduling independent tasks or memory requests the biggest difference between TDM and CCSP is found in the WCRT. Because the tasks or memory request are independent, the latency equals the WCRT of the task or memory request.

To indicate the quantitative differences between TDM and CCSP, we perform the following experiment. We execute ten independent tasks on a processor, where each task has an allocated rate of 10% of the processor. For simplicity, we consider tasks with an equal requested service of five slots. The tasks have priorities according to their task number, which means that task 1 has the highest priority and task 10 the lowest priority.

For CCSP we investigate different burstiness values, because the burstiness influences the WCRT of a task. The WCRT of a task is composed of the initial latency, the execution rate and the requested service of the task. The initial latency of a task scheduled with CCSP depends on the burstiness and allocated rate of all the higher priority tasks. Because of the burstiness the execution rate of a task can be temporarily higher than its allocated rate. This has two reasons: a task starts with more credits because of its own burstiness and/or a task can collect credits during its initial latency that is extended due to the burstiness of higher priority tasks. For clarity we only vary the burstiness of the highest priority task, because this influences the initial latency of all lower priority tasks, and is similar with varying the burstiness of several tasks.

The WCRT of the tasks in this experiment are shown in Figure 5.1. When all tasks have a burstiness of one, their WCRT is smaller than the WCRT of tasks in TDM, except for the lowest priority task that has an equal WCRT. Increasing the burstiness of the highest priority tasks has two effects. The first effect is that the WCRT of the highest priority task is reduced, until it is equal to its requested service. The second effect is an increase in WCRT for the lowest priority task(s). When the burstiness of the highest priority task is smaller or equal to the requested service of the tasks, only the lowest priority task has an increased latency and therefore WCRT. With a burstiness
between one and two times the requested service of the tasks, the two lowest priority tasks have an increased latency and WCRT. A burstiness between two and three times the requested service of the tasks influences the WCRT of the three lowest priority tasks, and so on. The WCRT of the other tasks is not influenced, their initial latency grows but during the extra time slots they have to wait, they collect credits that allow them to run at a higher rate than allocated for a small period of time after they start execution.

We have scheduled ten tasks with the same requested service at the same rate. We have seen that CCSP can schedule these tasks with very different latencies, while TDM can give them only one latency. This illustrates that latency and rate for TDM are coupled and for CCSP are decoupled.

![WCRTs of independent tasks](image)

Figure 5.1: WCRTs for independent tasks scheduled with TDM and CCSP

In reality the tasks of streaming applications are never independent, because they communicate data. We compare the behaviour of TDM and CCSP as task-level schedulers for different streaming applications. The latency and throughput of an application depend on the scheduler and the initial latency and rate of its tasks. The latency of an application is the time between the availability of its input data and the production of its output data. For a streaming application the application latency is of less importance than the throughput of the application. The throughput of the application is the time between the production of two application outputs and is
therefore the focus of our research. The application throughput can be calculated with the WCRT of its tasks and the behaviour of the application- and task-level scheduler.

We perform two series of investigations, in the first an application can use the entire processor, and in the second an applications shares the processor with others. We start each investigation with creating a formal model of the scheduled application. We want to compare the best throughput of the application, and therefore have to find the scheduler parameters that deliver this best throughput. To find these parameters we write a Matlab script that iterates through possible values of the parameters (exhaustive search). The script needs a description of the application as input, that consists of: the WCET, production and consumption rates of the tasks, the length of a time slot, the length of an operating slot, the mapping of the tasks on the processors, and the size of all FIFO buffers. The script iterates through some loops to test different combinations of scheduler parameters. For each combination of parameters Matlab calculates the throughput of the application model with Hebe [37, 38]. The parameters that produce the highest throughput for the application are selected. The output of the script is an array of all the parameter combinations that produce the optimal throughput.

For TDM the output consists of the time slices for each task and the length of the scheduler period. For CCSP the outputs are the allocated rates and priorities of a task, to limit the number of combinations the burstiness of all tasks is set to one. We choose one set of the best parameters for an experiment on the platform. We compare the results of the Matlab throughput calculation and the experiments.
In the first series of investigations the application-level scheduler only schedules one application. This application receives all the schedule time and therefore the effects of the application-level scheduler are not observed. Therefore the task-level scheduler is free to set the length of the scheduler period.

Our investigations covers all the parameters that influence the throughput of an application. The parameters can be grouped into application parameters (buffer size, different topologies), task parameters (execution time and execution rate), platform parameters (slot size and processor utility) and mapping of the application on the platform (number of tasks). The parameters are pictured in Figure 5.2 and are explained in more detail in the following list:

1. Buffer size. Tasks communicate data through FIFO buffers. There are two distinct situations. In a buffer bound application, the buffers are small and a task can not re-run immediately after it has finished, due to the fact that it has to wait for the next task to consume the data token from the buffer. The throughput of the total application is therefore limited by the size of the buffers. In contrast to a buffer bound application, we define a computation bound application. In a computation bound application the buffers are large enough, such that the throughput of the application is not limited by the buffer size. The throughput of the application is now given by the lowest execution rate of the tasks.

2. Task execution time distribution. In this investigation one task has a different execution time than the others.

3. Task execution rate. The consumption and production rate of a task fix its execution rate. We investigate the task execution rate by increasing the consumption rate of one task. Therefore the producing task runs more often than the consuming task.

4. Slot size. The size of the time slots on the processor determines the number of slots in which a task executes (requested service). When the time slots are small, there are more operating system slots, thus more overhead in the system. The execution of the task is fragmented over more time slots, and the maximum idle time within a time slot is small (internal slack). The slot size decides the balance between overhead and internal slack in the system and therefore the possible efficiency of the scheduler.

5. Number of tasks. In this investigation we vary the number of tasks that is mapped onto one processor. TDM can increase its period and divide the time slices equally over all tasks and this results in an equal WCRT for the tasks. CCSP gives each task a unique priority, and the initial latency of a task is determined by this priority assignment. CCSP can give tasks the same rate, but they have different initial latencies and thus different WCRT's.

6. Different topologies. The topology of an application describes the dependencies of the tasks in the application. When an application has more non-dependent tasks, the task-scheduler has more freedom to choose between tasks that are available
5.1 Exclusively Application Processor

at the same moment. There are many different topologies possible. Relevant structures are divergence, a task produces tokens for multiple consumers, and convergence, a task consumes tokens from multiple producers. The application can consist of balanced paths (paths with the same workload) or unbalanced paths. Studying all possible topologies is out of the scope of this research and is left for future work (Section 6.2). In this research we compare the behaviour of the schedulers on three different topologies, independent tasks, an application with convergence and divergence and a pipeline application.

In the second series of investigations applications share the processor with other applications. The throughput of the application is now influenced by both the application-level and task-level scheduler. The application-level scheduler is always TDM and fixes the length of the applications time slice. During the experiments with an application utilising a processor exclusively, we observed that the rate achieved by TDM in a computation bound application is the same as the rate achieved by CCSP. We have reasons to believe that the CCSP flexibility could be exploited when the tasks' execution times are diverse. Thus for the shared processor we explore only the distribution of the execution times of the tasks.

In Section 5.1 we present the first series of investigations, in which the application can use the processor exclusively. In Section 5.2 we observe the behaviour of the task-level schedulers on a processor that is shared with other applications, which is a more realistic case. Section 5.3 presents the conclusions of this chapter.

5.1 Exclusive application processor

In this section we perform a series of investigations in which the application can use the full processor capacity. Per investigation we vary one characteristic of the application or processor, to see the effects of that characteristic on the throughput of the application. In Section 5.1.1 we investigate the effects of the size of the FIFO buffers in the application. In Section 5.1.2 we investigate a different task execution time distribution for the application. The execution rate of tasks is varied in Section 5.1.3. Section 5.1.4 shows the effects of different slot sizes. In Section 5.1.5 we vary the number of tasks on a processor. Finally in Section 5.1.6 we compare three different topologies for the application.

The default parameters in the investigations are the following: The application consists of four tasks, in a pipeline (see Figure 5.3). The first two tasks are mapped onto the first processor and the last two tasks on the second processor. We only use two task per processor, because calculation with Matlab is very time-consuming with more than two tasks on a processor (with two tasks on a processor Matlab needs about an hour per point in the graph, with three tasks per processor this has increased to a day per point in the graph). The time slots have a size of 20,000 cycles and the operating system slot takes 5,000 cycles. The tasks in the pipeline have an equal execution time
of 100,000 cycles, and thus their requested service is five slots. Each task produces and consumes one token at the time. We want to observe the behaviour of buffer bound applications and therefore the size of the buffers in the application is one.

\[ w_1 = 5 \]
\[ w_2 = 5 \]
\[ w_3 = 5 \]
\[ w_4 = 5 \]
\[ d = 1 \]
\[ d = 1 \]
\[ d = 1 \]

Figure 5.3: Default application for investigations

5.1.1 Buffer size

In this investigation we use the default application where we vary size \( d \) of all buffers as shown in Figure 5.4. We are interested in two different situations: the application is buffer bound or the application is computation bound.

If the application is buffer bound, the throughput of the application is limited by the size of the FIFO buffers. In the formal model a buffer bound application has a critical cycle that includes the initial latency and rate of at least two tasks.

In a computation bound application, the throughput of the application is not limited by the size of the buffers. The task with the slowest execution rate forms the critical cycle of the graph and determines the throughput of the application.

Figure 5.4: Application used for buffer size investigation

In this experiment we demonstrate the difference in throughput between a buffer bound an computation bound application. In Figure 5.5 we show the results of our investigation with the default pipeline application and buffer sizes changing from one to three. Buffer size of one corresponds to a buffer bound application and buffer size of two or more corresponds to a computation bound application.

We first discuss the buffer bound application (buffer size one in our example). The formal models of TDM and CCSP are different for both schedulers, but the predicted throughput is the same. The experiments show that the predictions are conservative for both schedulers.

The application scheduled with TDM delivers 13% higher throughput in the experiment than the formal model predicted. This difference in throughput is caused by the fact that the timing analysis is performed with the WCRTs of tasks, while this WCRT is in
reality not always experienced by the task. The WCRT is calculated from the worst-case moment a task can fire, just after its time slice has finished, to the worst-case moment the task finishes execution. In the experiment this WCRT does not occur because a task is ready to fire when the former task finishes, which means that the time slice of that former task has just finished. This explains the difference in throughput for the formal model and experiment of TDM.

The throughput of the application scheduled with CCSP is 29% better in the experiment than predicted. This is an effect of the fact that the application is buffer bound. Because the application is buffer bound, two consecutive tasks are never able to execute at the same time. A task always has to wait for another task to fill or empty its buffer. When the two task are never available at the same moment, the initial latency of the low priority task in the formal model is never experienced. The low priority task can also use its burstiness credits to start execution at a higher rate than allocated, something that is not predicted by the model. The combination of lower initial latency and higher rate than predicted, explains the difference between the experiments and the timing analysis.

The experimental throughput of the application scheduled with TDM is worse than the experimental throughput of the application scheduled with CCSP. This is because the initial latency for task scheduled with CCSP is smaller than predicted in the experiments, while the initial latency of task scheduled with TDM are accurate (initial
latency for all tasks is zero).

We now discuss the computation bound application (buffer size greater than one in our example). The critical cycle of the application in the formal model consists of the rate of the slowest executing task in the application. The optimal throughput for the application can be reached if each task executes at the same rate. The exhaustive parameter search can find equal rates for the tasks, TDM because it can change the size of the scheduler period and CCSP because it can assign each task the same rate. The experiments for this case show a throughput of 0.5% more than predicted. This is fairly accurate, and the difference is caused by the rounding of the requested service to an integer number of slots. Since the throughput of a computation bound application is given by the rate of the tasks and the model is accurate for computation bound applications, we can conclude that the inaccuracy of the model for buffer bound applications is in the initial latency of the tasks.

In summary we varied the size of the FIFO buffers in the application and discussed two distinct situations: the application is buffer bound or computation bound. We observed that CCSP performs better in the experiments than TDM when the application is buffer bound, while both schedulers yield equal results in the formal timing analysis. The schedulers perform equal in the timing analysis and experiments when the application is computation bound. The formal model of both schedulers is not accurate in predicting the initial latency.

5.1.2 Task execution time distribution

In a real application, the tasks do not have an equal execution time. To find out how the execution time of a task influences the throughput of an application we vary the execution time of one of the tasks as shown in Figure 5.6. During the investigation it became clear that it does not matter which task has a different execution time. Therefore we show the results of changing the execution time of the first task, these results also hold for varying the execution time of one of the other tasks. In Section 5.1.1 we observed that with a computation bound application the throughput of the application is equal for both schedulers in the formal analysis as well as in the experiments and can be accurately predicted with the rate of the tasks in the formal model. Intuitively, we do not expect that we observe differences between the two schedulers with computation bound applications, because both schedulers can set all there parameters freely. The buffer bound application showed more interesting
differences between the formal analysis and the experiments and is therefore the focus of our research.

Figure 5.7: Throughput of an application where Task 1 has various execution times

Figure 5.7 shows the application throughput as function of the execution time of task 1. In this graph we observe two different regions, a flat and a descending one. In the flat region, the execution time of task 1 is smaller or equal to the execution times of the other tasks. The critical cycle of the application is in this case the cycle between the two tasks mapped onto the second processor. Therefore the execution time of task 1 does not influence the throughput of the application.

In the descending region the execution time of task 1 is bigger than the execution time of the other tasks. As a result, task 1 is included in the critical cycle. Because the critical cycle of the graph is longer, the application throughput decreases.

The difference between the predictions of the formal model and the experiments is explained in Section 5.1.1.

In summary we have studied the effects of changing the execution time of one of the tasks on the processor. If the task has a smaller execution time than the others, it does not influence the throughput of the application. If the task has a longer execution time than the others, the throughput of the application decreases.
5.1.3 Task execution rate

The execution rate of a task is fixed by its consumption and production rate. If a task produces more tokens, the task has to run less often that its consuming tasks. A task with a higher consumption rate needs more tokens and the task(s) producing its tokens have to run more often.

To observe the effects of a task with a higher execution rate, we increase the consumption rate $c$ of task 2 (see Figure 5.8). Therefore task 2 has to fire more often than the other tasks. For the reasons mentioned in Section 5.1.2 we only investigate the buffer bound application.

![Figure 5.8: Application used for execution rate investigation](image)

Figure 5.8: Application used for execution rate investigation

![Figure 5.9: Throughput for an application with different consumption rate](image)

Figure 5.9: Throughput for an application with different consumption rate

The results of this investigation can be found in Figure 5.9. Because of the higher
consumption rate, the first processor needs more time than the second processor, to execute its tasks. The tasks on the first processor form the critical cycle and the throughput of the application decreases if the consumption rate grows. The difference between the throughput of the application predicted by the formal model and the experiments can be found in Section 5.1.1.

In summary we have investigated the effects of a higher execution rate for one of the tasks in an application. The throughput of the application decreases with the consumption rate of the task, because the total amount of work in the application is increased.

### 5.1.4 Slot size

The size of the time slots on the processor, influences the requested service of a task (see Figure 5.10). We vary the size of the time slots in the system from 10,000 cycles to 100,000 cycles. The smallest time slot is 10,000 cycles, because this is the minimal size possible on our platform. When the time slot is 100,000 cycles, each task has a requested service of exactly one time slot. Increasing the size of the time slot further, does not decrease the requested service of the tasks, but decreases the throughput of the application.

If a time slot is not used completely, the processor is idle until the end of that time slot (internal slack). A small time slot has the advantage that there can not be much internal slack. The disadvantage is that after each time slot, there is a operating system slot and the overhead is large. A big time slot has much smaller overhead, but the possibility of much more internal slack in the system.

As in Section 5.1.2 we only observe the behaviour of the buffer bound application.

![Figure 5.10: Application used in slot size investigation](image)

The throughput of the application with different sized time slots is presented in Figure 5.11. The best throughput occurs when the execution time of the tasks is an exact multiple of the size of the time slot, otherwise there is internal slack.

For the first time we observe a situation where the formal models of TDM and CCSP do not predict the same throughput. The model for CCSP behaves different, because the initial latency of the low priority task is too pessimistic. Each task needs only one slot to execute and the maximum initial latency experienced by the low priority task is therefore one slot, while the formal model predicts an initial latency of two slots.

The experimental results for both schedulers are equal, because with requested rates of one for all tasks and buffer sizes of one the application is not computation bound,
but buffer bound. The differences between the formal and experimental results of both schedulers are explained in Section 5.1.1.

In summary the optimal slot size determines the balance between overhead and internal slack in the system. The internal slack can be reduced to zero when all tasks can fit their execution time exactly within a multiple number of slots. The overhead depends on the requested service of an application.

### 5.1.5 Number of tasks

Scheduling many tasks on a processor is harder for CCSP than for TDM. Because we look at buffer bound applications, both the initial latency and rate of tasks are included in the critical cycle. The best application throughput can be found when all critical cycles in the graph are equal. TDM can easily reach this optimal throughput by assigning each task the same initial latency and rate. Assigning all task the same initial latency and rate is not possible in CCSP. Therefore the number of tasks on a processor influences the throughput of an application scheduled with CCSP.

Finding the best scheduler parameters for the tasks is very hard, because the number of combinations grows exponentially with the number of tasks. For the investigations with four and five tasks on a processor we assigned the task priorities by hand, otherwise the

<table>
<thead>
<tr>
<th>Slot size</th>
<th>Rate (1/cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc tdm</td>
<td></td>
</tr>
<tr>
<td>calc ccsp</td>
<td></td>
</tr>
<tr>
<td>meas tdm</td>
<td></td>
</tr>
<tr>
<td>meas ccsp</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.11: Throughput with different time slot sizes
computer reached the limits of its memory and could not find optimal parameters. We did not perform the investigation with more than five tasks, to keep the computation time within a day.

![Figure 5.12: Application used for number of tasks investigation](image)

The throughput of the application, shown in Figure 5.12, that consists of a variable number of tasks mapped onto one processor, is presented in Figure 5.13. In the formal model for CCSP the optimum is found if the sum of the initial latencies and rates is equal for all tasks. For the sum to be equal, the task rates and initial latencies of the tasks scheduled with CCSP are unequal. Because TDM can schedule equal initial latencies and rates for the tasks, its application throughput is higher. The experiments with these parameters confirm this result: with more than two tasks on a processor,
CCSP performs worse than TDM.
As we have explained in Section 5.1.1 the latency of tasks in a buffer bound application scheduled with CCSP is much smaller than predicted with the formal model. This means that the optimal result from the formal model is not necessarily the real optimum. We performed another experiment where each task scheduled with CCSP receives an equal allocated rate (blue line in Figure 5.13). This experiment shows that CCSP results in a better application throughput than TDM even, with multiple tasks on a processor. This means that the formal model is inaccurate, and because the results for a computation bound application where accurate, the accuracy of the model is in predicting the initial latency of a task.

From this investigation where the number of tasks on a processor grows we can draw two conclusions. The first is that the CCSP formal model is not accurate in predicting the initial latency of a task. This is observed when we schedule more than two tasks on a processor. The second conclusion is that CCSP can deliver a better throughput than TDM even with multiple tasks on a processor.

### 5.1.6 Different topologies

Until now all investigations featured a pipeline application. This is of course not the only possible topology for an application. In this experiment we compare the three topologies shown in Figure 5.14: a pipeline application, a diverging and converging topology, and independent tasks. In this experiment we look at the buffer bound application (for reasoning why, see Section 5.1.1).

**Figure 5.14:** Three different task graph topologies with three tasks per processor
In Figure 5.15 we present the throughput for the three topologies. The two topologies with dependent tasks perform equal, because all tasks in the application have to execute before the last task can start. Because the independent tasks have no buffers to be buffer bound, these tasks can run at a higher rate than the dependent tasks.

Figure 5.15: Three different configurations. 1 is pipeline, 2 is split and three independent tasks

In summary we have tested three different topologies. The two topologies in which tasks have dependencies perform the same.

5.2 Shared application processor

In the previous investigations an application could use the entire processor. The conclusion was that CCSP performs better than TDM when an application is buffer bound. TDM and CCSP result in the same throughput with a computation bound application.

In the investigations in this section an application shares the processor with other applications. This shows the effects of the combination of the application-level scheduler and task-level scheduler. The application-level scheduler is a TDM scheduler, because it is composable. This application-level scheduler gives each application fixed slices of
the total processor time.
If the task-level scheduler is TDM the execution rate of a task depends on the period, time slice and execution time of the task. The period of the task-level scheduler is fixed by the application-level scheduler. Because the period is fixed, a task cannot receive any execution rate, but a limited set of values.
If the task-level scheduler is CCSP the execution rate of a task depends on the allocated rate, priority and burstiness of the task. The execution rate is therefore not influenced by the fixed size period.

Because the rate of tasks scheduled with TDM is limited by the period of the application-level scheduler and the rate of tasks scheduled with CCSP is not, the throughput of a computation bound application is different for both schedulers. In two investigations, both with a different task execution time distribution, we show the influence of the period of the application-level scheduler on the throughput of the application. The result of these investigations is presented in Section 5.2.1.

5.2.1 Task execution time distribution

In this section we look at the connection between the period of the application level scheduler and the throughput of an application. The applications receive 50% of the application-level scheduler period. We investigate two different task execution time distributions.

The first experiment is performed with an application that consists of a pipeline of three tasks, all situated on one processor, as shown in Figure 5.16. The first two tasks in the application have an execution time of one time slot and the third task has an execution time of a hundred time slots. The throughput of this application is presented in Figure 5.17. As we predicted, the period of the application-level scheduler does not influence the application throughput if the task-level scheduler is CCSP. For the TDM task-level scheduler the fixed period does have a big influence on the application throughput. We observe that only when the length of the period is a multiple of the sum of all task executions, TDM can schedule the application at the same rate as CCSP. When the number of slots needed for all tasks does not fit exactly in the period, TDM performs worse than CCSP. This is logical because TDM has to assign a fixed number of slots per period to a task. If the period grows, even when the period does not fit optimally, TDM can get closer and closer to the optimum.
For the second investigation we use an application of ten tasks in a pipeline, as shown in Figure 5.18. Each task has an equal execution time of five time slots.

![Application consisting of ten tasks with equal execution times]

The application throughput for both CCSP and TDM can be found in Figure 5.19. Again the application-level period has no influence on the CCSP scheduler. The TDM scheduler can assign each task the same rate when the application’s period is a multiple of the number of tasks in the pipeline. Otherwise TDM performs worse than CCSP.

In summary when the effects of both the application-level scheduler and the task-level scheduler are taken into account, CCSP performs better as task-level scheduler than TDM for a computation bound application. Only if TDM can assign all tasks a time slice proportional to their execution time, TDM and CCSP can deliver the same
38

CHAPTER 5. TDM VS. CCSP

Figure 5.19: Fixed application scheduler period and equal sized tasks

5.3 Conclusion

In this section we summarise the results of this chapter. An extensive conclusion is presented in Chapter 6.

In this chapter we have investigated the influences of multiple characteristics of the application and the platform on the throughput of applications scheduled with TDM or CCSP as task-level scheduler. There are two distinct situations: the application is buffer bound or the application is computation bound. When the application is computation bound, the throughput is limited by the execution rate of the slowest task. The throughput of the application can be accurately predicted with the formal model in this case. CCSP can schedule the tasks better than TDM, if the period of TDM is limited such that the tasks cannot execute at the same rate.

When the application is buffer bound, its throughput is limited by the size of its buffers. In this situation the experiments show that CCSP performs better than TDM, while their timing analysis predicts the same throughput for both schedulers. The formal model is not accurate, and since the model was accurate for computation bound application, the inaccuracy of the model is located in the initial latency of the tasks.
Conclusion and future work

In this chapter we present the conclusion and future work of our research. The conclusions can be found in Section 6.1. The future work in Section 6.2 discusses possibilities to extend our research.

6.1 Conclusion

In this thesis we consider a composable multi-processor implemented in FPGA, executing multi-tasking streaming applications. Streaming applications consist of a set of tasks communicating via blocking FIFO buffers. Each task of such an application executes iteratively, indefinitely. Applications are modelled as data-flow graphs and may have real-time rate (or throughput) constraints, i.e. the number of output data produced in a time unit should be higher than a given value. Each processor uses a hierarchical two-level scheduler: an inter-application, or application-level scheduler; and an intra-application, or task-level scheduler. The application-level scheduler is Time Division Multiplexing (TDM), to prevent inter-application interference, and thus create composability. The task-level scheduler might be any scheduler that guarantees that the rate constraint of an application is met.

In this context we compared the achievable rate of two task-level schedulers, Time Division Multiplexing (TDM), and Credit-Controlled Static-Priority (CCSP), on a set of streaming applications. The application rate is a function of the Worst-Case Response Time (WCRT) of its tasks. The WCRT of a task is the time between the moment its input data is available, and the moment it produces its output. TDM is a simple scheduler already implemented on the platform. Its known problems are that (1) it is non work-conserving, and (2) rather rigid, as each task has allocated to it a fixed time slice, on each fixed time period, denoted as allocated rate. If a task cannot use its slice due to data unavailability, that slice is wasted. Thus to decrease the response time of a task, for the purpose of increasing application rate, a large slice has to be allocated to a task each period, leading to potentially large processor time waste.

CCSP is a scheduler originating from the memory controller domain, and is more flexible than TDM in the sense that the response time of a task can be decreased without having to increase its allocated rate. Our comparison has the purpose of (1) determining which task-level scheduler delivers the best application rate, (2) investigating the accuracy of the existing models of these schedulers, and (3) identifying characteristics of streaming applications that would make one or the other scheduler more suitable.

We compared the task-level schedulers both formally and experimentally. For the formal comparison we used dataflow models of the scheduled applications. To determine
the highest rate of an application scheduled with TDM or CCSP, we performed an exhaustive search through all combinations of schedulers’ parameters. The experimental comparison was performed on an existing FPGA platform, with the scheduler parameters resulting from the exhaustive search.

We employed existing dataflow models for an application scheduled with TDM or CCSP. The dataflow model of an application scheduled with TDM is relatively simple, and finding the scheduler parameters for the highest rate has a cost that is linear with the number of tasks on the processor. For CCSP both the dataflow model and finding the best scheduler parameters are more difficult. Because of the combination of credits and priorities to decide which task should be scheduled, changing the rate or priority of one task immediately changes the behaviour of the other tasks. This makes the cost for finding the parameters for highest rate exponential with the number of tasks. Creating a dataflow graph model for CCSP that is accurate enough to find optimal rate, but also conservative enough to never miss deadlines, is hard. The CCSP model for a task can be one of the latency-rate, serial bi-rate, or non-serial bi-rate models. The differences between these three models are in accuracy and cases in which they are conservative. Thus each task is modelled with the most accurate but still conservative model from the three above. The analysis is costly because of the complexity of the models and the large number of possible parameters combinations, hence in some cases the application sizes (number of tasks) that could be studied is rather small.

During our investigation we realised that buffer sizes are a relevant application parameter. If the buffer sizes are small, the rate of an application is limited by several tasks. We denote this type of application as buffer bound. If the buffer sizes are large, the rate of an application is limited by a single task. We denote this type of application as computation bound. Thus in a computation bound application the maximum achievable rate is given by rate of the slowest task.

We investigated two different cases: (1) when an application can use the entire processor and (2) when the application shares the processor with other applications. As default we used an application consisting of four tasks, configured as a pipeline topology. The first two tasks of this application where mapped on one processor, the last two tasks on another processor. The time slots where 20,000 cycles, the operating system slot took 5,000 cycles, each task in the application has an execution time of 100,000 cycles, and therefore the requested service of each task is five slots. All tasks produce and consume one token and the size of the buffers is one. In this default application we changed one characteristic at the time, to observe its effect on the application rate. The characteristics are: (1) the size of the buffers, (2) the execution time distribution of the tasks, (3) the execution rate of the tasks, (4) the size of the time slots on the processor, (5) the number of applications mapped onto one processor, and (6) the topology of the application. The observations made when changing these characteristics are:

1. For a computation bound application, the formal models predict the rate of the application accurately, and both scheduler yield equal application throughput. This is not the case for a buffer bound application, where the application rate is higher in the experiments then in the timing analysis. The reason for this behaviour is the
size of the buffers, which make it impossible for two neighbouring tasks to execute at the same moment, therefore the initial latency for tasks scheduled with CCSP and the WCRT of tasks scheduled with TDM is inaccurately predicted. With a buffer bound application, CCSP reaches a higher experimental application rate than TDM.

2. When one task in an application has a longer execution time than the others, the longest task is part of the critical cycle and therefore gives the application rate. Since the application is buffer bound, CCSP performs better than TDM in the experiments, and the predicted application rate is not accurate.

3. A task with a high production rate runs less often than its consuming tasks. A task with a high consumption rate runs less than its producing task. The processor tile that contains the biggest workload, gives the application throughput. CCSP performs better than TDM, and the models are not accurate, since the application is buffer bound.

4. The size of the time slots influences the overhead and possible internal slack in the system. The overhead of the system is the smallest with big time slots, because a task needs less time slots to execute and thus there are less operating time slots needed. The possibility for much internal slack in the system is higher with a big time slot, because a task can finish after a short time and leave the rest of the time slot idle. In the experiments we observed that the best application rate can be achieved, when all tasks execution times are an exact multiple of the slot size. CCSP is able to use more of the available time slots than TDM, and therefore gives a higher application rate.

5. When scheduling more tasks on a processor, it is more difficult to find optimal parameters for CCSP with the formal model as the required computational effort is exponential in the number of tasks. The investigation showed that CCSP performed worse than TDM with more than two tasks on a processor, both in the experiments and timing analysis. Since we have observed before, that the initial latency of a task is not predicted accurately, we did not trust that the parameters found from the formal model where indeed the optimal parameters for the scheduler. Therefore we performed another experiment with CCSP where we assigned each task the same allocated rate and ignored the latency effects. This experiment showed that CCSP performed better than TDM even when there are more than two tasks on a processor. Therefore we can conclude that the formal model of CCSP is not accurate enough to provide a fair comparison with the results from TDM, when there are more than two tasks mapped onto a processor.

6. In this investigation we compared three applications consisting of six tasks, mapped onto three processors: (1) independent tasks, (2) an application with a diverging and converging topology and (3) a pipeline application. We observed that the two applications with data dependencies between the tasks performed the same, because all tasks had to execute before the last task could start execution. Since
the independent tasks can not be buffer bound, their rate was higher than the rate of the other applications.

In the case when an application shares the processor with other applications, the period of the application is set by the application-level TDM scheduler. We use a computation bound application for these experiments, and the formal model predicts the results of the experiments very accurately. The length of the period of the application-level scheduler, influences the rate of an application scheduled with TDM, because in TDM each task has assigned a fraction of the period, and that fraction determines the task’s response time. CCSP is not a periodic scheduler, thus the length of application-level period does not influence an application’s rate. Our experiments and timing analysis suggest that TDM performs best if each task gets a slice of the period proportional to its execution time. In this case TDM equals the performance of CCSP when the application is computation bound; in all other cases CCSP performs better.

When the application has exclusive processor usage, TDM and CCSP yield the same rate when the application is computation bound and CCSP achieves higher rate when the application is buffer bound. With a shared processor TDM can reach the same rate as CCSP, if the following conditions are met: (1) the application is computation bound, and (2) the application-level period is large enough such that the TDM scheduler can assign each task a slice proportional to its execution time. In all other cases CCSP performs better.

6.2 Future work

In this section we present some ideas for continuing our work, in two directions: (1) improve on the task-level scheduling formalism, and the generality of the experiments, and (2) improve the energy efficiency of the system, taking into account the task-level scheduler.

In some of our experiments, e.g., in those with more than two tasks on a processor, we have observed that the formal model for CCSP is not accurate enough to find the scheduler parameters for optimal rate. Thus it is hard to compare TDM and CCSP based only on their formal models. This inaccuracy creates difficulties in deciding which experiments should be performed. The non-serial bi-rate model is fairly accurate in the rate prediction, something that is proved by the fact that the experimental results with computation bound applications are very close to the predicted results. However, as we have seen in the investigations with a buffer bound application, the weak point of this model is the accurate prediction of the latency of a task, thus a potential research direction is in modelling the latency of a task.

Due to the model’s complexity, the exhaustive search for the CCSP scheduler parameters (priority, burstiness, rate) that give the highest application rate require a large computational effort. A (heuristic) method that decreases this computation effort would make more experiments possible, e.g., investigations of applications with more
than two tasks mapped per processor. This investigations would reveal if CCSP continues to perform better than TDM when more tasks have to be scheduled on a processor.

Our research mainly targets an application with a pipeline structure. We performed one experiment with a task topology different than a pipeline. For this experiment we have reasons to believe that the task topology does not influence the results of the scheduler, as long as all tasks have dependencies. Since this is not tested extensively, it could be worthwhile to look into more different topologies to support or dismiss this conclusion.

When an application could use a fraction of the processor, the experiments seem to indicate that any scheduler that can produce an allocation in which tasks get a fraction of the processor proportional with their execution times, is able to achieve the optimal rate of that application. However these suppositions should be investigated in detail.

In all investigations in this thesis, the tasks have a fixed production and consumption rate. In streaming applications a task could produce or consume a variable number of tokens, based on the input data. This variability influences the performance of the scheduler, because it is not possible anymore to create a single static order schedule for the application that produces optimal rate. We think it is possible that CCSP can handle this variability better than TDM, but this should be researched.

Our research compared the non work-conserving task-level schedulers. The created schedule contains a lot of idle processor time (slack). There are two different types of slack, internal and external slack. Internal slack is idle time within a time slot, because a task finishes execution before the time slot ends. This internal slack could be removed by allowing to assign the remaining part of a time slot to another task. This makes it possible to run multiple tasks within one time slot, without increasing the overhead of os-slots. External slack are complete time slots in which no task is scheduled. With TDM this can occur when a task can not fire when its time slice is reached and with CCSP when a task has not enough credits to execute. This external slack can be removed by calling a slack manager in the operating system time slot. Whenever the schedulers algorithm cannot find a task to execute, the slack manager could schedule any of the available tasks according to its algorithm. Different slack manager algorithms are possible. We have already performed some initial experiments with a fair slack manager, that picks the task that can run and has received the least slack slots until now. These initial experiments show a rate improvement of approximately 10% when a slack manager is used and are therefore very promising.

In our research we aim to maximise the rate of an application. When the energy consumption of the application is the most important requirement and only a minimum rate is set, the slack could be utilised for energy saving. In this case we could look at voltage frequency scaling methods, that reduce the voltage and frequency of the processor so the application just reaches its deadline, saving energy.


List of abbreviations

This list explains the abbreviations used in this thesis.

CCSP       Credit-Controlled Static-Priority
CompOSe    Composable Operating System Extreme
CSDF       Cyclo Static Dataflow
DDF        Dynamic Dataflow
FIFO       First-In First-Out
FPGA       Field Programmable Gate Array
HSDFG      Homogeneous Synchronous Dataflow Graph
LR         Latency-Rate
MCM        Maximum Cycle Mean
MPSoC      Multi-Processor System on Chip
MRDF       Multirate Dataflow
NoC        Network on Chip
RM         Rate Monotonic
RT         Response Time
SDFG       Synchronous Dataflow Graph
SRDF       Single-Rate Dataflow
TDM        Time Division Multiplexing
WCET       Worst-Case Execution Time
WCRT       Worst-Case Response Time
APPENDIX A. LIST OF ABBREVIATIONS
List of symbols

This table shows all symbols used in this thesis.

Table B.1: List of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g$</td>
<td>simple cycle in dataflow graph</td>
</tr>
<tr>
<td>$C(G)$</td>
<td>set of all simple cycles of dataflow graph $G$</td>
</tr>
<tr>
<td>$d$</td>
<td>delay or initial number of tokens on an edge in a dataflow graph</td>
</tr>
<tr>
<td>$E(g)$</td>
<td>set of all edges traversed by cycle $c$</td>
</tr>
<tr>
<td>$h$</td>
<td>delay on an edge between the high-rate and low-rate actor in the serial bi-rate model and non-serial bi-rate model</td>
</tr>
<tr>
<td>$p$</td>
<td>period of Time Division Multiplexing time wheel (in time slots)</td>
</tr>
<tr>
<td>$r$</td>
<td>response time of a task (in time slots)</td>
</tr>
<tr>
<td>$s$</td>
<td>size of time slice of a task in Time Division Multiplexing time wheel (in time slots)</td>
</tr>
<tr>
<td>$V(g)$</td>
<td>set of all actors traversed by cycle $c$</td>
</tr>
<tr>
<td>$w$</td>
<td>requested service of a task (in time slots)</td>
</tr>
<tr>
<td>$W$</td>
<td>Worst-Case Execution Time of a task (in cycles)</td>
</tr>
<tr>
<td>$X^+_x$</td>
<td>set of all tasks with a higher priority than task $x$</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>length of a time slot (in cycles)</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Maximum Cycle Mean of a dataflow graph</td>
</tr>
<tr>
<td>$\rho$</td>
<td>execution rate of a task (1/time slots)</td>
</tr>
<tr>
<td>$\rho'$</td>
<td>allocated rate of a task in Credit-Controlled Static-Priority (1/time slots)</td>
</tr>
<tr>
<td>$\rho^*$</td>
<td>high-rate of a task in the serial bi-rate and non-serial bi-rate model of Credit-Controlled Static-Priority (1/time slots)</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>burstiness of a task in Credit-Controlled Static-Priority (in time slots)</td>
</tr>
<tr>
<td>$\theta$</td>
<td>initial latency of a task (time slots)</td>
</tr>
<tr>
<td>$\phi$</td>
<td>priority of a task</td>
</tr>
</tbody>
</table>
CCSP implementation

This appendix shows the C-code of the CCSP task-level scheduler as implemented on our platform.

Listing C.1: Structure added to file os_types.h

typedef struct CCSP_s CCSP;

struct CCSP_s{
    int nr_tasks;
    int * task_order;
    int * priority;
    // Allocated rate of a task is numerator/denominator
    int * numerator;
    int * denominator;
    int * burstiness;
    int * credits;
};

Listing C.2: File os_task_scheduler_ccsp.h

#ifndef __OS_TASK_SCHEDULER_CCSP_
#define __OS_TASK_SCHEDULER_CCSP_

#include "os_types.h"

TCB* task_scheduler_ccsp(ACB * app);
#endif

Listing C.3: File os_task_scheduler_ccsp.c

#include "os_global.h"

TCB* task_scheduler_ccsp(ACB* app){
    int i , can_fire;
    TCB * itr_task , * sel_task=NULL;
    int sel_task_loc = 0; // location of selected task in ccsp_args arrays
    CCSP * ccsp_args = app->task_scheduler_args;

    static int initial = 1;
    if (initial==1) {
        // Give all tasks initial burstiness credits
        for (i = 0 ; i < ccsp_args->nr_tasks ; i++) {
            ccsp_args->credits[i] = ccsp_args->burstiness[i] * ccsp_args->denominator[i];
        }
    }

    //...
for (i = 0 ; i < ccsp_args->nr_tasks ; i++) {
    itr_task = os_get_task(ccsp_args->task_order[i], app->id);
    can_fire = 0;
    if ((itr_task->state == TASK_IDLE) || (itr_task->state==TASK_FINISH)) {
        can_fire = os_check_task_fr(itr_task);
    } else {
        can_fire = 1; // Task is preempted so it should fire
    }
    if (can_fire) {
        ccsp_args->credits[i] = ccsp_args->credits[i] + ccsp_args->numerator[i];
        if (ccsp_args->credits[i] >= ccsp_args->denominator[i]) {
            // Task has enough credits to run
            if (sel_task==NULL || ccsp_args->priority[i] < ccsp_args->priority[sel_task_loc]) {
                sel_task = itr_task;
                sel_task_loc = i;
            }
        } else {
            ccsp_args->credits[i] = ccsp_args->burstiness[i] + ccsp_args->denominator[i];
        }
    }
    // Deduct credits for selected task, if it is not the idle task
    if (sel_task==NULL) {
        sel_task=os_get_task(0, app->id);
    } else {
        ccsp_args->credits[sel_task_loc] = ccsp_args->credits[sel_task_loc] - ccsp_args->denominator[sel_task_loc];
    }
    return sel_task;
}
Curriculum Vitae

Aster Leegwater was born in Haarlem, The Netherlands, on August 12th of 1986. She grew up in Hoofddorp and completed her secondary education at the Atheneum College Hageveld in Heemstede. For her final exam in chemistry she received a prize because her mark was nationally the highest. In 2004 she started a bachelor Electrical Engineering at the Delft University of Technology. In this bachelor she included a minor Computer Science. She finished this bachelor with honours in 2007.

In the second year of her bachelors she moved to Delft and became an active participant of two student societies: DSMG Krashna Musika and DSZ Wave. This included playing the cello and double bass at the symphony orchestra of Krashna, swimming at Wave and committee work for both. Aster was also a student-assistant for various courses in programming and electrical engineering basics for the bachelors Electrical Engineering, Computer Science, Industrial Design and Airspace Engineering.

From September 2007 to September 2008 Aster paused her study to be the full-time treasurer of Krashna Musika. This year was the 40st birthday of Krashna and included a cooperation project with a choir from Leiden, a tour to Germany and Denmark and a big concert on the Grote Markt in Delft.

In September 2008 Aster started with the master program Embedded Systems at the TU Delft. During this program she did an internship of three months at Technolution in Gouda, where she researched the possibilities of using gsm base stations to track a route.

Aster likes to play the cello and double bass, learns ballroom dancing, reads lots of books and loves to play board games with friends.