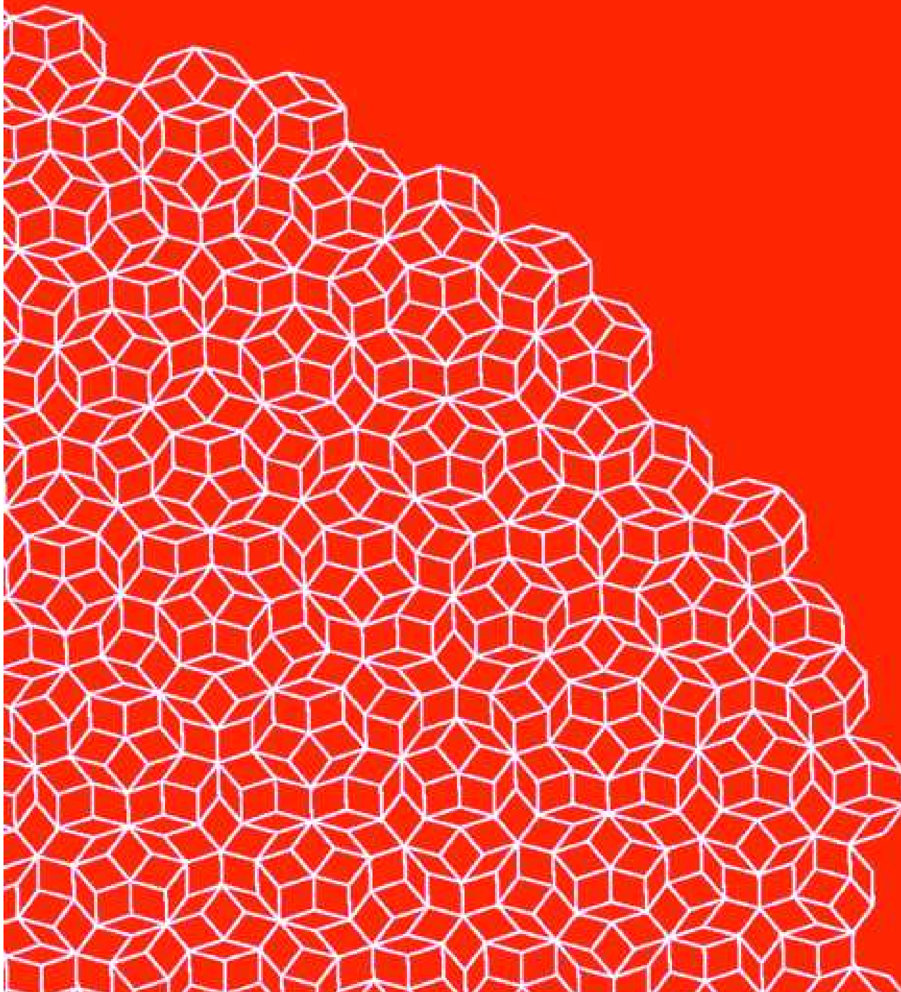
APERIODIC MULTIPROCESSOR SCHEDULING

for REAL-TIME STREAM PROCESSING APPLICATIONS



maarten wiggers

APERIODIC MULTIPROCESSOR SCHEDULING for REAL-TIME STREAM PROCESSING APPLICATIONS

Members of the dissertation committee:

Prof. dr. ir.	G.J.M. Smit	University of Twente (promotor)
Dr. ir.	M.J.G. Bekooij	NXP Semiconductors (assistant promotor)
Prof. dr. ir.	B.R. Haverkort	University of Twente
		Embedded Systems Institute
Prof. dr.	J.C. van de Pol	University of Twente
Prof. dr. ir.	A.A. Basten	Eindhoven University of Technology
		Embedded Systems Institute
Prof. dr.	E.A. Lee	University of California at Berkeley
Prof. dr.	S. Chakraborty	Technical University of München
Prof. dr. ir.	A.J. Mouthaan	University of Twente (chairman)

This research was conducted in a project of Philips Research, later NXP Semiconductors Research, and was supported by Philips Electronics and NXP Semiconductors. This work contributed to the Centre for Telematics and Informatics (CTIT) research program.

Copyright © 2009 by Maarten H. Wiggers, Eindhoven, The Netherlands

All rights reserved. No part of this book may be reproduced or transmitted, in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without the prior written permission by the author.

This thesis was edited in Emacs and typeset with LaTeX 2_ε. The cover shows an aperiodic tiling. Credits for the cover design go to Rob van Vught. This thesis was printed by Gildeprint, The Netherlands.

ISBN 978-90-365-2850-4
ISSN 1381-3617, CTIT Ph.D.-thesis series No. 09-146
DOI 10.3990/1.9789036528504

APERIODIC MULTIPROCESSOR SCHEDULING for REAL-TIME STREAM PROCESSING APPLICATIONS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Friday, June 19, 2009 at 16:45

by

Maarten Hendrik Wiggers

born on 1 July 1981
in Almelo, The Netherlands

This thesis is approved by:

Prof. dr. ir.	G.J.M. Smit	(promotor)
Dr. ir.	M.J.G. Bekooij	(assistant promotor)

Abstract

This thesis is concerned with the computation of buffer capacities that guarantee satisfaction of timing and resource constraints for task graphs with aperiodic task execution rates that are executed on run-time scheduled resources.

Stream processing applications such as digital radio baseband processing and audio or video decoders are often firm real-time embedded systems. For a real-time embedded system, guarantees on the satisfaction of timing constraints are based on a model. This model forms a load hypothesis. In contrast to hard real-time embedded systems, firm real-time embedded systems have no safety requirements. However, firm real-time embedded systems have to be designed to tolerate the situation that the load hypothesis is inadequate. For stream processing applications, a deadline miss can lead to a drastic reduction in the perceived quality, for instance the loss of synchronisation with the radio stream in case of a digital radio can result in a loss of audio for seconds. For power and performance reasons, stream processing applications typically require a multiprocessor system, on which these applications are implemented as task graphs, with tasks communicating data values over buffers.

The established time-triggered and event-triggered design paradigms for real-time embedded systems are not applicable. This is because time-triggered systems are not tolerant to an inadequate load hypothesis, for example non-conservative worst-case execution times, and event-triggered systems have no temporal isolation from their environment. Therefore, we introduce our data-driven approach. In our data-driven approach, the interfaces with the environment are time-triggered, while the tasks that implement the functionality are data-driven. This results in a system where guarantees on the satisfaction of the timing constraints can be provided given that the load hypothesis is adequate. If the load hypothesis is inadequate, then for instance non-conservative worst-case execution times do not immediately result in corrupted data values and undefined functional

behaviour.

Stream processing applications are increasingly adaptive to their environment, for example digital radios that adapt to the channel condition. This adaptivity results in increasingly intricate sequential control in stream processing applications. The implementations of stream processing applications as task graphs, consequently, have inter-task synchronisation behaviour that is dependent on the processed data stream. Currently, cyclo-static dataflow is the most expressive model that is applicable in our data-driven approach and that can provide guarantees on the satisfaction of timing constraints. However, cyclo-static dataflow cannot express inter-task synchronisation behaviour that is dependent on the processed data stream. Boolean dataflow can express inter-task synchronisation behaviour that is dependent on the processed data stream. However, for boolean dataflow, and models with similar expressiveness, deadlock-freedom is an undecidable property, and there is no known approach to conservatively decide on deadlock-freedom. Since deadlock-freedom guarantees progress, the ability to (conservatively) decide on deadlock-freedom is necessary to guarantee satisfaction of timing constraints.

A second trend is that stream processing applications increasingly process more independent streams. Typically, timing constraints are specified per stream. We apply on every shared resource a run-time scheduler that by construction guarantees every task a resource budget. These schedulers allow to provide timing guarantees per stream that are only dependent on the load hypothesis for the processing of this stream. However, currently, there is only limited support for the inclusion of the effects of run-time scheduling in dataflow graphs in order to provide guarantees on the satisfaction of the timing constraints for this stream.

The programming of stream processing applications on embedded multiprocessor systems involves the partitioning of the application in a task graph, binding tasks to processors and buffers to memories, selection of scheduler settings, and determination of buffer capacities. All these decisions together should result in a configuration for which we can guarantee that the timing constraints of the application are satisfied. The determination of buffer capacities that are sufficient to guarantee satisfaction of the timing constraints is an essential kernel of automated programming flows for stream processing applications. However, currently, buffer capacities are determined with dataflow analysis by iterating through the possible buffer capacities and for every selection of buffer capacities analyse whether the timing constraints are satisfied. Both the iteration as well as the analysis have an exponential complexity in terms of the graph size.

This thesis presents an algorithm that uses a new dataflow model, variable-rate phased dataflow, to compute buffer capacities that guarantee satisfaction of timing and resource constraints for run-time scheduled task graphs that have inter-task synchronisation behaviour that is dependent on the processed data stream.

Variable-rate phased dataflow allows to model task graphs with inter-task synchronisation behaviour that is dependent on the processed stream, examples include DRM and DAB digital radio, MP3 decoding and H.263 video decoding. Previously, no techniques were available to guarantee the satisfaction of timing constraints for this class of applications. Furthermore, we show that the effects of run-time schedulers can be conservatively included in variable-rate phased dataflow, given that by construction these run-time schedulers provide every task a resource budget. These two essential extensions together with the low run-time and low computational complexity of our algorithm enable automated programming flows for a significantly broader class of applications and architectures.

Acknowledgements

This work would not have been possible without the support of the following people, for which I would like to thank them.

My daily supervisor and assistant promotor Marco Bekooij. We worked so closely together that it is not always clear who contributed which piece to the presented approach. With his ability to keep one eye on the big picture of academic and practical relevance and the other eye on the nitty gritty technical details, Marco has been a most excellent supervisor.

My promotor Gerard Smit. At every moment in the past four years, Gerard expressed confidence in the research direction and provided me freedom in pursuing my research.

My co-workers at Philips Research and later NXP Semiconductors Research. I would like to mention: Andreas Hansson, Benny Åkesson, Arno Moonen, Tjerk Bijlsma, Aleksandar Milutinović, Orlando Moreira, and Kees Goossens. Together forming a coherent research team, in every sense of the word. Further the role of Jef van Meerbergen in the initial phase of this research effort and in past research projects on which this research is based is easily underestimated. Marcel Steine made an important contribution to the research results in his master project. Furthermore, the following co-workers have played an important role in making my research possible: Marcel Pelgrom, Pieter van der Wolf, Albert van der Werf, and Ad ten Berg. The students and staff of the Embedded Systems cluster at the University of Twente who always provide the most pleasant atmosphere. The graduation committee members and reviewers of my work with their constructive and stimulating feedback.

My friends who always provide a most pleasant distraction from the everyday work-rhythm. My parents, Bert and Margreet, my brothers and other relatives who are all always there to give all support. Above all, my fiancée Anna, thank you so much for your endless support and for bringing so much love and happiness in my life.

Maarten, Eindhoven, May 2009

Contents

Abstract **v**

Acknowledgements **ix**

1 Introduction **1**
1.1 Stream Processing Applications 2
1.2 Embedded Multiprocessor Programming 7
1.3 Problem Statement 9
1.4 Contributions 10
1.5 Approach 11
1.6 Outline 12

2 Related work **13**
2.1 Time-Triggered Scheduling 14
2.2 Event-Triggered Scheduling 16
2.3 Data-Driven Scheduling 22
2.4 Conclusion 26

3 Synchronisation and Arbitration Requirements **27**
3.1 Synchronisation Requirements 28
3.2 Execution Times 35
3.3 Arbitration Requirements 38
3.4 Timing Constraints 46
3.5 Resource Constraints 47
3.6 Conclusion 48

4 Conservative Dataflow Simulation and Analysis **49**
4.1 Functionally Deterministic Dataflow 50
4.2 Conservative Dataflow Modelling 57
4.3 Dataflow Simulation 59

4.4 Dataflow Analysis . . . 60
4.5 Applying Dataflow Analysis . . . 63
4.6 Conclusion . . . 71

5 Modelling Run-Time Scheduling **73**
5.1 Modelling Run-Time Scheduling with Response Times . . . 74
5.2 Modelling Run-Time Scheduling with Latency and Rate . . 82
5.3 Dataflow Analysis with Latency and Rate Model . . . 86
5.4 Accuracy Evaluation . . . 94
5.5 Conclusion . . . 100

6 Computation of Aperiodic Multiprocessor Schedules **101**
6.1 Introduction . . . 102
6.2 Graph Definition . . . 112
6.3 Buffer Capacities . . . 117
6.4 Unbounded Iteration . . . 143
6.5 Modelling Run-Time Scheduling . . . 162
6.6 Experiment . . . 164
6.7 Conclusion . . . 166

7 Case Study **167**
7.1 Execution in Isolation . . . 169
7.2 Execution on Shared Resources . . . 175
7.3 Data-Dependent Inter-Task Synchronisation . . . 180
7.4 Conclusion . . . 183

8 Conclusion **185**
8.1 Summary . . . 186
8.2 Contributions . . . 188
8.3 Outlook . . . 189

List of Symbols **191**

Bibliography **195**

List of Publications **205**

Chapter 1

Introduction

This thesis is concerned with the computation of buffer capacities that guarantee satisfaction of timing and resource constraints for task graphs with aperiodic task execution rates that are executed on run-time scheduled resources. Stream processing applications are often implemented on a multiprocessor system as task graphs. The increasing adaptivity to their environment of these applications results in input-data dependent, i.e. aperiodic, task execution rates. Because tasks have aperiodic execution rates and applications process multiple independent streams, run-time scheduling of tasks is applied.

This thesis presents a new dataflow model that can model task graphs with aperiodic task execution rates together with an algorithm that efficiently computes buffer capacities for the modelled task graph that satisfy timing and resource constraints. Furthermore, we show that this dataflow model can accurately include the effects of run-time scheduling. The computation of sufficient buffer capacities is intended to be part of a design flow to program applications on a given multiprocessor system such that timing constraints are satisfied.

The outline of this chapter is as follows. In Section 1.1, we will discuss our application domain of stream processing applications. A positioning of buffer capacity computation in the larger problem to program applications on a given multi-processor system is provided in Section 1.2. Section 1.3 more precisely describes the buffer capacity computation problem addressed in this thesis, while Section 1.4 discusses the contributions of this thesis. Section 1.5 presents our general approach. In Section 1.6, an outline of our approach is provided together with the outline of this thesis.

1.1 Stream Processing Applications

Embedded systems are computing systems for which the physical environment with which they interact forms an integral part of their design. Since the physical environment is inherently temporal, the metric properties of time play an essential part of the functionality of embedded systems (Lee 2005; Henzinger and Sifakis 2007).

Firm Real-Time Embedded Systems Embedded systems react to events in their physical environment. The computation of this reaction requires time. An embedded system can be designed such that (1) given a hypothesis on the time required to compute the reaction we can reason and show beforehand that the system produces its reaction on time, or (2) there is no such hypothesis and the timely response of the system is only tested after the complete system is ready for deployment in its environment (Kopetz 1997). We call systems that need to be designed following the first paradigm real-time systems, and we call systems designed following the second paradigm best-effort systems. The adequacy of the design of real-time systems reduces to the probability that the hypothesis on the time required to compute a reaction, called load hypothesis, is conservative, while for best-effort systems the adequacy of the design is based on a probabilistic argument that all relevant cases have been tested (Kopetz 1997). In this thesis, we focus on real-time embedded systems.

Within the domain of real-time embedded systems, we see that the inadequacy of the load hypothesis has different consequences. There are systems for which a too late response can have catastrophic consequences to the physical environment, and there are systems for which a too late response is perceived as a (severe) quality degradation. The first type of system involves safety and is called a hard real-time embedded system, while the second type of system is called a firm real-time embedded system. Hard real-time embedded systems can be found in cars, airplanes, and power plants. Firm real-time embedded systems can be found in consumer electronics. For example, a too late response by the embedded system can imply loss of synchronisation with a radio stream resulting in a severely degraded experienced quality of the system, because there is for instance no audio for a number of seconds.

The design of hard and firm real-time systems is different. For a hard real-time system the load hypothesis should hold with probability one and a fault hypothesis is required that describes the assumed faults that can occur. This implies a proof obligation on the load hypothesis, and, furthermore, implies measures to have the system tolerate the faults as specified in the fault hypothesis. For a firm real-time system, we strive to let the load hypothesis hold with a high probability, but include measures to have the system tolerate the situation that the load hypothesis is inadequate. For firm real-time systems, we have no further fault hypothesis apart from

the hypothesis that the load hypothesis can be inadequate.

Properties and Trends in Stream Processing Applications In this thesis, we focus on firm real-time embedded systems that process multiple streams of data. An example system is a car-entertainment system in which concurrently different audio streams and also video streams are processed. We say that a stream is processed by a job. Each job has its own requirements on responsiveness to events in the physical environment and its own load hypothesis. We see four trends in this application domain.

1. increasing computational requirements of jobs
2. increasing adaptivity to the environment of jobs
3. increasing integration of jobs in one system
4. increasing context dependent resource provisions by architecture

Aiming to provide an increasingly improving quality to the user, stream processing jobs have increasing computational requirements. For jobs that process audio streams, this is for example, because of a greater fidelity of the audio, and more independent audio tracks. In case of video, we have larger displays, with higher resolutions and more accurate display of colours. In order to provide this increasing quality, signals with a larger bandwidth need to be processed with increasingly advanced (post-)processing algorithms. Both lead to a larger computational requirement of stream processing jobs.

In order to communicate these signals with increasing bandwidth over physical communication channels with limited bandwidth, stream processing applications are increasingly more adaptive to the communicated signal, applying aggressive compression techniques to reduce their bandwidth, and to the conditions of the physical channel, adapting for instance the modulation scheme. Examples of stream processing with aggressive compression techniques include MP3 and AAC audio encoding, and H.263 and H.264 video decoding applications. Digital radios often have various adaptation schemes to adapt to changing channel conditions, examples include DAB and DRM digital radio. This increasing adaptivity leads to an increasingly intricate and growing part of the stream processing job for control processing.

The increasing integration of jobs in one stream processing system is driven by a quest to reduce manufacturing and design cost by resource sharing, and by a quest to provide ever more features to the user. To reduce manufacturing and design cost, the functionality that was previously provided by separate chips on a board that each had their own external memory is increasingly provided by a single chip with a single external memory. On top of this, users expect an ever increasing feature set, thereby

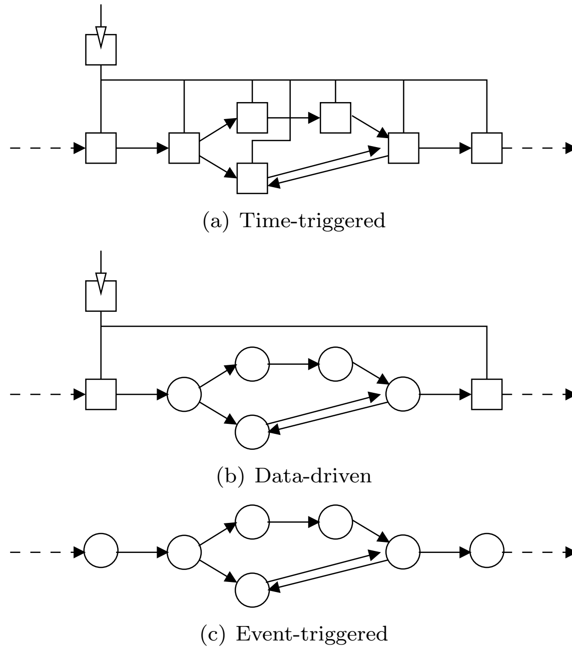
requiring more functionality to be implemented by the stream processing system.

Architectural components such as processors, interconnect, and memories provide an amount of resources that is dependent on an increasingly larger state, i.e. dependent on a longer history and dependent on more aspects of their usage. Examples include, deep pipelines, branch prediction, caches, and banked external memory. Even though these techniques do increase the performance over large enough time intervals, it is difficult and sometimes impossible to model their effect in a load hypothesis, because of dependencies on the processed data stream and because of dependencies on other jobs that share this resource.

Addressing the Trends The increasing computational requirements of stream processing jobs combined with a virtually constant maximum power dissipation requirement is addressed by the application of heterogeneous multiprocessor systems. On such a multiprocessor system, the stream processing job is implemented in a distributed fashion as a graph of tasks that communicate values over buffers. In order to prevent data races proper inter-task synchronisation needs to be added to the task graph. We assume that jobs are specified to deterministically map input values to output values. To prevent that we need to specify, understand and verify the functionality of a task graph in which the output values depend on the schedule of the task graph, we require that the implementation of a job as a task graph is functionally deterministic.

There are two trends that lower the probability that the load hypothesis of jobs that process streams of data is adequate, (1) stream processing jobs are increasingly adaptive to their environment, and (2) the amount of resources provided by current architectures is increasingly context dependent. This increasing number of states in the functionality and the architecture makes it increasingly difficult to find the input values and initial state of the architecture that together lead to the worst-case execution time. In the domain of stream processing jobs, we, therefore, see that for many jobs determination of worst-case execution times is no longer practical.

A successful approach to the design of real-time embedded systems is the so-called time-triggered paradigm. In the time-triggered paradigm tasks are periodically triggered by timers. Both architectures (Kopetz 1997) as well as programming approaches (Benveniste et al. 2003; Henzinger et al. 2003) exist to support the time-triggered paradigm. However, this paradigm relies on worst-case execution times, since the functional behaviour is not specified in case the load hypothesis is not adequate, and the task execution is not finished before the next trigger arrives. Another approach to the design of real-time embedded systems is the so-called event-triggered paradigm. In the event-triggered paradigm all tasks, also the interfaces to the environment, are triggered by the arrival of events.



(a) Time-triggered

(b) Data-driven

(c) Event-triggered

Figure 1.1: Three paradigms in real-time embedded systems.

An extensive body of literature is available to reason about these systems (Sha et al. 2004; Butazzo 1997; Jersak et al. 2005; Haid and Thiele 2007). The event-triggered paradigm relies on a conservative characterisation of the events from the environment and furthermore relies on worst-case and best-case execution times to derive a conservative characterisation of the output events. We introduce our data-driven paradigm which deviates from these two paradigms by letting the tasks that implement the stream processing job be triggered by events, i.e. the arrival of data, while the interfaces with the environment are still time-triggered. Time-triggered input interfaces ensure that the job is temporally isolated from its environment, while time-triggered output interfaces remove all jitter introduced by the job executing on the multiprocessor system.

These three paradigms are illustrated in Figure 1.1. In the time-triggered system, shown in Figure 1.1(a), all tasks are triggered by clocks that are derived from a single master-clock. In the data-driven system, shown in Figure 1.1(b), the interfaces are triggered by clocks that are derived from a single master-clock, but the tasks that implement the functionality of the job are triggered by the arrival of data. In the event-triggered system, shown in Figure 1.1(c), all tasks are triggered by events.

In a time-triggered system, tasks are triggered according to an at design-time computed periodic schedule. We will show in this thesis, that for a data-driven system it is sufficient to show at design time that a valid schedule exists such that the time-triggered interfaces can always produce and consume their data. Since we only need to show existence of a schedule, we can reason in terms of a worst-case schedule that bounds the schedules, i.e. arrival times of data, that can occur in the implementation. As a consequence, tasks in a data-driven system can execute aperiodically, while satisfying timing constraints. These tasks execute aperiodically as a result of varying execution times and inter-task synchronisation behaviour that is dependent on the processed data stream.

In a data-driven system, the tasks that implement the stream processing job are triggered on the arrival of data. Therefore, in such a data-driven system, data is not corrupted or lost in the buffers over which these data-driven tasks communicate. This is also the case if the execution time of a task exceeds an unreliable worst-case execution time estimate. This implies that even when the schedule that was derived at design time does not pessimistically bound all data arrival times, then this does not necessarily imply corruption of data in the implementation. In a time-driven system, the data is corrupted in this situation because data would be overwritten in a buffer. Typically, the functionality of the jobs is not robust to corruption of data inside the job, i.e. on the buffers over which the tasks communicate, while often the functionality is robust to corruption of data at the interfaces to the environment. This implies that the data-driven paradigm can better tolerate inadequacy of the load hypothesis than the time-triggered paradigm, and therefore better satisfies the requirements of a firm real-time embedded system.

Next to the increasing adaptivity of jobs and the fact that stream processing applications increasingly execute on architectures where the amount of provided resources is highly context dependent, a third trend is the increasing integration of more jobs in the same stream processing application. This means that the number of streams that are processed concurrently and independent of each other is increasing. This is done with the expectation that sharing resources between multiple jobs leads to reduced manufacturing and design costs. However, without proper care and measures, resource sharing creates dependencies between jobs, which can result in the situation that inadequacy of the load hypothesis of any job can cause a too late response of any job. This situation with cross-dependencies between all jobs requires load hypotheses that are adequate with a very high probability, but in practice often results in a significant test and re-design effort. We remove all these dependencies between jobs by applying run-time schedulers on every resource that provide a minimum resource budget to a job that is independent of other jobs.

Modelling and Analysis For firm real-time systems guarantees need to be provided on the satisfaction of timing constraints given that the load hypothesis is correct. This requirement makes the correctness of the model and corresponding analysis that provides the guarantees testable. If the load hypothesis is correct and the analysis claims that the timing constraints are satisfied while the job executing on the multiprocessor system does not satisfy its timing constraints, then the model or analysis is incorrect.

In this thesis, we focus on stream processing applications that are firm real-time embedded systems. Therefore, a model is required to capture the load hypothesis and to reason about the satisfaction of the timing constraints by these stream processing applications. However, stream processing jobs become increasingly adaptive which results in task graphs with inter-task synchronisation behaviour that depends on the processed data stream. Currently, cyclo-static dataflow (Bilsen et al. 1996) is the most expressive model for data-driven systems that can guarantee satisfaction of timing constraints. However, cyclo-static dataflow cannot capture inter-task synchronisation behaviour that is dependent on the processed data stream, such as found in stream processing applications like MP3 decoding and H.263 video decoding. On the other hand, boolean dataflow (Buck 1993) is a model that can capture inter-task synchronisation behaviour that is dependent on the processed data stream. However, for instance for boolean dataflow and models of comparable expressiveness there are no known approaches that can provide guarantees on the satisfaction of timing constraints. Furthermore, there is only limited support for the inclusion of the effects of run-time scheduling in cyclo-static dataflow (Bekooij et al. 2005). This thesis presents a new model that is more expressive than cyclo-static dataflow. In fact, every cyclo-static dataflow graph is a valid instance of our new model. This new model allows to provide guarantees on the satisfaction of timing constraints for task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream. Furthermore, we show that the effects of run-time schedulers that guarantee resource budgets can be included in this dataflow model.

1.2 Embedded Multiprocessor Programming

In this thesis, we consider part of the problem of programming a stream processing job on a multiprocessor system. The programming involves finding a suitable partitioning of the job into a task graph, task to processor assignment, scheduler settings and buffer capacities such that the timing constraints of the job are guaranteed to be satisfied. To cope with the complexity of this programming problem, we divide this problem into sub-problems that can be individually approached. This for instance allows to have certain sub-problems be solved manually by the programmer of the

system, while for other sub-problems algorithms and tools exist. We take the following sequence of sub-problems as our reference.

1. determination of a partitioning of the job into a task graph
2. task to processor and buffer to memory assignment
3. determination of execution times of tasks
4. determination of scheduler settings
5. determination of buffer capacities
6. analysis of timing constraints

We will first explain these steps before discussing the merits of this flow. No matter whether the job is given as sequential code or is already partitioned, in the first step, a partitioning needs to be determined that suits this multiprocessor system with its set of jobs. Algorithms and tools exist for this step in case the memory access patterns to the shared data structures are well-behaved (Kienhuis et al. 2000). Currently, the first steps are made in case of general access patterns (Bijlsma et al. 2008). Given this task graph, the second step assigns tasks to processors and buffers to specific memories. With this assignment, execution times of the tasks can be determined in the third step. The execution time is the time required by a task execution when it is executed without interruption on the processor. Note that in a multiprocessor system the memory hierarchy is shared between processors. We require that the memory hierarchy is organised such that execution times can be determined that are independent of other tasks. This is possible in our multiprocessor system, because each shared resource in the memory hierarchy has a run-time scheduler that guarantees minimum resource budgets. The execution time does not include the time spent waiting on input data to become available to this task, but does include the time required to access data after it is available, for example loads and stores to possibly non-local memories. Subsequently, we determine scheduler settings for the schedulers on the processors in step 4. After which, we determine buffer capacities in step 5. The effects of all these settings on the temporal behaviour of the job are included in a model that is analysed to verify whether satisfaction of the timing constraints can be guaranteed. State-of-the-art is to model these effects in cyclo-static dataflow and use so-called maximum cycle mean analysis to analyse whether the timing constraints are satisfied.

Even though this reference flow can be optimised by applying the analysis after each step to not further explore solutions that are already infeasible, it remains that the determination of settings is separate from the analysis of the timing constraints. Let us consider the determination of buffer capacities. Maximum cycle mean analysis can provide us with the

cycle in the cyclo-static dataflow graph that violates the timing constraints. However, it remains an open issue how best to use this information to traverse the space of possible buffer capacity settings, even more because there can still be other cycles that at the same time violate the timing constraint. An algorithm that directly computes a buffer capacity that is guaranteed to satisfy the timing constraints prevents this extensive iteration. Since we backtrack through this flow if settings lead to violation of timing constraints the algorithm to determine buffer capacities that satisfy the timing constraints is invoked numerous times when exploring the settings in steps 1 through 4. This makes a fast algorithm to compute buffer capacities that satisfy the timing constraints a necessity to further automate the programming of stream processing jobs on multiprocessor systems.

1.3 Problem Statement

The problem addressed in this thesis is to construct an algorithm with a low run-time to compute buffer capacities of run-time scheduled task graphs with inter-task synchronisation behaviour that is dependent on the processed stream, where these buffer capacities are sufficient to guarantee satisfaction of the timing constraints given the load hypothesis of this task graph.

The foremost challenge is to find a model that both can capture task graphs with inter-task synchronisation behaviour that is dependent on the processed stream as well as allows for the efficient computation of buffer capacities. This is because task graphs can deadlock dependent on the buffer capacities. Deadlock-freedom is therefore a necessary property that needs to be established in order to be able to guarantee satisfaction of timing constraints. However, if the expressiveness of a model is Turing-complete, then deadlock-freedom is an undecidable property, because of the undecidability of the halting problem for Turing machines. Therefore, there is no effective procedure to check deadlock-freedom of Turing-complete models, such as for example boolean dataflow (Buck 1993). However, this does not exclude the existence of procedures to conservatively check deadlock-freedom of such Turing-complete models, at the cost of rejecting instances that actually do not deadlock. On the other hand, the most expressive known dataflow model for which deadlock-freedom is a decidable property for any graph topology is cyclo-static dataflow (Bilsen et al. 1996). Cyclo-static dataflow cannot capture task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream. The challenge is therefore to determine a model that is more expressive than cyclo-static dataflow and that furthermore allows for an algorithm that computes buffer capacities that are sufficient to satisfy a timing constraint.

The second challenge is to determine a set of run-time schedulers that allows to provide guarantees on the satisfaction of timing constraints given

the load hypothesis of *only* this task graph and to show that the effects of these run-time schedulers can be included in the model with which buffer capacities are computed. Bounding the effect of commonly used run-time schedulers as static priority preemptive is only possible given the load hypothesis of all tasks that share this resource. In contrast to static priority preemptive scheduling, the effects of the schedulers that we apply is bounded given the load hypothesis of only the task under consideration. This allows to determine scheduler settings for the tasks of a job independent of other jobs, while still being able to provide guarantees on the satisfaction of timing constraints. Even though it has been shown that the effects of certain run-time schedulers can be included in cyclo-static dataflow (Bekooij et al. 2005), the set of run-time schedulers as well as the accuracy of the model is limited.

The final challenge is that state-of-the-art algorithms that compute buffer capacities that guarantee satisfaction of timing constraints using cyclo-static dataflow have exponential complexity (Stuijk et al. 2008). State-of-the-art is to iterate through different possible buffer capacities and for each selection of buffer capacities analyse whether the timing constraints are satisfied. The iteration through the possible options as well as the analysis both individually have an exponential complexity in the size of the cyclo-static dataflow graph.

1.4 Contributions

The main contributions of this thesis are the following.

1. A new dataflow model called Variable-rate Phased Dataflow (VPDF) that allows to capture task graphs with inter-task synchronisation behaviour that is dependent on the processed stream (Chapter 6).

2. An algorithm that uses the VPDF graph to compute buffer capacities that guarantee satisfaction of timing and resource constraints (Chapter 6).

3. The conservative inclusion in VPDF of the effects of a set of run-time schedulers that guarantee tasks a minimum resource budget (Chapter 5).

Every cyclo-static dataflow graph that is an intuitive model of a task graph is a VPDF graph, i.e. every cyclo-static dataflow graph in which all actors do not have any auto-concurrency. The algorithm that computes conservative buffer capacities has a polynomial complexity in the size of the cyclo-static dataflow graph, at the cost of an inexact result. Furthermore, we improved the accuracy with which the effects of run-time schedulers that guarantee tasks a minimum resource budget are modelled in dataflow

graphs. The validity of our analysis is confirmed by simulation in both a dataflow simulator as well as in a cycle-accurate simulator.

1.5 Approach

Providing guarantees on the satisfaction of timing constraints for an individual job is in general not possible, and requirements on hardware and software are required to enable the provision of such guarantees (Thiele and Wilhelm 2004). In (Thiele and Wilhelm 2004), two reasons are given that preclude the possibility of guarantees on temporal behaviour. The first reason is that guarantees can depend on information that is unknown. The second reason is that even if all required information is known, the system complexity can be such that no useful bound can be derived on its temporal behaviour. Our approach is to remove these problems by proposing restrictions on hardware and software such that guarantees on the satisfaction of timing constraints can be provided.

Consider, for example, a static priority preemptive run-time scheduler. The interference imposed by the high priority task on the low priority task depends on the execution time and execution rate of the high priority task. If the high and low priority task are part of different jobs, then the guarantees for the job with the low priority task depend on the execution time and execution rate of the high priority task from another job. The other job or the execution time and execution rate of the high priority task might not be known, making it impossible to provide guarantees on the satisfaction of timing constraints by the job with the low priority task. We require that all resources that are shared between jobs have a run-time scheduler that by construction provides a minimum resource budget to each task, which means that this minimum budget is independent of other tasks. An example scheduler is time-division multiplex, where the start and finish times of slices allocated to tasks are determined by a clock.

The following issue arises in the software. If tasks are allowed to use so-called non-blocking synchronisation primitives that can test for absence of data, then the functional behaviour of these tasks can depend on the arrival times of data in their input buffers. Our analysis takes place on a model that has only conservative upper bounds on arrival times of data in buffers. We require that tasks are not allowed to test for absence of data and do not use these non-blocking synchronisation primitives.

Chapter 3 discusses these and other restrictions on the hardware and software. Even though we have strived to weaken our restrictions to the best of our abilities, we make no claim to the necessity of these restrictions. However, we will show that a large class of realistic multiprocessor architectures and stream processing applications satisfy our requirements.

1.6 Outline

The outline of this thesis is as follows. We first further put our problem statement and approach in context by a detailed discussion of related work. Subsequently, we present the organisation of our system and discuss our requirements on the multiprocessor system and the implementation of the job as a task graph. This will amount to a discussion on our requirements on the input to the analysis presented in later chapters. After this, we present our dataflow analysis framework in steps, with first an introduction to dataflow analysis and then a discussion of our main contributions. The first step is to only model task graphs that have inter-task synchronisation behaviour that is independent of the processed stream and are executed on a system without run-time scheduling. This is a discussion of current state-of-the-art dataflow modelling. Subsequently, in our second step, we will model the same class of task graphs in case they are executed on multiprocessor systems with shared resources that each have a run-time scheduler that guarantees resource budgets. This is Contribution 3 of this thesis. Next, in our third and final step, Contribution 1 is discussed, which is a new dataflow model that allows to model task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream. Together with this new dataflow model, Contribution 2 is discussed, which is an approach that computes buffer capacities that are guaranteed to satisfy timing and resource constraints. Examples that illustrate how the modelling techniques and corresponding analysis can be applied in practice are discussed subsequently. After which we conclude this thesis, with summarising the approach and discussing future work.

The discussion of related work can be found in Chapter 2. Chapter 3 discusses the requirements on the input to our analysis. An introduction to the state-of-the-art in dataflow analysis is provided in Chapter 4, while Chapter 5 extends this dataflow analysis to enable modelling of the effects of run-time scheduling. Chapter 6 presents a new dataflow model and its associated buffer computation algorithm. In Chapter 7 the applicability of our analysis approach is illustrated with a number of examples, while this thesis concludes in Chapter 8.

Chapter 2

Related work

Abstract – *In this chapter, we describe our problem and contribution in more detail by relating and contrasting our problem and contribution with existing approaches. We discuss approaches that guarantee satisfaction of timing constraints in the time-triggered, event-triggered, and data-driven paradigms. While inspired by approaches from the time-triggered and event-triggered paradigms, our contribution extends dataflow modelling approaches from the data-driven paradigm.*

In Chapter 1, we introduced our problem and provided an initial positioning of this problem and our contribution. In this chapter, we will discuss related work in detail by highlighting aspects on which they differ and highlighting aspects that provided inspiration. We restrict our comparison to important approaches that guarantee satisfaction of timing constraints for all input streams for task graphs that execute on multiprocessor systems. Approaches that rely on probabilistic arguments are not seen as related, because we required that guarantees need to be provided for all input streams based on a load and fault hypothesis. The two main differentiators with related approaches are, (1) robustness to non-conservative upper bounds on execution times, and (2) expressiveness of the task model. Secondary differentiators are, (i) the class of allowed run-time schedulers, (ii) the accuracy of the analysis, and (iii) the run-time of the analysis.

The outline of this chapter is as follows. Section 2.1 discusses multiprocessor scheduling approaches that apply time-triggered task scheduling. Even though these approaches can deal with expressive task models and

include a broad class of run-time schedulers, they depend on conservative upper bounds on execution times, which violates our design requirements. Subsequently, Section 2.2 discusses multiprocessor scheduling approaches that apply event-triggered scheduling. These approaches have less expressive task models than the time-triggered approaches, with, for instance, restrictions on the topology of the task graph. Furthermore, these approaches rely on a characterisation of event arrivals on the inputs of every task, which makes these approaches not robust to non-conservative upper bounds on execution times, while also requiring non-trivial conservative lower bounds on execution times. Then in Section 2.3, we discuss current approaches that are applicable within our data-driven paradigm, with time-triggered interfaces and event-triggered tasks that implement the functionality of the job. Even though these approaches are robust to non-conservative upper bounds on execution times, these current approaches have task models with limited expressiveness and only support a limited class of run-time schedulers. We conclude, in Section 2.4, by summarising the relation between our approach and the discussed existing approaches and discussing aspects of these approaches that inspired our work.

2.1 Time-Triggered Scheduling

We define the class of time-triggered approaches as those approaches in which task executions are initiated by a strictly periodic clock signal. The time-triggered paradigm requires conservative upper bounds on the task execution times to prevent that data is lost during the inter-task communication. Even though this requirement is impractical to satisfy in our context of stream processing applications, as argued in Chapter 1, there are a number of interesting aspects to these approaches.

First of all, the time-triggered approach is well-established and extensive experience has, for instance, been obtained with time-triggered architectures (Kopetz 1997). Secondly, a large body of literature has been established for schedulability analysis of tasks that are periodically initiated and that execute on run-time scheduled resources (Butazzo 1997; Sha et al. 2004). Even though these are important contributions, we will focus in this section on the programming of time-triggered systems. The synchronous languages (Benveniste and Berry 1991; Benveniste et al. 2003) are the dominant approach to program time-triggered systems. We will focus on the expressiveness aspects of this programming approach.

Synchronous Languages The following features are essential for characterising the synchronous paradigm (Benveniste et al. 2000), of which the languages Esterel (Boussinot and De Simone 1991), Lustre (Halbwachs et al. 1991), and Signal (Le Guernic et al. 1991) are the most important examples. A synchronous program is a non-terminating sequence of reac-

tions. The synchronous hypothesis is that each reaction is atomic and can be seen as instantaneous. This allows to see the execution of a synchronous program as a sequence of discrete events. Within a reaction decisions can be taken based on the absence of events. With the synchronous hypothesis the parallel composition of two synchronous programs is deterministic, when this composition is defined.

In the synchronous approach it is said that an activation clock can be associated with a synchronous program, since reactions can be seen as discrete events. However, one needs to be careful when interpreting the notion of a clock. A synchronous program is a sequence of discrete events and the distance between these events does not need to be constant, only the ordering of events is of importance. The ordering of events is defined with respect to the ticks of the activation clock. The times at which these ticks occur is determined by the environment of the program.

The synchronous language Lustre (Halbwachs et al. 1991) allows different parts of a synchronous program to be activated by different clocks, and has operations on clocks that allow one part of the program to control the activation clock and thereby the activation of another part. Programs with different activation clocks require a consistency check. For Lustre a clock calculus was constructed that by syntactical substitutions determines whether two clocks are the same. This provides for an efficient procedure to determine whether a program is consistent, while an exact check that involves the semantics of the program is in general undecidable (Halbwachs et al. 1991). The notion of consistency for synchronous programs, e.g. Lustre and Signal programs, has close relations with the notion of consistency for dataflow graphs (Lee 1991). Similarly to the consistency check for Lustre, the efficiency of the consistency check for variable-rate phased dataflow as presented in Chapter 6 also relies on the fact that it is not exact. This consistency check can reject valid programs, because the dataflow model has only limited support for modelling relations between the behaviours of different parts of the program. Assuming and modelling independency of behaviours of different parts of the program enables an efficiently computable consistency check, but can lead to the false conclusion that the program should be rejected.

An important difference between dataflow and the synchronous approach is that in the synchronous approach events are globally ordered while events are partially ordered in dataflow. This difference allows synchronous programs to test for absence of values, while remaining functionally deterministic. However requiring a global ordering of events is problematic when implementing synchronous programs on systems with no global notion of time. Approaches to implement synchronous programs on systems with no global notion of time transform the synchronous program to explicitly communicate the presence or absence of values instead of letting the program test for absence of values (Benveniste et al. 2000). This would mean that on multiprocessor systems where processors each

have their own clock this difference in expressiveness disappears.

An essential difference between synchronous languages and dataflow is that dataflow has queues that buffer tokens. A reaction of a synchronous program is required to finish before the next tick of the activation clock. There is no such requirement for a dataflow actor. Instead, buffering allows subsequent firings to compensate for each others firing durations. For example, consider a dataflow actor that consumes one token in every firing. Further, consider that this dataflow actor has firing durations of four and two time units in an alternating fashion. Even though this actor has a maximum firing duration of four time units, it can keep up with a production rate of one token every three time units. This is because of buffering. It is not clear how this behaviour can be expressed in synchronous languages.

In (Lublinerman et al. 2009) it is stated that unified treatment of separate compilation of synchronous programs is a largely open research problem. Code-generation implies sequentialisation and introduces ordering constraints. The ordering constraints as they are introduced by sequentialisation can result in deadlock in case of separate compilation of synchronous programs (Benveniste et al. 2000). It can be interesting to investigate whether dataflow modelling as presented in this thesis can contribute to approaches for this problem. This is because dataflow actors can model the interfaces of the compiled synchronous programs, after which dataflow analysis can determine whether the composition of synchronous programs deadlocks, i.e. whether the dataflow graph deadlocks. While (Lublinerman et al. 2009) discusses separate compilation of synchronous programs in a synchronous context, (Benveniste et al. 2000) discusses separate compilation of synchronous programs in a distributed system with no global notion of time. In the latter case, dataflow can serve as the formalism that allows to analyse the composition of the different synchronous programs, i.e. to form the model for their coordination (Papadopoulos and Arbab 1998).

2.2 Event-Triggered Scheduling

A family of related approaches to derive buffer capacities and end-to-end latencies for event-triggered systems originated from the network calculus of (Cruz 1991a,b). As custom in literature, we will also name this family of approaches network calculus even though significant extensions have been made since (Cruz 1991a,b) resulting in the work described in (Le Boudec and Thiran 2001). In this section, we will first sketch the evolution of network calculus, after which we will discuss real-time calculus (Thiele et al. 2000; Haid and Thiele 2007) and Symta/S (Jersak et al. 2005), which both apply concepts from network calculus to the domain of stream processing applications that execute on embedded multiprocessor systems. In real-time calculus and Symta/S the concepts that were developed to reason about buffer capacities and end-to-end latencies of connections are applied

to reason about buffer capacities and end-to-end latencies of task graphs.

Network calculus (Cruz 1991a,b) was introduced to provide guarantees on required buffer capacities and end-to-end latency of data flowing through a network connection, this in contrast to the statistical assertions provided by traditional queuing theory. Input to the analysis are characterisations of the input traffic of each connection in the network and characterisations of the network elements, most notably schedulers. The characterisation of the input traffic is by specifying an upper bound on the traffic that is injected into the connection in any interval of time, where this upper bound is specified by a parameter that specifies the average rate and a parameter that specifies the burstiness. The characterisation of the schedulers is by an upper bound on the delay that can depend on the traffic characterisations of all connections served by this scheduler. This is problematic in case there is no (conservative) traffic characterisation for some connections. Furthermore, as we will further discuss in Section 3.3 this leads to cyclic resource dependencies and results in a complex analysis problem for which an approach is provided in (Cruz 1991b). It is, however, unclear what the accuracy is of this approach. The reasoning in (Cruz 1991a,b) is based on backlogged periods, which are the time intervals in which there continuously is data in the input buffer waiting to be scheduled.

In (Stiliadis and Varma 1998), the class of allowed schedulers is restricted to those schedulers for which the interference experienced by one connection is independent of the arrival rate of data on the other connections that share this resource. Consequently, if worst-case execution times for these other streams are known, then guarantees on buffer capacities and end-to-end latency can be provided for a connection in isolation. The restriction to the just specified class of schedulers breaks the earlier mentioned cyclic resource dependencies and results in a more straightforward analysis. The schedulers are charaterised by a latency and a rate parameter. This model was the inspiration for the work in Chapter 5, in which we also characterise the effects of scheduling by these same two parameters. Our model is applicable for the set of schedulers that is defined in (Stiliadis and Varma 1998). The reasoning in (Stiliadis and Varma 1998) is based on busy periods, which is a less intuitive concept than backlogged periods, but leads to more accurate results. While a backlogged period depends both on arrival times in the buffer and the times at which data is removed from this buffer, a busy period depends on the arrival times and the allocated rate with which data is removed from this buffer. A busy period is independent of the actual times at which data is removed from the buffer. The latency and rate characterisation of a scheduler is defined on busy periods and is independent of arrival rates of data and is an abstraction of the scheduler itself, given that conservative worst-case execution times are known.

The original work of Cruz (Cruz 1991a,b) has been extended to a framework of arrival and service curves (Le Boudec 1998; Le Boudec and Thiran 2001). Arrival curves provide an upper bound on the input traffic that

is valid over any interval, and service curves provide lower bounds on the provided service that are valid over any interval. The service curve framework has an analysis with a higher accuracy than the original work of Cruz. This is because curves instead of a single delay are applied and because the derivation of end-to-end bounds on delay by convoluting arrival and service curves are tighter than summing up delays per network element. The class of schedulers addressed in the service curve framework is larger than the class of latency-rate servers as defined in (Stiliadis and Varma 1998). This comes at the cost of reduced modelling accuracy. Service curves provide guarantees over any interval, while a latency-rate characterisation is only valid in a busy-period. A latency-rate characterisation is for example more accurate in case of a starvation-free scheduler with priorities (Åkesson et al. 2008). This example scheduler has budgets and priorities for each task. If the highest priority task is activated sufficiently often then at some point in time it will run out of budget and will need to wait until its budget is replenished. Providing a guarantee over any interval will lead to the conclusion that the highest priority task will first need to wait for its budget before it can start. A characterisation with busy periods will lead to the conclusion that the first activation of the highest priority task can immediately start, because it starts a new busy period and all budget is always available at the start of a busy period.

Real-time calculus (Thiele et al. 2000) is based on the arrival and service framework from (Le Boudec 1998) and shares the properties of the arrival and service curve framework as described in the previous paragraph. While network calculus aims to provide guarantees on required buffer capacities and maximal end-to-end latency for network connections, real-time calculus aims to provide the same guarantees on buffer capacities and end-to-end latency for stream processing applications that execute on embedded multiprocessor systems. To this end, extension have been made that among others allow to use application knowledge to obtain more accurate analysis results (Maxiaguine et al. 2004) and that allow tasks to have complex activation conditions (Haid and Thiele 2007).

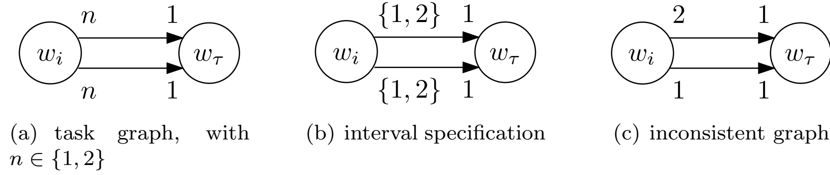
Our approach as described in this thesis is fundamentally different from network calculus, even though our token transfer curves as shown in Chapter 6 were inspired by network calculus. Our approach is not based on backlogged or busy periods and also does not bound arrivals in time intervals, instead we determine upper bounds on individual arrival times. In network calculus arrivals are bounded for all possible actual schedules. These actual schedules are all such that they satisfy the throughput constraint, while satisfaction of the end-to-end latency constraint is verified by the analysis through network calculus. In our approach, we make an abstraction of the actual system in the form of a dataflow model. Dataflow models have the property that they have monotonic temporal behaviour. This property is essential in our approach as it enables us to construct a schedule that is such that all actual schedules have earlier arrival times

than this constructed schedule. Temporal monotonicity, therefore, only requires us to show that timing constraints are satisfied for a single constructed schedule instead of for all actual schedules. Furthermore, as long as the constructed schedule is conservative to the actual schedules the construction can take into account both constraints and optimisation criteria. Our algorithm to construct schedules as described in Chapter 6 constructs a schedule that satisfies timing and resource constraints and optimises the schedule to minimise resource usage. The construction of a schedule can directly manipulate start times of tasks to a certain objective. In the actual system, start times are only indirectly manipulated through for example scheduler settings. We expect that direct manipulation of start times results in more accurate analysis results. Furthermore, with network calculus, finding settings that satisfy constraints and optimise resource usage is an iterative process as changes to the settings result in different actual schedules and a different input to the analysis. Our algorithm directly computes buffer capacities and does not iterate through various buffer capacity options. The class of run-time schedulers for which our dataflow analysis is applicable is the class of latency-rate schedulers as defined in (Stiliadis and Varma 1998), which is a subset of the class of schedulers that (Cruz 1991a,b; Le Boudec 1998; Thiele et al. 2000) consider.

Independently of the differences between our approach and network calculus (based) approaches, a number of open issues can be identified that are shared by the network calculus approaches.

Network calculus has as input (1) an upper bound on packet sizes, i.e. synchronisation granularity, per buffer, and (2) upper and lower bounds on the number of data items that arrive in any given time interval. This implies that no coupling is specified between the synchronisation granularities on different buffers adjacent to a task. The task graph, as shown in Figure 2.1(a), has a task that produces n data items per executions, with n a value that is either one or two which is allowed to change from execution to execution. It is clear that, for deadlock-free execution, a buffer capacity of two data items is sufficient on both buffers. However, if only intervals are specified per buffer, as shown in Figure 2.1(b), then the specification allows for an unbounded number of executions of the situation shown in Figure 2.1(c). Such a sequence of executions results in an unbounded accumulation of data on the top buffer, and requires an unbounded buffer capacity for this buffer. Even though the example task graph could be implemented with a single buffer, task graphs with three tasks exists for which this problem cannot be evaded in this way. The specification of intervals of synchronisation granularities limits the topology of graphs for which network calculus is applicable to trees.

The next two points that we will raise seem artificial in the domain of network connections, but are essential in the domain of stream processing applications. Because real-time calculus and Symta/S aim to apply concepts from network calculus in the domain of stream processing appli-



(a) task graph, with $n \in \{1, 2\}$ (b) interval specification (c) inconsistent graph

Figure 2.1: With multiple paths between two tasks, the existence of bounded buffer capacities requires a coupling between variation in transfer quanta on these paths.

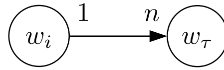


Figure 2.2: Task graph, $n \in \{1, 2\}$.

cations, we will discuss the following aspects of network calculus in the context of stream processing applications implemented as task graphs.

In network calculus, the analysis starts from the input traffic characterisation and progresses along the task graph to derive traffic characterisations on the various buffers. This implies that analysis of a task graph that has a sink with a throughput requirement is outside the scope of network calculus. Examples of such task graphs are audio and video decoders that read their input data from a disk and have a strictly periodically executing sink. Suppose in the task graph of Figure 2.2 that task w_τ is required to execute strictly periodically. In this case, the traffic of w_i can no longer be characterised independently. This can be seen by the fact that if w_i produces at the maximum consumption rate of w_τ, then a buffer of unbounded capacity is required for lower rates of w_τ in order not to lose data. If task w_i produces data at a lower rate than the maximum consumption rate, then data will not always arrive in time in the buffer to satisfy the throughput constraint. Our approach as presented in Chapter 6 can take into account that start times of task w_i will be delayed dependent on the consumption rate of w_τ, because task w_i will only start as soon as there is an empty location available in the buffer.

The progression of the analysis from the source of the task graph also makes the inclusion of cyclic dependencies in the task graph problematic, since the analysis will need to iterate through this cycle until a fixpoint has been reached.

As we will discuss further in Chapter 3, we require our tasks to first wait on sufficient empty locations in a buffer before data is written into that buffer. This is a robust mechanism to prevent that data is overwritten,

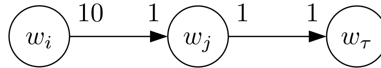


Figure 2.3: Task graph with burst.

while it at the same time keeps jitter under control. This flow control mechanism results in so-called back-pressure. Consider for instance the task graph of Figure 2.3. With no flow control, the burst of ten data items produced by task w_i also influences the required capacity of the buffer between tasks w_j and w_τ. With flow control, the burst of ten data items is absorbed in the buffer between task w_i and task w_j, and task w_i is prevented from producing a new burst until tasks w_j and w_τ have processed sufficient data to sufficiently empty the buffer between tasks w_i and w_j.

Already in (Cruz 1991a) it has been recognised that flow control leads to smaller buffers. However, the approach in (Cruz 1991a) is to insert so-called rate-regulators, which have the problematic aspect that their correct functioning, i.e. no data is overwritten, depends on conservative upper-bounds on execution times. In contrast, waiting on sufficient empty locations is independent of the quantitative temporal behaviour of the tasks.

In the literature on network calculus, results do exist for task graphs with a chain topology where data producing tasks wait on sufficient empty locations in their output buffer (Agrawal 1999). However, it is not clear what the accuracy is of this analysis nor what the computational complexity is of computing this fixpoint. Furthermore, extensions to general graph topologies, allowing for instance cycles, and the inclusion of constraints on maximum buffer capacities are required in order for this approach to be applicable in our stream processing application domain. In our domain, we typically have constraints on maximum buffer capacities and they have two types of origins. The first type is a functional requirement. For this type, a change in the buffer capacity implies a change in the functionality. Examples are adaptive filters with buffers that hold previous samples and video decoders with buffers that hold reference frames. The second type is a resource availability requirement. In case the buffer is part of the multiprocessor system that we are programming and implemented in hardware, then the buffer has a fixed size. Otherwise, in case, the buffer is implemented in software, then the buffer has a constrained maximum size. This holds especially for buffers implemented in on-chip memories.

Real-time calculus (Thiele et al. 2000; Haid and Thiele 2007) and Symta/S (Jersak et al. 2005) aim to apply concepts from network calculus to derive bounds on required buffer capacities and on end-to-end latency for stream processing applications that execute on embedded multiprocessor systems. Both approaches have, for example, made extensions that allow

tasks to have complex activation conditions to satisfy requirements of this domain, however from network calculus they inherited all the just discussed limitations.

2.3 Data-Driven Scheduling

In the data-driven paradigm, we have tasks that start based on the availability of data and we have interfaces that start periodically. Both aspects can be captured in a dataflow model. With dataflow modelling, task graphs are modelled by dataflow graphs. A dataflow graph consists of actors interconnected by queues. For every task that waits on sufficient data to start, there is a corresponding actor that waits on a corresponding number of tokens. We also model the interfaces as dataflow actors. Dataflow modelling shows that the throughput constraint is satisfied by showing that a schedule exists for the dataflow graph in which the actors that model the interfaces can execute strictly periodically.

In this section, we discuss dataflow modelling by discussing dataflow models that require increasingly dynamic scheduling, going from static-order scheduling to quasi static-order scheduling to run-time scheduling. This follows a trend to introduce more expressive models.

Static-Order Scheduling Single-rate (Reiter 1968), multi-rate (Lee and Messerschmitt 1987), and cyclo-static (Bilsen et al. 1996) dataflow can model task graphs with inter-task synchronisation behaviour that is independent of the processed data stream. For these dataflow models a fully static or static-order schedule can be constructed. Single-rate dataflow is also known as homogeneous synchronous dataflow (Lee and Messerschmitt 1987) and as marked graphs (Commoner et al. 1971). Multi-rate dataflow is also known as synchronous dataflow (Lee and Messerschmitt 1987). A fully static schedule determines the task invocation order and the start times, while a static-order schedule only determines the task invocation order. Deadlock-freedom is a decidable property of these models, because for every instance of these models it can be verified whether a non-terminating schedule exists (Reiter 1968; Lee and Messerschmitt 1987; Bilsen et al. 1996).

A static-order schedule can be executed in a so-called self-timed fashion, where the actors fire as soon as sufficient tokens are present to satisfy the firing rule. Bounds on throughput (Sriram and Bhattacharyya 2000) and latency (Moreira and Bekooij 2007) can be derived with maximum cycle mean analysis for single-rate dataflow graphs that execute in a self-timed fashion (Sriram and Bhattacharyya 2000). For multi-rate and cyclo-static dataflow algorithms exist (Sriram and Bhattacharyya 2000; Bilsen et al. 1996) to construct an equivalent single-rate dataflow model on which maximum cycle mean analysis can be performed to derive throughput and

latency of the self-timed execution of the multi-rate or cyclo-static dataflow graph. The self-timed execution of a single-rate dataflow graph first has a transient schedule before it settles into a periodic regime (Sriram and Bhattacharyya 2000). A problem brought forward in (Bambha et al. 2002) is that maximum cycle mean analysis only provides information about the length of this periodic regime (Sriram and Bhattacharyya 2000), while the transient can have a length that is exponential in the graph size (Sriram and Bhattacharyya 2000). In this thesis, and independently in (Moreira and Bekooij 2007), it is shown that there always exists a periodic schedule with a period equal to the maximum cycle mean that provides an upper bound on the start times during the transient phase. Furthermore, if an actor fires strictly periodically in this constructed periodic schedule, then the self-timed schedule of the dataflow graph still allows for strict periodic firings of this actor, because tokens can only arrive earlier in the self-timed schedule than in the constructed periodic schedule. This means that maximum cycle mean analysis can show that the self-timed schedule allows an actor to fire strictly periodically.

In the approach presented in (Bambha et al. 2002) the effects of run-time scheduling on shared resources of the multiprocessor system are not included in the dataflow analysis. In this thesis, we will show that if the run-time scheduler provides resource budgets to the tasks sharing this resource, then the effects of this run-time scheduler can be included in the dataflow analysis.

There are, however, the following problems with dataflow analysis by maximum cycle mean analysis: (1) expressiveness of the models, (2) it does not compute settings, and (3) run-time of the analysis. First of all, maximum cycle mean analysis is defined for single-rate dataflow. The most expressive model for which maximum cycle mean analysis is currently applicable is cyclo-static dataflow, which cannot model task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream. Secondly, maximum cycle mean analysis can be used to determine the throughput and latency given the scheduler settings and buffer capacities. In order to derive these settings one has to iterate through the possible options. The third problematic aspect is that the size of the single-rate dataflow graph that corresponds with a multi-rate or cyclo-static dataflow graph can be exponentially larger than the size of this multi-rate or cyclo-static dataflow graph. This implies that analysis of throughput and latency of multi-rate and cyclo-static dataflow graphs has exponential complexity. The approach from (Ghamarian et al. 2006) does not explicitly create the single-rate dataflow graph that corresponds with the multi-rate dataflow graph, but instead executes a self-timed schedule of the multi-rate dataflow graph. This schedule is executed until the periodic regime is detected. This approach has the same exponential complexity, since the transient behaviour that occurs before the periodic regime is reached can be long and the periodic regime itself can be exponential in the number of

tokens on the cycles in the graph that determine the throughput (F. Bacelli et al. 1992; Sriram and Bhattacharyya 2000). In this thesis, we present variable-rate phased dataflow, which is a new dataflow model that is more expressive than cyclo-static dataflow. Given that every actor has a self-edge with one initial token, every cyclo-static dataflow graph is a variable-rate phased dataflow graph. We present an algorithm that can be applied on a variable-rate phased dataflow graph to directly compute buffer capacities that satisfy timing and resource constraints, i.e. constraints on throughput and latency and constraints on maximum buffer capacities. This is a synthesis approach that does not iterate through the possible buffer capacities. In case this algorithm is applied on cyclo-static dataflow graphs, then this algorithm has a low-degree polynomial computational complexity in the graph size (Wiggers et al. 2007c). This low computational complexity comes at the cost that the algorithm is conservative, but not exact. This sub-optimality implies that the algorithm is not always able to determine buffer capacities that satisfy the constraints even though these buffer capacities do exist.

Quasi Static-Order Scheduling There are a number of approaches that allow to model inter-task synchronisation behaviour that depends on the processed data stream by switching between different dataflow graph instances. A static-order schedule is constructed per dataflow graph instance, which together with the switches creates a so-called quasi static-order schedule. Approaches that construct quasi static-order schedules include (Bhattacharya and Bhattacharyya 2001; Buck 1993; Girault et al. 1999; Neuendorffer and Lee 2004). These approaches require the existence of a static-order schedule for the (sub)graph, and switch only when an iteration of this schedule has finished. This requires that changes in the inter-task synchronisation behaviour can only occur every (sub)graph iteration. This is a (global) requirement on the graph. However, for instance, a variable length decoder changes its consumption quantum dependent on the processed data stream and independent of other tasks production and consumption quanta, i.e. it makes a local decision on its consumption quanta, which is independent of graph iterations. The model presented in this thesis does not have this global requirement and can model these local decisions that depend on the processed data stream.

The approach presented in (Sen et al. 2005) proposes to have tasks produce and consume a constant number of data structures, where these data structures have a variable size. In this way a task graph is created that can be modelled by a dataflow model for which a static-order schedule exists. However, this approach is not applicable for the task graph of Figure 2.2. This is because this approach requires that a data structure of size n is produced by task w_i, while w_i might not know n.

Run-Time Scheduling In the last years, work has been published on using multi-rate dataflow to model task graphs that execute on run-time scheduled resources and that have inter-task synchronisation behaviour that is independent of the processed data stream (Bekooij et al. 2005; Moreira et al. 2005; Stuijk et al. 2007). The reason to apply run-time scheduling in these works is that a multi-processor system is assumed on which multiple task graphs can execute concurrently, where these task graphs can be started and stopped by the user. In these approaches only time-division multiplex and round-robin scheduling are applied. In this thesis, we model task graphs with inter-task synchronisation behaviour that is allowed to depend on the processed data stream. Furthermore, we allow these tasks to be scheduled by run-time schedulers that guarantee every task a resource budget.

Models that are too expressive to allow for the construction of a schedule at design-time require run-time scheduling. Cyclo-dynamic dataflow (Wauters et al. 1996) and bounded dynamic dataflow (Pankert et al. 1994) apply run-time arbitration to allow for data-dependent execution rates, but do not provide an approach to calculate buffer capacities that guarantee satisfaction of a throughput constraint. The model presented in this thesis also applies run-time scheduling to allow for data-dependent execution rates. However, we do present an algorithm that computes buffer capacities that satisfy a throughput constraint. We show that the computed buffer capacities are such that for any processed data stream there always exists a schedule of actor firings that satisfies the throughput constraint. Showing the existence of a schedule is sufficient, because run-time scheduling is applied.

Another aspect that is different from related work is the following. We allow parameters to attain the value zero, which models conditional execution of tasks. Existing work that allows conditional execution of tasks, however, has its drawbacks. For boolean dataflow (Buck 1993) graphs, we know that a consistent graph can still require unbounded memory. For well-behaved dataflow (Gao et al. 1992), we know that any graph constructed using the presented rules only requires bounded memory. However, no procedure is given that decides whether any given (boolean) dataflow graph is a well-behaved dataflow graph. Also for Kahn process networks (Kahn 1974; Kahn and MacQueen 1977) there is no efficient procedure to decide for any given Kahn process network whether bounded buffers are sufficient for all input streams (Parks 1995).

In contrast to existing work, we present a simple decision procedure to check whether any given task graph is a valid input for the algorithm that computes buffer capacities that satisfy a throughput constraint.

2.4 Conclusion

In this chapter, we have discussed approaches that guarantee satisfaction of timing constraints within the time-triggered, event-triggered, and data-driven paradigms.

We have seen that the time-triggered and event-triggered paradigms are not robust to non-conservative upper bounds on execution times. Models for the time-triggered paradigm require that tasks are triggered by a clock signal, which imposes restrictions on the expressiveness of these models. Existing models in the event-triggered paradigm have restrictions on the topology of the task graph. Cyclic dependencies are, for example, difficult to analyse with current models within this paradigm. Further, the accuracy and run-time of the corresponding analysis is not known for all cases. Even though the time-triggered and event-triggered paradigms do not satisfy our requirements, aspects that resemble or inspired our work have been highlighted.

Dataflow modelling is the approach to guarantee satisfaction of timing constraints within our data-driven paradigm. Guarantees on satisfaction of timing constraints can be provided for given upper bounds on execution times. In case these upper bounds are not conservative, then, in our data-driven paradigm, this does not immediately imply that timing constraints are violated.

We extend state-of-the-art dataflow modelling by a new dataflow model that is amenable to an efficient analysis that guarantees satisfaction of timing constraints for all input streams. Furthermore, this thesis shows that the effects of a class of run-time schedulers can be conservatively included in dataflow analysis. The novelty of our dataflow model is that it enables to provide guarantees on end-to-end timing constraints for task graphs with tasks that have aperiodic execution rates.

Chapter 3

Synchronisation and Arbitration Requirements

ABSTRACT – *In this chapter, we discuss requirements on the inputs to our analysis that computes buffer capacities. This thesis aims to compute buffer capacities that guarantee satisfaction of timing and resource constraints of jobs implemented as task graphs that execute on a multiprocessor system with run-time schedulers. In order to accomplish this goal, we need to impose certain requirements on the implementation of a job as a task graph, and on the type of run-time schedulers that are applied in this multiprocessor system. Our requirements are presented and discussed in this chapter.*

In order to be able to compute buffer capacities that satisfy timing and resource constraints, requirements on the input to the analysis that computes these buffer capacities are necessary. In this chapter, we discuss our requirements on the inputs to our analysis that computes buffer capacities. As illustrated by Figure 3.1, the inputs to our analysis are (1) a task graph, (2) execution times, (3) scheduler settings, (4) timing constraints, and (5) resource constraints.

The outline of this chapter is as follows. First in Section 3.1, we discuss our requirements on the implementation of jobs as task graphs on a multiprocessor system. Then, in Section 3.2, we present our task model in more detail and present our definition of execution times. In Section 3.3, we discuss various options on multiprocessor scheduling. We will conclude

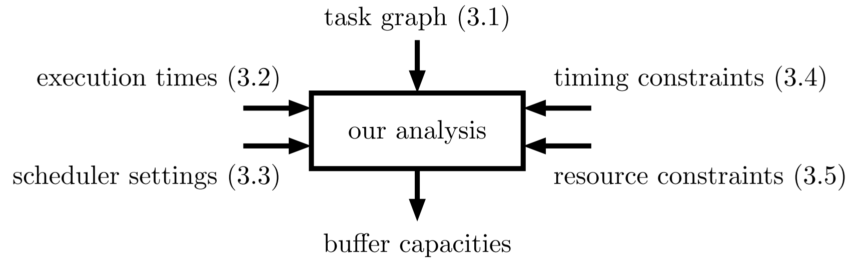


Figure 3.1: Input and output of the analysis presented in this thesis.

that requiring a run-time scheduler from the class of budget schedulers on every resource in the multiprocessor architecture allows to provide guarantees on the temporal behaviour per job. This is a necessary condition for the specified buffer capacity computation problem. Subsequently, in Sections 3.4 and 3.5, we specify the constraints that need to be satisfied by our analysis that computes minimal buffer capacities.

3.1 Synchronisation Requirements

In this section, we discuss our requirements on the implementation of a job on a multiprocessor system. We will first discuss our requirements on the programming model. This will amount to the requirement that the job is implemented by a task graph, where the tasks explicitly synchronise over FIFO buffers, as illustrated in Figure 3.2. We will require that this synchronisation is such that the resulting task graph is functionally deterministic. Subsequently, we will require that jobs interface with their environment by time-triggered interfaces, which are illustrated by squares in Figure 3.2. Time-triggered input interfaces provide temporal isolation from the environment, and time-triggered output interfaces remove jitter in the arrival times of the output values. The environment of the job includes both the system environment and the non-real time control processing. The non-real time control processing is responsible for translating events in the system environment to control events for the job, which includes both starting and stopping jobs on request of the end-user as well as the proper selection of job parameter values. Jobs themselves can be highly adaptive to their environment and can include complex sequential control. In other words, the job includes both real-time stream processing as well as the real-time control that steers this stream processing.

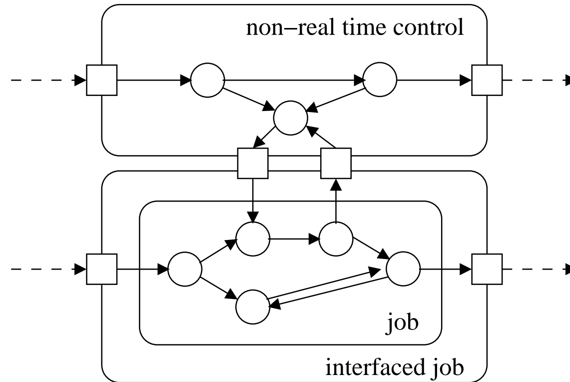


Figure 3.2: Organisation of functionality in a job that processes a stream of data, which is controlled by parameters determined by non-real time control software. The square nodes denote time-triggered interfaces, while the round nodes denote data-driven tasks. Solid arrows denote FIFO buffers.

3.1.1 Programming Model

In this section, we define our programming model. Following (Culler et al. 1999), a programming model is the conceptualisation of the multiprocessor system that the programmer uses in coding jobs. A programming model specifies how parts of the job that execute in parallel communicate information to one another and what synchronisation operations are available to coordinate their activities. Jobs are written assuming a particular programming model.

We assume a shared address space multiprocessor system [1]. In a shared address space multiprocessor system, communication is through writing and reading values in memory locations. In order to provide semantics to this communication, we need to define a model that specifies constraints on the order in which these memory operations become visible with respect to one another. This is called the programming model of a shared address space multiprocessor system. The programming model for a shared address space multiprocessor system includes (1) a so-called system specification, (2) a programmer's interface, and (3) a translation mechanism. The system specification defines which orderings of memory accesses are guaranteed to be preserved, and the mechanisms that the programmer has to enforce ordering of memory accesses. To prevent that the programmer needs to be aware of orderings of memory accesses, a set of rules and annotations is specified. These rules and annotations are such that if the coding of the job follows the rules and provides sufficient annotations, then this leads

[1] Can be implemented with distributed memories.

to a "safe" execution of the job, which means that the semantics of the execution of the job on this multiprocessor system is fully specified. This set of rules and annotations is called the programmer's interface. Finally, a translation mechanism translates the annotations into enforcement of memory access ordering.

In this section, we define the concepts that form the programmer's interface. This programmer's interface is such that efficient translation mechanisms can be build, in the form of run-time libraries (de Kock 2000; Nieuwland et al. 2002; van der Wolf et al. 2004), that allow "safe" execution of these jobs on a multiprocessor system that provides streaming consistency (van den Brand and Bekooij 2007), which is introduced subsequent to the introduction of our programmer's interface.

Our programmer's interface implements a job by a task graph, where tasks can execute in parallel and communicate over FIFO buffers on which they synchronise on containers.

Definition 3.1 (Task) *A task is a (finite) sequence of program statements that is executed infinitely often.*

Definition 3.2 (Container) *A container is a set of memory locations.*

Definition 3.3 (Acquired container) *A container is acquired, if a task has exclusive access to the container.*

Definition 3.4 (Released container) *A container is released, if a task previously acquired this container and now signals that it will no longer access this container before first again acquiring this container.*

Definition 3.5 (FIFO buffer) *A FIFO buffer b is adjacent to two tasks, of which one is the source of the FIFO buffer and the other task is the destination of the FIFO buffer. A FIFO buffer contains a finite set of containers, which is partitioned into four sets, (1) containers acquired by the source, (2) containers released by the source, (3) containers acquired by the destination, and (4) containers released by the destination. Containers released by the source are called full containers, and containers released by the destination are called empty containers. We require that the set of full containers is organised as a queue, i.e. the full queue, and that the set of empty containers is organised as a queue, i.e. the empty queue. The source can only release full containers at the tail of the full queue, and the destination can only acquire full containers from the head of the full queue. The destination can only release empty containers at the tail of the empty queue and the source can only acquire empty containers from the head of the empty queue.*

We define a task graph as a weakly connected directed multi-graph, where the vertices denote tasks and the edges denote buffers, with the

direction of the edge representing the direction in which data values are communicated. A weakly connected directed graph is a graph of which the underlying undirected graph is connected. A directed graph is a directed multi-graph, if there are edges that have the same source and destination, i.e. if there are at least two edges that cannot be distinguished based on their source and destination vertex. Therefore, in case of a multi-graph, we distinguish between edges based on their label.

Definition 3.6 (Task graph) *A task graph $T = (W,B)$ is a weakly connected directed multi-graph that consists of a (finite) set of vertices, W, which are tasks, and a (finite) set of labeled directed edges, B, which are FIFO buffers.*

Our programmer's interface includes the rules that at any time at most one task accesses a container and that tasks only communicate over FIFO buffers and only synchronise on containers, as specified by Requirements 3.1, 3.2, and 3.3, respectively.

Requirement 3.1 (Mutually exclusive access) *A task only accesses a container, if this container is currently acquired by this task.*

Requirement 3.2 (Inter-task communication) *All communication of data values between tasks is through writing and reading values in containers from FIFO buffers.*

Requirement 3.3 (Inter-task synchronisation) *All synchronisation between tasks is by acquiring and releasing containers on FIFO buffers.*

The presented programmer's interface allows for execution of jobs on a multiprocessor system that supports streaming consistency or a stronger memory consistency model. A stronger memory consistency model is a model that guarantees preservation of more orderings of memory accesses. This means that a stronger memory consistency model allows for less reordering in its implementation, and will, in general, have lower performance. In streaming consistency, acquires may overtake releases, in case they are associated with different FIFO buffers. Release consistency (Gharachorloo et al. 1990) is an example memory consistency model that is stronger than streaming consistency, in which acquires may never overtake releases. This guarantee on ordering preservation means that release consistency allows for a programmer's interface such as for example POSIX threads (Pthread 1995).

Stream processing applications can be intuitively programmed with the presented programmer's interface in which the communication over FIFO buffers leads to a direct coupling between synchronisation actions and the associated memory locations. This coupling allows for execution on multiprocessor systems that support streaming consistency, which is a relatively weak memory consistency model

3.1.2 Functional Determinism

A programmer's interface specifies rules and annotations such that the functional behaviour of a job that is executed on a shared address space multiprocessor system is defined. In this section, we present sufficient conditions on the implementation of a job such that the output values of a job are completely determined by the input values of the job. If the output values of a job are completely determined by its input values, and thus for instance independent of the task execution schedule, then we call this a functionally deterministic job.

Definition 3.7 (Functional determinism) *A job is functionally deterministic if the output values of the job are completely determined by the input values of the job.*

As briefly discussed in Chapter 1, we take functionally deterministic jobs as a design requirement. This allows for a clean well-defined specification of the functional behaviour and a clear criterion for correctness of the implementation on a multiprocessor system. For a job to be functionally deterministic, we need the following definitions, in order to define a sufficient condition on the implementation of the job as a task graph.

We assume that all inter-task synchronisation is explicit, which means that tasks call specific functions to synchronise.

Definition 3.8 (Acquisition primitive) *A function is an acquisition primitive if the function allows the task to acquire a container.*

Definition 3.9 (Release primitive) *A function is a release primitive if the function allows the task to release a container.*

Requirement 3.4 (Explicit synchronisation) *Tasks exclusively acquire a container by calling an acquisition primitive and exclusively release a container by calling a release primitive.*

In case a task is a source on this FIFO buffer, then a blocking acquisition primitive only returns control after the specified number of empty containers is present. In case the task is a destination on this FIFO buffer, then a blocking acquisition primitive only returns control after the specified number of full containers is present in the FIFO buffer, i.e. no test for absence is allowed.

Definition 3.10 (Blocking acquisition primitive) *A blocking acquisition primitive is an acquisition primitive that only returns control to the task if the a-priori specified number of to be acquired containers is present in the specified FIFO buffer.*

In Section 3.2, we discuss our requirements on the implementation of blocking acquisition primitives. In Chapter 4, we show that if all acquisition primitives of tasks are blocking acquisition primitives and all tasks are functional, then the task graph, and therefore the job, is functionally deterministic.

Definition 3.11 (Functional task) *A task is functional if the number of to be acquired tokens on a FIFO buffer is a function of values from previously acquired containers, and the number of containers released by this task and the values contained in them are a function of values from previously acquired containers.*

In this thesis, we only consider task graphs in which all acquisition primitives are blocking acquisition primitives and all tasks are functional.

Requirement 3.5 (Functionally deterministic task graph) *Task graphs exclusively have blocking acquisition primitives and functional tasks.*

The approaches presented in (Jersak et al. 2005; Haid and Thiele 2007) allow for so-called or-activation of tasks, which is implemented by acquisition primitives that test the FIFO buffer for sufficient containers and return the result of this test to the task. These approaches, therefore, allow task graphs to have functionally non-deterministic behaviour. Because the functional behaviour of such task graphs is no longer completely determined by the input values of these task graphs, determining bounds on the temporal behaviour of the task graphs in these approaches is significantly more complicated than in our approach.

3.1.3 Interfacing

Jobs need to interface with their environment to provide the requested functionality to the end-user. This implies that, in general, a job both needs to be able to receive events from the environment and to produce output events to the environment.

We require that all interfaces of a job with its environment are time-triggered. This implies that these interfaces strictly periodically sample, i.e. observe, the environment and strictly periodically produce outputs to the environment. Time-triggered interfaces do not wait on sufficient containers to be present on their adjacent FIFO buffers. This means that buffer overflow at the input interfaces and buffer underrun at the output interfaces can occur. The advantage of requiring that all interfaces are time-triggered interfaces is that the temporal behaviour of a job is completely isolated from its environment (Kopetz 1991).

We call the implementation of a job together with its interfaces an interfaced task graph.

Definition 3.12 (Interface) *An interface starts strictly periodically and has one adjacent FIFO buffer.*

Definition 3.13 (Interfaced task graph) *An interfaced task graph is a task graph, where the set of vertices is augmented with a finite number of interfaces and the set of edges is augmented with a finite set of FIFO buffers.*

While the task graph is functionally deterministic, the interfaced task graph can be functionally non-deterministic. This is because it does not always hold that the output of the job is completely determined by the input values received by the input interfaces nor does it hold that the output values produced at the output interfaces are completely determined by the input values of the job.

This is because (1) interfaces are arbiters that decide in bounded time, (2) non-deterministic alignment of sampling times with times at which events occur in the environment, and (3) interfaces of a job cannot have the same clock.

Interfaces are Arbiters Input interfaces make a discrete decision based on a continuous range of input values. Therefore, there are input values for which the input interface cannot make a reliable decision on the correct discrete output (Chaney and Molnar 1973).

Non-Deterministic Alignment Dependent on the alignment of the sampling times with the times at which events in the environment occur, it can happen that two events in the environment are observed by two input interfaces to occur in the same sampling period or in different sampling periods. This phenomenon is described in (Caspi 2001) by the statement that logical time is not real-time, where logical time is the time of our discrete time system and real-time is the continuous time frame of the environment.

Interfaces have Different Clocks Distribution of one clock to different interfaces inherently leads to different clock signals at the interfaces due to process variation and environmental variation (Friedman 2001).

On top of these effects that are inherent to strictly periodic execution, it can occur that the FIFO buffer adjacent to an input interface overflows or that the FIFO buffer adjacent to an output interface underruns. This means that data is overwritten, i.e. lost, or read again, i.e. duplicated. However, if no buffer overflow or underrun occurs then, in our system, the values in the buffers adjacent to the output buffers are completely determined by the values written into the input buffers.

In general, the non-determinism introduced by the interfaces will be disturbing for the end-user, i.e. the end-user will perceive a loss in quality. Therefore, we have the objective to minimise these non-deterministic effects, and, for instance, strive to have no buffer overflow and underrun. Despite this objective, jobs need to be robust to the non-determinism introduced by the interfaces. In our application domain, a buffer underrun at the output interfaces can be compensated by the intrinsic interpolation done by the human ear and eye. A buffer overflow at the input interface leads to loss of input data for the job. However, typically this cannot be distinguished from loss of data in the environment, as, for instance, due to changing channel conditions in case of a digital radio receiver. Jobs in our application domain often have various mechanisms to deal with loss of data in the environment. The non-determinism introduced by having multiple input interfaces is, typically, not problematic, because jobs often process only a single stream of data. Other inputs of a job determine parameters of the processing of the stream, but these parameter values do not need to be aligned with specific values in the processed stream. The non-determinism introduced by the fact that interfaces are arbiters can be removed by making assumptions on the frequency with which events in the environment occur. In our application domain, this frequency with which events occur is typically determined as part of a standard.

3.2 Execution Times

We have now defined how the functionality of a job is implemented and how jobs interface with their environment. Furthermore, we discussed how values received from and produced to the environment depend on the temporal behaviour of the job. In order to reason about the temporal behaviour of a job, we need to associate the elapse of time with functionality. This elapse of time is called execution time, and, as discussed in Chapter 1, execution times are input to our analysis of the temporal behaviour of a job. We associate execution times with the fragment of code that is executed by a task in between acquisition primitives. To define this more precisely, we need the concepts of a code-fragment and a non-blocking code-segment.

For reasons of conciseness, our code listings include `read` and `write` primitives (de Kock 2000), which are implemented by a blocking acquisition primitive followed by a copy of values from the FIFO buffer, which is in turn followed by the corresponding release of the acquired containers.

A code-fragment is a maximal sequence of program statements with one entry and one exit point, which can include an acquisition primitive but only if this primitive is the first program statement of this code-fragment. Listing 3.1 has six code-fragments. code-fragment I includes lines 3 and 4. code-fragment II includes lines 5, 6, and 7. code-fragment III includes lines 8, 9, and 10. code-fragment IV includes line 11. code-fragment V includes

lines 12, 13, and 14, and code-fragment VI includes lines 15, 16, and 17.

Definition 3.14 (code-fragment) *The sequence of program statements of a task is statically partitioned in a sequence of code-fragments, where each acquisition of containers and each branch and join in the control-flow of the task starts a code-fragment.*

A non-blocking code-segment is a set of sequences of code-fragments that can occur during execution, where these sequences start with a program statement that is an acquisition primitive and only include this single acquisition primitive. Listing 3.1 has three non-blocking code-segments. The first non-blocking code-segment starts at line 3 and includes the set of sequences of code-fragments $\{\langle \mathrm{I}, \mathrm{II}\rangle, \langle \mathrm{I}, \mathrm{IV}, \mathrm{V}\rangle, \langle \mathrm{I}, \mathrm{IV}\rangle\}$. This is because both the for-loop and the while-loop can be entered or not. In the first case the non-blocking code-segment is until the write in line 8, in the second case this non-blocking code-segment is until the write in line 15, while in the third case this non-blocking code-segment is until this same read in line 3. The second non-blocking code-segment starts at line 8 and is the following set of sequences of code-fragments $\{\langle \mathrm{III}, \mathrm{II}\rangle, \langle \mathrm{III}, \mathrm{IV}, \mathrm{V}\rangle, \langle \mathrm{III}, \mathrm{IV}\rangle\}$, because after the write of line 8 either again the write of line 8 is reached or the write of line 15 is reached or the read of line 3 is reached. The third non-blocking code-segment is $\{\langle \mathrm{VI}, \mathrm{V}\rangle, \langle \mathrm{VI}\rangle\}$.

Definition 3.15 (Non-blocking code-segment) *A non-blocking code-segment is a (finite) set of (finite) sequences of code-fragments. Each occurrence of an acquisition primitive in the code of a task starts a non-blocking code-segment. A non-blocking code-segment includes all possible sequences of code-fragments that start from this occurrence of an acquisition primitive. These sequences terminate with a code-fragment that precedes another occurrence of an acquisition primitive.*

We define execution times for executions of non-blocking code-segments, where these execution times among other factors depend on the resource on which the non-blocking code-segment executes and the inputs for this non-blocking code-segment. Figure 3.3 shows a time-line with an execution of the second non-blocking code-segment from Listing 3.1, which is started after the arrival of three containers. The execution time is from the arrival time of the third container until the finish of this non-blocking code-segment, given an un-interrupted execution of this non-blocking code-segment.

Definition 3.16 (Task execution) *A task execution is an execution of a non-blocking code-segment, which is an execution of one of the sequences of code-fragments of this non-blocking code-segment.*

```
 1  while(1)
 2  {
 3    int a = read(1,b1);                       I
 4    int bound = f(a);
 5    for(int i = 0; i < bound; i++)
 6    {                                         II
 7      ...
 8      write(3,b2);
 9      ...                                     III
10    }
11    int quantum = g(a);                       IV
12    while(...)
13    {                                         V
14      ...
15      write(quantum,b2);
16      ...                                     VI
17    }
18  }
```

Listing 3.1: A task w_i with six code-fragments and three non-blocking code-segments formed by sequences of code-fragments.

Definition 3.17 (Start of a task execution) *The start of a task execution is at the time the acquisition primitive detects that sufficient containers are present in the required FIFO buffer.*

Definition 3.18 (Finish of a task execution) *The finish of a task execution is at the time that the next non-blocking code-segment calls its acquisition primitive.*

Definition 3.19 (Execution time) *The execution time of a task execution is the time from the start of this task execution until the finish of this task execution, in case this task is executed in isolation on this resource and without any interruptions.*

A blocking acquisition primitive is in the end implemented by repeated checks whether a synchronisation condition is satisfied, i.e. whether a certain synchronisation variable has a particular value. Two invocations of this check are included in our definition of execution time, the last check that failed and the check that discovers that the condition is satisfied. This seemingly innocent definition excludes a common implementation of blocking acquisition primitives, which is an implementation by counters. This is because both tasks update these counters in the implementation of a

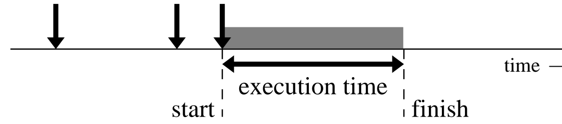


Figure 3.3: The execution time is from the arrival of sufficient containers until the finish of an un-interrupted execution of the non-blocking code-segment.

FIFO administration using counters. The problematic aspect of this implementation is that both tasks first test the counter before they update it. Because both tasks update the counter, this test is required to be part of an atomic test-and-set, which means that during this test the other task cannot access this variable. An atomic test-and-set does not prohibit one of the tasks to continuously test the variable, thereby precluding progress of the other task. Therefore, an implementation of blocking acquisition primitives by counters results in unbounded execution times. We implement the FIFO administration by pointers that each have only one task that updates them (Nieuwland et al. 2002). Accesses to these pointers return in a bounded time.

3.3 Arbitration Requirements

This section discusses the scheduling of a task graph on a multiprocessor system, and explains our choice for (semi-) static assignment scheduling with budget schedulers. We will first discuss options for multiprocessor scheduling. This will result in a choice for a run-time scheduler per resource. The second part of this section will describe three classes of run-time schedulers and define the class of schedulers that we apply in our system, which is called the class of budget schedulers.

3.3.1 Multiprocessor Scheduling

Scheduling a task graph on a multiprocessor system consists of (1) assigning executions of tasks to processors, (2) specifying the order of task executions per processor, and (3) specifying at which time the task execution starts. In (Lee and Ha 1989) a scheduling taxonomy is presented based on which of these three types of decisions are made at run-time or at design-time. Depending on which decisions are done when, the strategies as listed in Table 3.1 are obtained.

This taxonomy shows a spectrum of options and is not exhaustive. Quasi-static order scheduling (Lee 1988) is a strategy in between static-order and static-assignment. In quasi-static order scheduling some ordering

	assignment	ordering	timing
fully static	design-time	design-time	design-time
static-order	design-time	design-time	run-time
static-assignment	design-time	run-time	run-time
fully dynamic	run-time	run-time	run-time

Table 3.1: Multiprocessor scheduling strategies.

decisions are made at run-time, but only where absolutely necessary, i.e. only if these decisions depend on values in the processed stream.

Semi-static assignment (Strik et al. 2000; Moreira et al. 2005) is a scheduling strategy that is in between static assignment and fully dynamic. In semi-static assignment, the processor assignment is done at the time a job is started by the user and remains unchanged while the job is executing. This is an attractive option in case the job-mix is highly dynamic or unknown at design time, because it allows to compute the processor assignment based on currently available resources.

In this thesis, we apply (semi-) static-assignment scheduling, because it is the most cost-effective option that satisfies the requirements imposed by our application domain. Since in this thesis the processor assignment step is outside our scope, for our purposes semi-static assignment is equivalent to static-assignment scheduling. We will refer to both strategies as static-assignment scheduling. The other scheduling strategies do not satisfy our design requirements, because of the following reasons.

Fully Static Specification of the start times at design-time requires knowledge of the execution times at design-time. If this knowledge is not conservative, then the functional behaviour of a fully statically scheduled multiprocessor system is undefined. Since, in our application domain, safe upper bounds on execution times are not always available, we have a design requirement that specifies robustness to unsafe estimations of upper bounds on execution times. Fully static scheduling does not satisfy this design requirement.

Furthermore, to determine the order of task executions on a processor at design-time, we need to know for each task how many executions need to be part of one iteration through this static order. This implies that for each task its number of executions relative to the number of executions of the other tasks on this processor needs to be known. However, this relative number of task executions can depend on the processed data stream. For instance, task w_i as described by Listing 3.1 produces a number of containers on buffer `b2` that depends on the value of `quantum` which is determined by the processed stream. Therefore, the task that reads from buffer `b2`

has a number of executions relative to one execution of task w_i that is not known at design time. This implies that fully static scheduling is not applicable for all applications in our domain.

Static-Order Compared to fully static, static-order determines the start times at run-time, leading to a well-defined functional behaviour independent of execution times. However, since static order determines the ordering of task executions at design time, also with this strategy, not all applications in our domain can be scheduled.

Furthermore, if static-order scheduling is applied, then the temporal behaviour of a job depends on the execution times and execution rates of other jobs. In our system, we have a design requirement that requires bounds on the temporal behaviour per job, i.e. independent of other jobs, because execution times and execution rates of other jobs might not be known at design-time. Static-order scheduling over multiple jobs does not satisfy this design requirement.

Fully Dynamic Assigning task executions to processors at run-time means that instructions and data associated with a task potentially need to be moved from one memory local to a processor to another memory local to a different processor. This implies additional scheduling overhead, but also implies additional uncertainty in the temporal behaviour of the application. Any approach that provides upper bounds on the temporal behaviour of task executions will need to take these re-locations into account, which will result in less accurate analysis results and a resource allocation that potentially has low resource utilisation.

Furthermore, assigning task executions to processors at run-time requires a dispatcher, which introduces a dependency between the temporal behaviour of all tasks, since all tasks signal the finish of their executions to this dispatcher. We expect that providing guarantees on the temporal behaviour per job as well as scaling this approach to a large number of tasks can be problematic.

Even though we exclude static-order scheduling over multiple jobs, we see (quasi) static-order scheduling within one job, or for a subset of tasks of one job, as an attractive option, because of its cost-effectiveness (Moreira et al. 2007; Stuijk et al. 2007; Falk et al. 2008). This topic is, however, outside the scope of this thesis. In this thesis the task graph is given. This means that the job is partitioned into the task graph in a previous step of the multiprocessor programming flow of Chapter 1. We assume that any (quasi) static-order scheduling is performed on the task graph before the task graph is provided to us as input to our analysis and that (quasi) static-order scheduling is part of the first step in the multiprocessor programming flow of Chapter 1, i.e. the partitioning of the job into a task graph.

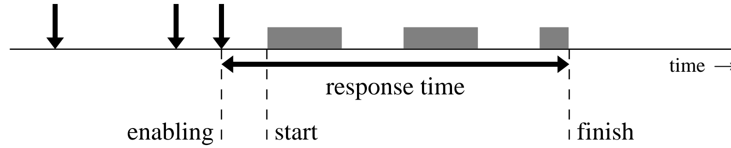


Figure 3.4: The response time is from the arrival of sufficient containers until the finish of an execution of a non-blocking code-segment.

3.3.2 Processor Scheduling

Given static-assignment scheduling, every resource has its individual run-time scheduler. The sharing of a resource inherently leads to a difference in time between the availability of sufficient containers to let a task execution start and the point in time that the task execution is actually started. We say that the task is enabled during this interval of time, which is the time interval between the enabling time and the start time of the task execution. In literature, arrival and release times are concepts similar to our enabling time.

Definition 3.20 (Enabling of a task execution) *A task execution is enabled if its acquisition primitive is currently called and sufficient containers are present in the required FIFO buffer to let the acquisition primitive return successfully.*

Definition 3.21 (Enabling time of a task execution) *The earliest time from which a task execution is enabled is the enabling time of this task execution.*

A pre-emptive run-time scheduler can cause an increase in the time between the start and finish of a task execution. The time interval from the enabling time until the finish time is called the response time of this task execution, as illustrated in Figure 3.4 for the same task execution as shown in Figure 3.3.

Definition 3.22 (Response time) *The response time of a task execution is the time interval from the enabling time of this task execution until the finish time of this task execution.*

For certain run-time schedulers, the upper bound on the response time of a task execution depends on properties of other tasks that share this resource. One of these properties is the number of executions of a task within a certain time interval. We call this the execution rate of a task.

Definition 3.23 (Execution rate) *The execution rate of a task is the number of task executions in a given time interval.*

scheduler class	dependency of upper bound on response time	
	execution time	execution rate
deterministic	X	X
latency-rate	X	-
budget	-	-

Table 3.2: Classes of run-time schedulers. Dependency on properties of other tasks on the same resource is denoted by X, and independency is denoted by -.

The upper bound on the response time of a task execution intrinsically depends on the execution time of this task execution, the scheduler settings, the scheduling overhead, and the inter-task synchronisation overhead. In the following discussion, we assume that there is no scheduling and inter-task synchronisation overhead. At the end of this section, we come back to this assumption and discuss the consequences of scheduling and inter-task synchronisation overhead. Next to the mentioned dependencies, the upper bound on the response time can additionally depend on the execution times and rates of other tasks that share the same resource. This depends on the particular run-time scheduler. Based on dependencies of the upper bound on the response time on execution times and rates of other tasks, we define the three classes of run-time schedulers, as listed in Table 3.2: (1) deterministic schedulers, (2) latency-rate schedulers, and (3) budget schedulers. The relation between these classes is one of set inclusion, i.e.

$$\text{deterministic} \quad \supset \quad \text{latency-rate} \quad \supset \quad \text{budget}$$

We will first discuss these three classes after which we will give examples of schedulers in these classes.

Deterministic Schedulers A deterministic scheduler is a scheduler that makes deterministic scheduling decisions, which means that the next scheduled task is completely determined by the state of the scheduler and the set of tasks that are currently claiming the resource. This is the broadest class of schedulers considered in this thesis and is the class of schedulers that is for example considered in Network Calculus as defined in (Cruz 1991a,b). The upper bound on the response time of a task execution scheduled by a deterministic scheduler depends on both the execution time and the execution rate of other tasks.

Latency-Rate Schedulers In (Stiliadis and Varma 1998) a class of schedulers, called latency-rate schedulers, is defined that is a subclass of

the class of deterministic schedulers. By definition, tasks scheduled by latency-rate schedulers have upper bounds on their response times that do not depend on the execution rate of other tasks. The upper bounds on response times can, however, still depend on the execution times of other tasks.

Budget Schedulers Here, we define the class of budget schedulers, which is a subclass of latency-rate schedulers. By definition, task executions scheduled by a budget scheduler have an upper bound on their response time that is independent of other tasks, i.e. independent of the execution time and execution rate of other tasks.

Definition 3.24 (Budget scheduler) *A budget scheduler guarantees every task a minimum amount of time R in every interval of time with length Q.*

We call R the budget in interval Q. Budget schedulers are the subclass of a-periodic servers (Butazzo 1997) that satisfy Definition 3.24.

An example scheduler that is a deterministic scheduler but not a latency-rate or budget scheduler is static-priority preemptive (SPP). With SPP the upper bound on the response time of executions of a task with a low priority depends on both the execution time as well as the execution rate of higher priority tasks. If there is no upper bound on the execution time or no upper bound on the execution rate of a higher priority task, then the upper bound on the response time of the lower priority task is indefinite. Therefore, SPP is not a latency-rate scheduler and consequently also not a budget scheduler.

Round-robin scheduling is an example scheduler that is a latency-rate scheduler but not a budget scheduler. The upper bound on the response time of a task execution in case of round-robin scheduling does not depend on the execution rate of other tasks, because every task in the round-robin list is executed once per iteration through this round-robin list. This implies that round-robin is a latency-rate scheduler. Round-robin is not a budget scheduler, because the upper bound on the response time of a task execution in case of round-robin scheduling does depend on the execution times of other tasks, because only after a task execution finishes the next task in the list is executed. Therefore, safe upper bounds on the execution times of all tasks that share a resource scheduled by a latency-rate scheduler are required to be known to derive safe upper bounds on the response times.

The class of budget schedulers includes, but is by far not limited to, time-division multiplex, polling server (Sprunt et al. 1989) and constant bandwidth server (Abeni and Butazzo 2004). This class excludes the total bandwidth server (Spuri and Buttazzo 1996), because it cannot provide

a budget to tasks that is guaranteed to be independent of the execution times.

Application of schedulers from the class of deterministic schedulers can lead to so-called cyclic resource dependencies. An example is shown in Figure 3.5. In this figure, the solid arrows denote communication of data between tasks over queues of infinite depth. Let all four tasks consume and produce one container per execution and let us assume that we know safe upper and lower bounds on their execution times. Furthermore, let, on Processor1, task w_c have the high priority and let task w_a have the low priority. On Processor2, let task w_b have the high priority and let task w_d have the low priority. Finally, let us assume that we know upper bounds on the number of containers that arrive in a given interval of time on the input queues of tasks w_a and w_d.

Even with this large number of assumptions, which violate a number of our design requirements from Chapter 1, it is still problematic to compute an upper bound on the arrival times of containers on the queue from task w_a to task w_b. This is because the upper bound on response times of task w_a depends on the upper bound on the execution rate of task w_c. The upper bound on the execution rate of task w_c depends on the upper bound on the execution rate of task w_d. The upper bound on the execution rate of task w_d depends on the lower bound on the execution rate of task w_b. The lower bound on the execution rate of task w_b depends on the difference between the upper and lower bound on arrival times of containers on the queue from task w_a to task w_b. Therefore, we have a cyclic dependency when reasoning about an upper bound on the arrival times of containers even though both task graphs are a-cyclic. Analysis techniques exist that address this problem (Cruz 1991b; Jonsson et al. 2008). However, these analysis techniques rely on the assumptions we made for this example, which violate a number of our design requirements. Furthermore, we are not aware of any work that evaluates the accuracy or complexity of these analysis techniques.

By definition, latency-rate schedulers break this cyclic dependency by not allowing the upper bound on the response time of task executions to depend on the execution rate of other tasks. However, latency-rate schedulers still allow the upper bound on the response time of task executions to depend on the execution times of other tasks. Therefore, in order to satisfy the design requirement to characterise the temporal behaviour of individual jobs, we, in principle, apply budget schedulers in our multiprocessor system. However, there are resources in our multiprocessor system that only execute tasks for which safe upper bounds on their execution times can be, relatively, straightforwardly derived. This includes, for instance, the memory controller (Åkesson et al. 2007). On these resources, we apply latency-rate schedulers, because safe upper bounds on the response times

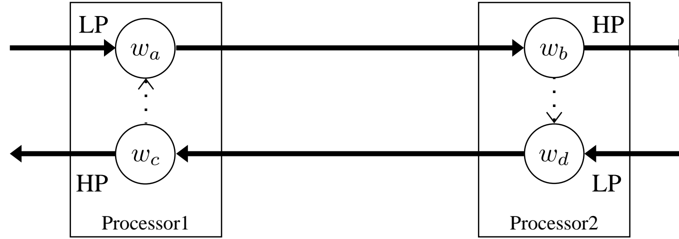


Figure 3.5: Example cyclic resource dependency with static priority pre-emptive scheduling.

of task executions can be derived with latency-rate schedulers.

In order to satisfy our design requirement to characterise the temporal behaviour of jobs, we require that the scheduling and inter-task synchronisation overhead can be bounded independently of the tasks that execute on this resource. This is to ensure that the upper bound on the response time of task executions is independent of tasks that are part of other jobs. This is because also with the classes of schedulers that we apply, the upper bound on the response times still depends on the scheduling and synchronisation overhead.

Not for all budget schedulers can the scheduling overhead be bounded independently of the tasks that execute on this resource. For example, for the constant-bandwidth server the number of times a task is pre-empted in a given time interval depends on the execution times and rates of the other tasks. The constant-bandwidth server is, therefore, a budget scheduler, in case a single pre-emption does not require time. Otherwise, if a pre-emption does require time, then the constant-bandwidth server cannot provide a guaranteed budget in a given time interval to the tasks that is independent of the tasks that execute on this resource. Therefore, in case a pre-emption does take time the constant-bandwidth server is not a budget scheduler. Time-division multiplex, and priority-based budget scheduling (Steine et al. 2009) are examples of budget schedulers for which the number of pre-emptions can be bounded independently of tasks that are part of other jobs.

In our system, the blocking acquisition primitives are implemented by busy-waiting. This is possible because tasks have guaranteed budgets and if one task is busy waiting, then this does not affect the upper bounds on the response times of other tasks. The alternative implementation is to use interrupts. Determining an upper bound on the number of interrupts can be dependent on the tasks that share the resource, therefore it can be difficult to provide a bound on the inter-task synchronisation overhead

that is independent of tasks that are part of other jobs. Furthermore, this alternative implementation with interrupts can be complicated, because both the number of containers on which the task is waiting to arrive as well as the buffer on which the task is waiting for containers can vary from task execution to task execution, where this variation can be determined by values in the processed stream. For example, the task from Listing 3.1 has such data-dependent variation. Our approach of implementing the blocking acquisition primitives by busy waiting results in a straightforward implementation of this data-dependent inter-task synchronisation.

We say that a multiprocessor system is predictable if bounds on the temporal behaviour of a job can be computed at design-time, where these bounds are independent of other jobs. A multiprocessor is said to be composable if the actual temporal behaviour of each job is independent of the other jobs (Hansson et al. 2009a). Even though composability has important benefits in terms of fault containment, it is outside the scope of this thesis.

In (Hansson et al. 2009a) the implementation of an example predictable multiprocessor architecture is presented in great detail. Other examples of predictable multiprocessor systems can be found in (Bekooij et al. 2004; Moonen et al. 2005; Bekooij et al. 2007).

3.4 Timing Constraints

Jobs provide a certain functionality. We have required that the implementation of a job is by a functionally deterministic task graph, which means that the output values of the task graph are completely determined by the input values of this task graph.

We assume that the required functionality is specified on the interfaces of the task graph. For example, in our application domain, the required functionality often follows from standards, which describes the continuous time signals that are input to the input interfaces and/or output of the output interfaces. From the description and specification of these signals, the periods of the input and output interfaces of a job often follow immediately.

However, as we have seen, the task graph together with its interfaces is no longer functionally deterministic. As discussed, the implementation of the job as a task graph provides a certain functionality, while the required functionality is specified on the interfaces of the job with its environment. We assume that the provided functionality and the required functionality are such that the required functionality is correctly implemented, if there is no buffer overflow at the input interfaces and no buffer underrun at the output interfaces. This implies that for the task graph suitable settings need to be determined such that there are always sufficient empty containers at the input interfaces and sufficient full containers at the output

interfaces in order not to have buffer overflow and buffer underrun, respectively. This is a throughput constraint that needs to be satisfied by the task graph mapped on the multiprocessor system.

The task graph mapped on the multiprocessor system not only needs to satisfy the just described throughput constraint, but also the following latency constraint. Consider the initial situation in which the tasks did not yet process any data but are all waiting for containers to arrive. The latency constraint that the task graph should satisfy specifies the maximum time interval from the first start of the input interfaces until the first start of the output interfaces, where the start times of the interfaces are such that there is no buffer overflow at the input interfaces and no buffer underrun at the output interfaces.

Implementation Since unrelated clocks will drift apart (Kopetz 1997) and result in a system that inherently cannot satisfy the just described throughput constraint, we require that in the implementation of the multiprocessor system the clocks of all interfaces are all derived from one single clock.

Analysis In our analysis of the temporal behaviour of a task graph mapped on a multiprocessor system, we will not check for availability of sufficient containers such that there is no buffer overflow or buffer underrun. Instead, during our analysis, we will consider the interfaces as normal tasks that wait on the arrival of sufficient containers and have a constant response time and subsequently check for strictly periodical execution of these tasks.

Discussion Variation in response times and synchronisation behaviour will result in variation in the production times of containers by the task graph. While in literature, for example (Abeni and Butazzo 2004), it is often assumed that stream processing applications have so-called maximum jitter requirements on this variation in production times, in our system all variation is required to be absorbed by the FIFO buffers adjacent to the interfaces. Therefore, in our system, jitter is dealt with in the analysis that computes the required FIFO buffer capacities that guarantee that there are always sufficient containers for the interfaces.

3.5 Resource Constraints

In this thesis, we are addressing part of a programming problem. Given a task graph, given the timing constraints, and given a multiprocessor architecture, the programming problem is to determine (1) a task to processor assignment, (2) scheduler settings and (3) buffer capacities that (i) are fea-

sible given this multiprocessor architecture, on which already other task graphs can be mapped, and that (ii) satisfy the timing constraints.

In this thesis, we assume that the task to processor assignment and the scheduler settings are determined. Given the task to processor assignment and scheduler settings, we determine buffer capacities that satisfy constraints on maximum buffer capacities and timing constraints. In a predictable multiprocessor system, multiple FIFO buffers can be allocated in the same memory. Therefore, the size of the memory is the actual constraint. Taking the size of the container into account, our analysis model can straightforwardly translate a constraint on buffer size into a constraint on buffer capacity and vice versa. However, the fact that we consider a constraint on the maximum capacity per buffer implies that we might need to consider multiple sets of constraints over buffer capacities to find a solution that satisfies the constraint on the memory size.

3.6 Conclusion

In this chapter, we have discussed our assumptions on the inputs to our analysis. The input includes a task graph, execution times, scheduler settings, timing constraints, and resource constraints. We require that the job implemented as a task graph interfaces with its environment by strictly periodic sampling. Furthermore, we require that the task graph is functionally deterministic and we provided sufficient conditions for a functionally deterministic task graph. Tasks call blocking acquisition primitives and the code fragment in between subsequent blocking acquisition primitives is called a non-blocking code-segment. We associate execution times with executions of non-blocking code-segments. Furthermore, we discussed multiprocessor scheduling and concluded that (semi-)static assignment scheduling satisfies our design requirements, because it applies a run-time scheduler per resource. By applying budget schedulers that by construction guarantee resource budgets to tasks, (semi-)static assignment allows, in a cost-effective way, for conservative bounds on the temporal behaviour of jobs, where these bounds are independent of execution times and execution rates of other jobs. Data can be lost at the strictly periodically sampling interfaces of a job. We assume that the specified functionality is provided to the environment of the system if no data is lost at the interfaces of the job. This implies that space should be available on time at the input interfaces and data should be available on time at the output interfaces, which form the timing constraints that the mapping of the job on the multiprocessor system should satisfy. The resource constraints that are taken into consideration in this thesis are constraints on buffer capacities.

Chapter 4

Conservative Dataflow Simulation and Analysis

Abstract – *This chapter defines when a dataflow graph is temporally conservative to a task graph and discusses dataflow simulation and dataflow analysis in detail, which are two existing approaches. We show that, with dataflow simulation, satisfaction of timing constraints can be guaranteed for a single input stream for any functionally deterministic task graph. Furthermore, we show that, with dataflow analysis, satisfaction of timing constraints can be guaranteed for any input stream of task graphs with tasks that have periodic execution rates. A task graph has tasks with periodic execution rates, if the inter-task synchronisation behaviour is independent of the processed data stream.*

In Chapter 3, the requirements on the input to our analysis that determines buffer capacities that satisfy timing and resource constraints are defined. This analysis is done by modelling the task graph by a dataflow graph. In this chapter, we will show that if we require a specific relation between the task graph and the dataflow model, then the dataflow model is temporally conservative to the task graph. Temporally conservative means that if the analysis that is applied on the dataflow model guarantees satisfaction of the timing constraints by the dataflow graph, then also the modelled task graph will satisfy the timing constraints.

In this chapter, we discuss dataflow simulation and dataflow analysis. With dataflow simulation, we can provide guarantees on the satisfaction

of timing constraints for any functionally deterministic task graphs, for a single input stream. While dataflow analysis can only provide guarantees on the satisfaction of timing constraints for task graphs with inter-task synchronisation behaviour that is independent of the processed data stream, these guarantees are valid for any input stream.

This chapter assumes that there is no run-time scheduling. In Chapter 5, we extend both approaches and show that dataflow graphs can conservatively model the execution of any functionally deterministic task graph on resources that are scheduled at run-time by budget schedulers.

Both dataflow simulation as well as dataflow analysis only provide guarantees on the satisfaction of timing constraints for given buffer capacities. In Chapter 6, we present a new dataflow model that is more expressive than the existing dataflow models that allow for guarantees for all input streams. In fact this new dataflow model subsumes existing dataflow models that are amenable to analysis. This new dataflow model provides guarantees on the satisfaction of timing constraints for task graphs in which the inter-task synchronisation behaviour does depend on the processed data stream. Furthermore, we present, in Chapter 6, an algorithm that can be applied to this new dataflow model to compute buffer capacities that satisfy given timing and resource constraints.

The outline of this chapter is as follows. To prevent repetition, this chapter starts with the most expressive model considered in this thesis, functionally deterministic dataflow, and subsequently adds restrictions to obtain less expressive models. Functionally deterministic dataflow is defined in Section 4.1. Section 4.2 presents a sufficient condition on the relation between task graph and dataflow graph to have a dataflow graph that is temporally conservative to the task graph. Section 4.3 shows that guarantees on the satisfaction of timing constraints can be obtained for a given input stream and for given buffer capacities through simulation of the dataflow graph. The remainder of this chapter discusses a dataflow model that allows to guarantee satisfaction of timing constraints for any input stream. This dataflow model with its associated analysis techniques is defined in Section 4.4. Section 4.5 explains that the model is conservative even though there are various sources of dynamism that are present in the implementation but not in the model. This chapter is concluded with a summary and discussion of the concepts presented in this chapter in Section 4.6.

4.1 Functionally Deterministic Dataflow

This section introduces functionally deterministic dataflow and uses this model to show that our task graphs are functionally deterministic, i.e. the output values of the task graph are completely determined by the input values. Furthermore, we define how time is included in functionally

deterministic dataflow graphs and introduce the two essential properties of monotonic and linear temporal behaviour of functionally deterministic dataflow graphs.

In the next sections, we show that we can simulate this dataflow model to obtain guarantees on the satisfaction of timing constraints for one input stream. In this section, we start by introducing dataflow graphs and subsequently define functionally deterministic dataflow graphs and show that our task graphs are functionally deterministic. We conclude this section by introducing time into functionally deterministic dataflow and showing that functionally deterministic dataflow is temporally linear and monotonic.

4.1.1 Dataflow

A dataflow graph is a graph that consists of actors that communicate tokens over queues. As defined in (Lee and Parks 1995), an actor has firings and each firing of an actor maps tokens from input queues into tokens on output queues. A set of firing rules specifies when an actor can fire. More specifically, a firing rule is a condition that specifies the tokens that need to be present on specific input queues before the actor can fire. A firing consumes, i.e. removes, input tokens and produces output tokens.

Definition 4.1 (Actor) *An actor has an unbounded sequence of actor firings. An actor has a set of firing rules that specify when the actor can fire its next firing.*

Definition 4.2 (Actor firing) *An actor firing has a firing rule and consumes the input tokens that are guaranteed to be present by satisfaction of this firing rule. Furthermore, an actor firing produces output tokens, where these output tokens have a certain specified relation with the consumed input tokens.*

Definition 4.3 (Firing rule) *A firing rule is a condition associated with one actor that specifies the tokens that need to be present on specific input queues before the actor can fire.*

Definition 4.4 (Dataflow graph) *A dataflow graph is a directed graph that consists of a (finite) set of actors interconnected by a (finite) set of queues. Queues hold tokens.*

4.1.2 Functionally Deterministic Dataflow

In this section, we present the sufficient conditions, from (Lee and Parks 1995), for a dataflow graph to be functionally deterministic, i.e. output values are only determined by input values. This will enable us to show that task graphs are functionally deterministic, if all acquisition primitives of tasks are blocking acquisition primitives and all tasks are functional.

A sufficient condition for an actor to be functionally deterministic is that each firing of this actor is functional and that the set of firing rules is sequential. A firing is functional if it is side-effect free, i.e. the produced tokens in a firing are a function of the consumed tokens in that firing. A set of firing rules is sequential if there exists a pre-defined order in which the firing rules can be applied.

A dataflow graph is allowed to contain cycles. In general, initial tokens need to be placed on queues of these cycles in order to enable a non-terminating execution of the dataflow graph. We say that a dataflow graph is functionally deterministic if the tokens produced on the output queues of the graph are completely determined by tokens from the input queues and initial tokens in the dataflow graph. Because actors are interconnected by queues, if all actors in the dataflow graph are functionally deterministic, then also the dataflow graph is functionally deterministic (Lee and Parks 1995).

4.1.3 Functionally Deterministic Task Graphs

We require that in a task graph all acquisition primitives of tasks are blocking acquisition primitives and that all tasks are functional tasks. By showing a one-to-one relation between these task graphs and functionally deterministic dataflow graphs, we will show that these task graphs are functionally deterministic.

We associate with each task in the task graph a unique actor in the dataflow graph. Then, we associate with each FIFO buffer in the task graph two queues in opposite directions connecting the corresponding actors. One of these two queues models the flow of full containers, while the other queue models the flow of empty containers. A full or empty container that is present when the task graph is started is reflected in the dataflow graph by an initial token on the corresponding queue. An example is shown in Figure 4.1, where actor v_1 corresponds with task w_1, actor v_2 corresponds with task w_2, and the two queues with d initial tokens correspond with the FIFO buffer that contains d initially empty containers.

We associate the state of a task, which is carried from one non-blocking code-segment to the next non-blocking code-segment, with one token on a queue from and to the corresponding actor, i.e. one initial token on a self-edge. If a set of firing rules of the actor firing is such that depending on the value of the token on this self-edge the number of tokens to be consumed from each other queue is completely determined, then the set of firing rules is sequential. It is clear that a set of firing rules can be constructed such that the number of tokens consumed from various queues depending on the value of this token has a one-to-one correspondence with the number of containers acquired from various buffers depending on the state of the task. This implies that each task is associated with a functionally deterministic actor, and that each buffer is associated with two queues. Because there

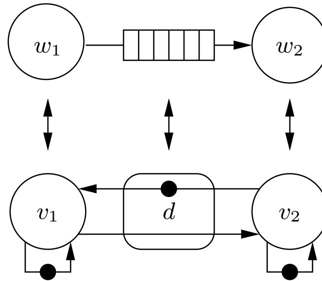


Figure 4.1: One-to-one correspondence between example task graph and dataflow graph.

is a one-to-one relation between the task graph and a functionally deterministic dataflow graph, we can say that also the task graph is functionally deterministic.

We see that our task graphs, as defined in Section 3.1, can be modelled by dataflow graphs in which each actor has a sequential firing rule and each actor firing is functional. These two properties are sufficient for a dataflow graph to be functionally deterministic but not necessary. However, we will associate functionally deterministic dataflow graphs exclusively with dataflow graphs in which all actors have sequential firing rules and all actor firings are functional. Even though this is not strictly correct, because it treats a sufficient condition as a necessary condition, it clarifies the exposition and does not change the essence of our results.

4.1.4 Timed Dataflow Graph

If a dataflow process is functionally deterministic, then the tokens produced by a dataflow process are only determined by the tokens arriving on the input queues. This implies that the produced tokens are independent of the arrival times of tokens on the input queues of dataflow processes.

As in (Sriram and Bhattacharyya 2000), we can extend a functionally deterministic dataflow model to include time by separating the token consumption and token production of each actor firing. Instead of defining an actor firing as an atomic action in which tokens are consumed and produced, we define an actor firing as two atomic actions. One action consumes tokens, while the other action produces tokens. Since the produced tokens are a function of the consumed tokens, we require that the action that produces tokens is not before the action that consumes tokens. It is clear that this does not change the functionality.

Definition 4.5 (Actor enabling) *An actor is enabled if it has a firing rule that is satisfied.*

Definition 4.6 (Actor firing enabling time) *The enabling time of an actor firing is the earliest time at which this actor firing is enabled.*

Definition 4.7 (Actor firing start) *The start of an actor firing is the event at which consumption of all tokens in an actor firing takes places.*

Definition 4.8 (Actor firing finish) *The finish of an actor firing is the event at which production of all tokens in an actor firing takes place.*

Definition 4.9 (Timed actor firing) *A timed actor firing consists of an actor firing start event and an actor firing finish event, where the actor firing finish event occurs later than the actor start event.*

Definition 4.10 (Actor firing duration) *The difference in time between the finish and start of an actor firing is the actor firing duration.*

Definition 4.11 (Dataflow graph schedule) *A dataflow graph schedule associates start times with actor firings that are later than or equal to the enabling times of these actor firings.*

4.1.5 Linearity

A dataflow graph has linear temporal behaviour if on any chain of actor firings that each depend on each other, the addition of a delay Δ to the finish time of i_1 at the start of this chain cannot lead to any delay in finish times on this causality chain that is larger than Δ. This definition is formalised in Definition 4.12. We first introduce a number of required concepts, before we discuss Definition 4.12 in more detail.

Let $\sigma(G, \rho, u, \delta)$ be a schedule of dataflow graph G, given that actor firing i has firing duration $\rho(i)$, and has a difference between start time and enabling time $u(i)$, where, furthermore, queue j has a number of initial tokens given by $\delta(j)$. For any actor firing i_d, let $f(i_d)$ be the finish time of i_d in schedule $\sigma(G, \rho, u, \delta)$, and let $f'(i_d)$ be the finish time of i_d in schedule $\sigma(G, \rho', u', \delta)$.

Let us say that actor firing i_c is directly causally dependent on actor firing i_b, if there is a token that is consumed by i_c and produced by i_b. Let us say that actor firing i_c is causally dependent on actor firing i_a if there exists a chain of directly causally dependent actor firings from i_a to i_c, let us call this a causality chain. Let $I(i_a)$ be the set of actor firings that are causally dependent on actor firing i_a. Let us partition $I(i_a)$ in sets $I_1(i_a), \ldots, I_n(i_a)$ where the set $I_k(i_a)$, with $1 \leq k \leq n$, consists of actor firings that have a causality chain of length k from actor firing i_a. Let $J(i_a)$ be the set of actor firings on which actor firing i_a is causally dependent. Let us also partition $J(i_a)$ in sets $J_1(i_a), \ldots, J_n(i_a)$ where the set $J_k(i_a)$, with $1 \leq k \leq n$, consists of actor firings that have a causality chain of length k to actor firing i_a.

In Definition 4.12, we take an actor firing i_x and look for a set of actor firings $J_k(i_x)$ such that i_x is causally dependent on the firings from this set and such that for all actor firings on causality chains in between $J_k(i_x)$ and i_x no delay is introduced, i.e. $\rho'(i) \leq \rho(i) \wedge u'(i) \leq u(i)$. We bound the delayed finish times of actor firings in $J_k(i_x)$ as maximally Δ larger than the reference finish times. We show that because no actor firing on a causality chain from an actor firing in $J_k(i_x)$ to i_x introduces additional delay, the finish time of i_x is also maximally delayed by Δ.

Definition 4.12 (Linearity) *A dataflow graph has linear temporal behaviour if and only if for all $\sigma(G, \rho, u, \delta)$ and $\sigma(G, \rho', u', \delta)$*

$$\begin{aligned} \forall i_x \bullet (\forall i_k \in J_k(i_x) \bullet f'(i_k) \leq f(i_k) + \Delta \wedge \\ \forall i \in \bigcup_{i_k \in J_k(i_x)} I(i_k) \bullet \rho'(i) \leq \rho(i) \wedge u'(i) \leq u(i)) \Rightarrow \\ f'(i_x) \leq f(i_x) + \Delta \end{aligned} \tag{4.1}$$

with $\Delta \geq 0$.

Theorem 4.1 *Functionally deterministic dataflow graphs have linear temporal behaviour.*

Proof. Given an actor firing i_x, we find the set of actor firings $J_k(i_x)$ that have causality chains with a length of k to actor firing i_x, and that furthermore have for all causally dependent actor firings i that $\rho'(i) \leq \rho(i)$ and $u'(i) \leq u(i)$. Given this set of actor firings $J_k(i_x)$, we need to show that $\forall i_k \in J_k(i_x) \bullet f'(i_k) \leq f(i_k) + \Delta \Rightarrow f'(i_x) \leq f(i_x) + \Delta$. We show that this holds for any $i_k \in J_k(i_x)$ by induction over the causality chains from i_k to i_x. Actor firings in the set $I_a(i_k) \cap J_{k-a}(i_x)$ have a dependency distance of a from i_k and are all on causality chains from i_k to i_x.

Base step. We need to show that $\forall i_1 \in I_1(i_k) \cap J_{k-1}(i_x) \bullet f'(i_k) \leq f(i_k) + \Delta \Rightarrow f'(i_1) \leq f(i_1) + \Delta$. This is true, because of the following. Because actor firings i_1 are on a causality chain from i_k to i_x, actor firing i_1 is only causally dependent on firings in $J_k(i_x)$. With finish times of actor firings in $J_k(i_x)$ maximally delayed by Δ, the enabling time of i_1 is maximally delayed by Δ. With $\rho'(i_1) \leq \rho(i_1)$ and $u'(i_1) \leq u(i_1)$, we have that the finish time of i_1 is maximally delayed by Δ.

Induction step. We need to show that $\forall i_{n+1} \in I_{n+1}(i_k) \cap J_{k-n-1}(i_x)$ $\forall i_n \in I_n(i_k) \cap J_{k-n}(i_k) \bullet f'(i_n) \leq f(i_n) + \Delta \Rightarrow f'(i_{n+1}) \leq f(i_{n+1}) + \Delta$, with $n + 1 \leq k$. Actor firings i_{n+1} only depend on actor firings $i_n \in I_n(i_k) \cap J_{k-n}(i_x)$. The finish time of any such i_n has a maximal delay of Δ. This implies that the token production times of any such i_n is maximally delayed by Δ, which implies that the enabling time of any firing i_{n+1} is maximally delayed by Δ. With $\rho'(i_{n+1}) \leq \rho(i_{n+1})$ and $u'(i_{n+1}) \leq u(i_{n+1})$, we have that the finish time of i_{n+1} is maximally delayed by Δ. □

4.1.6 Monotonicity

A dataflow graph has monotonic temporal behaviour, if a decrease in actor firing durations or a decrease in the time between enablings and starts of actor firings or an increase in the number of initial tokens allows for a schedule that does not have later start times. Monotonic temporal behaviour is not to be confused with monotonicity of dataflow graphs over input and output sequences of values from (Lee and Parks 1995).

Definition 4.13 (Monotonicity) *A dataflow graph has monotonic temporal behaviour if and only if for all* $\sigma(G, \rho, u, \delta)$ *and* $\sigma(G, \rho', u', \delta')$

$$(\forall i, j \bullet \rho'(i) \leq \rho(i) \wedge u'(i) \leq u(i) \wedge \delta'(j) \geq \delta(j)) \Rightarrow (\forall i \bullet f'(i) \leq f(i)) \quad (4.2)$$

where i *denotes an actor firing and* j *denotes a queue, and with* $f'(i)$ *and* $f(i)$ *the finish times of actor firing* i *in schedule* $\sigma(G, \rho', u', \delta')$ *and schedule* $\sigma(G, \rho, u, \delta)$*, respectively.*

As shown in the following theorem, monotonic temporal behaviour follows from linear temporal behaviour, except that additionally an increase in the number of initial tokens is shown to not result in later start times.

Theorem 4.2 *Functionally deterministic dataflow graphs have monotonic temporal behaviour, given that the additional initial tokens do not change the functional behaviour of the dataflow graph.*

Proof. Given a queue j, then we have that the enabling times of firings of the actor that produces on this queue are independent of the number of tokens on j, and, therefore, of the number of initial tokens on j. Because its enabling times are independent of the number of initial tokens also the start and finish times of the firings of this actor are independent of the number of initial tokens on j. The actor that consumes from j is enabled by the presence of tokens on j. Placing a token initially on a queue means that this token now has an arrival time that is smaller than or equal to its arrival time in case it was not initially placed on this queue. A smaller than or equal arrival time of this token can only result in a smaller than or equal enabling time of the actor firing that requires the presence of this token. Let us call this actor firing i_k. A smaller than or equal enabling time of i_k can only result in a smaller than or equal finish time of i_k. This is because $\rho'(i_k) \leq \rho(i_k)$ and $u'(i_k) \leq u(i_k)$. Because Theorem 4.1 tells that functionally deterministic dataflow graphs have linear temporal behaviour, $f'(i_k) \leq f(i_k)$ implies $f'(i_x) \leq f(i_x)$, with i_x any firing that is causally dependent on i_k. This is because, for any firing i, we have $\rho'(i) \leq \rho(i)$ and $u'(i) \leq u(i)$. Therefore, the premises of Equation (4.2) implies finish times of actor firings that are smaller than or equal to the reference finish times of these actor firings. □

In the next section, we apply the fact that functionally deterministic dataflow graphs have monotonic temporal behaviour to introduce sufficient conditions on the dataflow graph such that upper bounds on container arrival times are given by the arrival times of the corresponding tokens.

4.2 Conservative Dataflow Modelling

In this section, we present sufficient conditions on the relation between the dataflow model and the task graph such that the dataflow model allows us to derive conservative times at which sufficient containers are available for non-blocking code-segments to start. The first condition is that there is the following one-to-one correspondence between containers and tokens.

Property 4.1 *For each buffer in the task graph, there are two unique queues in the dataflow graph. Furthermore, for each container in the task graph, there is one token in the dataflow graph.*

Let $a(c)$ be the arrival time of container c, and let $\hat{a}(c)$ be the arrival time of the token that corresponds to container c. In the next definition, consumptions destroy containers and tokens and productions create containers and tokens. Given that Property 4.1 holds, the following definition says that a dataflow graph is temporally conservative to a task graph if the fact that container arrival times on input FIFO buffers of the task graph are bounded from above by token arrival times on the corresponding queues in the dataflow graph implies that container arrival times on all FIFO buffers are bounded from above by token arrival times on their corresponding queues.

Definition 4.14 *Given that Property 4.1 holds for dataflow graph G and task graph T. This dataflow graph G is temporally conservative to T if and only if*

$$(\forall c_i \in C_I \bullet a(c_i) \leq \hat{a}(c_i)) \implies (\forall c \in C \bullet a(c) \leq \hat{a}(c)) \tag{4.3}$$

where C_I is the set of containers that are either initially present or arrive on input buffers of T and C is the set of all containers.

By constructing the same dataflow graph as in Section 4.1.3, we can derive a requirement in terms of non-blocking code-segments and actor firings that is more straightforward to verify than the more implicit requirement given by Equation (4.3). The required one-to-one correspondence between tasks and actors and between the task executions and actor firings is more precisely described in the following property.

Property 4.2 *For each task w in the task graph there is a unique actor v in the dataflow graph. For the relation between the input and output*

buffers of task w and the input and output queues of actor v, Property 4.1 holds. Furthermore, for each task execution i of task w there is a unique firing rule in the set of firing rules of actor v that is only satisfied if (1) the tokens that correspond with the containers consumed by task execution i are present, and (2) the token that signals the finish of the previous actor firing is present. Furthermore, satisfaction of the firing rule that corresponds with task execution i enables an actor firing that computes the same function as task execution i except that tokens are produced in an atomic action on queues instead of containers being produced on buffers.

Let $e(w, i)$ be the time at which execution i of task w is externally enabled, which means that sufficient containers are available on all adjacent buffers.

Definition 4.15 (Task execution external enabling) *A task execution is externally enabled if there are sufficient containers present in the required FIFO buffer to let the acquistion primitive of this task execution return successfully.*

Definition 4.16 (Task execution external enabling time) *The earliest time at which the task execution is externally enabled is the external enabling time.*

Let $\hat{e}(v, i)$ be the external enabling time of the corresponding actor firing, which is the earliest time at which the tokens are present that correspond with the required containers of the modelled execution i of task w. This means that the external enabling time is independent of the presence of the token signalling the finish of the previous actor firing.

Definition 4.17 (Actor firing external enabling) *On all input queues of this actor that are different from the queues that are from this actor, there are sufficient tokens to satisfy the firing rule on these queues.*

Definition 4.18 (Actor firing external enabling time) *The external enabling time is the earliest time at which the actor firing is externally enabled.*

Further, let $f(w, i)$ be the finish time of execution i of task w and let $\hat{f}(v, i)$ be the finish time of the corresponding actor firing.

The following theorem provides a more straightforward check on the dataflow graph, then the requirement specified in Definition 4.14. This is because this check is on external enabling and finish times of actor firings instead of token arrival times. Intuitively, it should hold that if tokens arrive not earlier than containers on the input queues than they should not arrive earlier on the output queues. The relation between task graph and dataflow graph that is considered in Theorem 4.3 is illustrated in Figure 4.2.

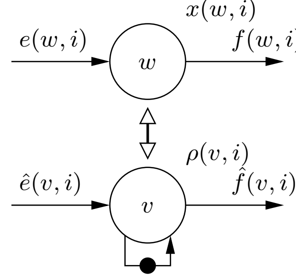


Figure 4.2: Relation between task graph and dataflow graph that satisfies Property 4.2.

Theorem 4.3 *Given that Property 4.2 holds for task graph T and dataflow graph G. If Equation* (4.4) *holds for any execution i of any task w, then G is temporally conservative to T.*

$$e(w, i) \leq \hat{e}(v, i) \Rightarrow f(w, i) \leq \hat{f}(v, i) \tag{4.4}$$

Actor v corresponds to task w.

Proof. Dataflow graph G is temporally conservative to task graph T, if given a starting situation in which all token arrival times are conservative no actor firing can lead to token arrival times that are not conservative. In G, actor firings consume and produce the same amount of tokens as their corresponding task executions consume and produce containers. Furthermore, task executions consume containers not before their start and produce containers not after their finish, while actor firings consume tokens at their start and produce tokens at their finish. This implies that if Equation (4.4) holds, then no actor firing produces its tokens earlier than the corresponding task execution produces its containers. This implies that given token arrival times that are conservative every actor firing leads to token arrival times that are again conservative, which implies that G is temporally conservative to T. □

4.3 Dataflow Simulation

We can now obtain guarantees on satisfaction of the timing constraints for a single input stream for any functionally deterministic task graph. We assume that buffer capacities are given and that the task graph executes on resources without resource sharing.

First, we determine the execution time of each task execution. Subsequently, we execute the task graph in a discrete time simulation environment, by letting each task execution consume all its containers at the

start, wait until the simulation time has advanced by the execution time of this task execution and produce all containers. In this way, we basically create the corresponding dataflow graph while executing the task graph. Theorem 4.3 tells us that the arrival times observed in this simulation are conservative. If the arrival times observed in this simulation satisfy the timing constraints, then for this input stream it is guaranteed that the task graph executing on the multiprocessor system also satisfies the timing constraints.

We have now obtained guarantees on the satisfaction of timing constraints for a single input stream. We have no guarantees for other input streams. In fact for other streams the task graph can even deadlock with the given buffer capacities. This is intrinsic to the expressiveness of this model. Because functionally deterministic dataflow can model any function it is Turing-complete. Because the halting problem is undecidable over Turing machines, deadlock-freedom is in general undecidable for functionally deterministic dataflow graphs. This means that there is no algorithm that given any functionally deterministic dataflow graph can efficiently compute whether there is an input stream for which it deadlocks.

The remainder of this chapter deals with a restricted dataflow model in which the number of tokens produced and consumed is fixed for all firings of the dataflow actors. For this dataflow model, deadlock-freedom is decidable, and we are able to provide guarantees on the satisfaction of the timing constraints for all input streams.

4.4 Dataflow Analysis

In this section, we introduce Single-Rate Dataflow and its associated analysis technique called maximum cycle mean analysis. Furthermore, we informally discuss multi-rate dataflow and cyclo-static dataflow, which are two dataflow models that are more expressive than single-rate dataflow. However, maximum cycle mean analysis is also applicable for these models, after translation to a corresponding single-rate dataflow graph.

We first define a number of general graph concepts and sets of numbers. Let E be a finite set of labeled directed edges. We say that edge $e \in E$ is an output edge of $\mathbf{src}(e)$, which is the source vertex of e, and an input edge of $\mathbf{dst}(e)$, which is the destination vertex of e. A path in a directed graph is a finite, non-empty sequence $\langle e_1, e_2, \ldots, e_n \rangle$ with $e_i \in E$ and $\mathbf{dst}(e_1) = \mathbf{src}(e_2)$, $\mathbf{dst}(e_2) = \mathbf{src}(e_3)$, $\ldots$, $\mathbf{dst}(e_{n-1}) = \mathbf{src}(e_n)$. A path from a vertex to itself is called a cycle. If a cycle visits a vertex only once then it is called a simple cycle. A directed graph is a strongly connected directed graph, if for every pair of vertices v_i and v_j, with $v_i \neq v_j$, there is a path from v_i to v_j and a path from v_j to v_i.

Let $\mathbb{R}$ denote the set of real numbers, and let $\mathbb{R}^+$ denote the set of non-negative real numbers, i.e. $\mathbb{R}^+ = \{r | r \geq 0, r \in \mathbb{R}\}$. Further, let $\mathbb{N}$

denote the set of non-negative natural numbers, and let $\mathbb{N}^*$ denote the set of positive natural numbers, i.e. $\mathbb{N}^* = \{n | n \geq 1, n \in \mathbb{N}\}$.

4.4.1 Single-Rate Dataflow

A Single-Rate Dataflow (SRDF) Graph G, also known as a computation graph (Reiter 1968) or as a homogeneous synchronous dataflow graph (Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2000) or as a marked graph (Commoner et al. 1971), is a directed multi-graph $G = (V, E, \rho, \delta)$. We have that V is a finite set of vertices, and that E is a finite set of labeled directed edges. The vertices in an SRDF graph represent actors and the edges represent queues of tokens. The number of tokens on the queue represented by edge e is given by $\delta(e)$, with $\delta : E \rightarrow \mathbb{N}$. In an SRDF graph, an actor v has a single actor firing duration that is given by $\rho(v)$, with $\rho : V \rightarrow \mathbb{R}^+$. Furthermore, in an SRDF graph, in every firing, actors produce a single token on all adjacent output queues and consume a single token on all adjacent input queues.

Of an SRDF graph, we can compute the so-called maximum cycle mean (Reiter 1968; Sriram and Bhattacharyya 2000). Of SRDF graph G, let $O(G)$ be the set of simple cycles of G. Further, with simple cycle $o \in O(G)$, let $V(o)$ be the set of vertices on o and let $E(o)$ be the set of edges on o. The maximum cycle mean of G is defined by

$$\mu(G) = \max_{o \in O(G)} \frac{\sum\limits_{v \in V(o)} \rho(v)}{\sum\limits_{e \in E(o)} \delta(e)} \tag{4.5}$$

A cycle of G that has the maximum ratio of firing times and tokens and determines the maximum cycle mean is called a critical cycle of G.

From (Reiter 1968; Moreira and Bekooij 2007), we know that G has a strictly periodic schedule in which all actors of G have a difference between subsequent starts of ϖ, if and only if $\varpi \geq \mu(G)$. Therefore, $\mu(G)$ is the minimal possible period of any strictly periodic schedule of graph G.

4.4.2 Dataflow Analysis and Expressiveness

We have just defined SRDF graphs and the associated analysis technique of maximum cycle mean analysis to compute the minimum period for which a strictly periodic schedule of the dataflow graph exists. However, SRDF is a restrictive model in which all actors in the graph fire at the same rate. Figure 4.3 shows examples of two more expressive dataflow models, Multi-Rate Dataflow (MRDF) in Figure 4.3(a), and Cyclo-Static Dataflow (CSDF) in Figure 4.3(b). MRDF is also known as Synchronous Dataflow (Lee and Messerschmitt 1987), and extends SRDF by allowing actors to fire at different but fixed relative rates. CSDF (Bilsen et al. 1996) extends

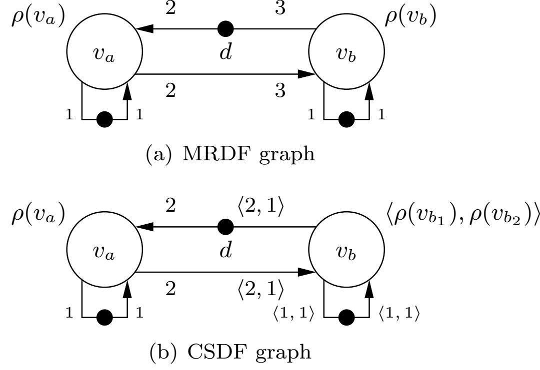


Figure 4.3: Examples dataflow graphs for which maximum cycle mean analysis can be applied on their corresponding SRDF graphs.

MRDF by specifying for each actor a fixed length sequence of phases, or states, through which the actor cycles, i.e. while an MRDF actor has the same behaviour in every firing a CSDF actor has a behaviour that varies in a cyclic fashion, where this behaviour includes the token production and consumption quanta and the firing duration.

In an SRDF graph, each actor produces and consumes a single token on all adjacent queues in every firing. In an MRDF graph, actors have positive token production and consumption quanta, i.e. in every firing a positive number of tokens are produced and consumed on all adjacent queues. In a CSDF graph, actors have a fixed finite sequence of token production and consumption quanta that is infinitely often repeated. While the sum of the quanta over this finite sequence is required to be a fixed positive quantum, a single firing of a CSDF actor produces and consumes a fixed non-negative quantum on particular adjacent queues. The fixed token production and consumption quanta together with the fixed number of tokens in the graph implies that the dependencies between firings of different actors in MRDF and CSDF graphs is fixed. This means that, in MRDF and CSDF graphs, the enabling of a particular actor firing always depends on the token production of the same set of other actor firings. These dependencies between MRDF and CSDF actor firings can be encoded in an SRDF graph, and (Lee and Messerschmitt 1987; Sriram and Bhattacharyya 2000) and (Bilsen et al. 1996) provide algorithms for this translation. This enables analysis techniques that are defined for SRDF graphs, such as maximum cycle mean analysis, to be applicable for MRDF and CSDF graphs, i.e. after translation. Note, however, that information is lost in this translation. An SRDF graph has actors, queues and tokens, but no concept that allows to group different actors to one original MRDF

or CSDF actor and no concept to group different queues and tokens to one original MRDF or CSDF queue. Another problematic aspect is that the SRDF graph that corresponds to an MRDF or CSDF graph can be exponentially larger than the MRDF or CSDF graph (Pino and Lee 1995). This implies that any algorithm that has (pseudo)-polynomial complexity in terms of the SRDF graph has exponential complexity in terms of the MRDF or CSDF graph. Maximum cycle mean analysis is an example of an algorithm that has pseudo-polynomial complexity in terms of the SRDF graph (Dasdan 2004).

For SRDF actors, a strictly periodic schedule exists that has a period equal to the maximum cycle mean. Also for MRDF and CSDF graphs, the maximum cycle mean is the minimal period in which the graph can revisit the same state, where the state is given by the token distribution over the queues and for each actor the time interval since this actor started (Stuijk et al. 2008). MRDF and CSDF actors fire multiple times within this minimal period given by the maximum cycle mean and, in general, a multi-dimensional periodic schedule is required to obtain a schedule that attains this minimal period (Verhaegh et al. 2001).

4.5 Applying Dataflow Analysis to Task Graphs

In this section, we show how the temporal behaviour of task graphs with a specific inter-task synchronisation behaviour can be conservatively modelled and analysed by single-rate dataflow graphs. We start with describing how task graphs that consist of two tasks communicating over a single buffer can be modelled by a single-rate dataflow graph. For these task graphs, the sufficient relation between dataflow graph and task graph as discussed in Section 4.2 is applicable. However, we will need to generalise this relation in order to model a task graph with a general topology. This is because a task that is adjacent to two buffers has at least two acquisition primitives, i.e. at least one for each buffer. This implies that subsequent task executions consume and produce containers from and on different buffers, while in the single-rate dataflow model every actor firing produces and consumes one token on each adjacent queue. This requires a generalisation of the relation between dataflow graph and task graph from Section 4.2 to have a one-to-one relation between actor firings and sequences of task executions instead of a one-to-one relation between actor firings and single task executions. We will start this section by discussing task graphs that can be modelled with a dataflow graph such that there is a one-to-one relation between actor firings and single task executions.

4.5.1 Modelling Task Executions

By three examples in which a producer-consumer task graph has increasingly dynamic temporal behaviour, we explain and illustrate why the data-

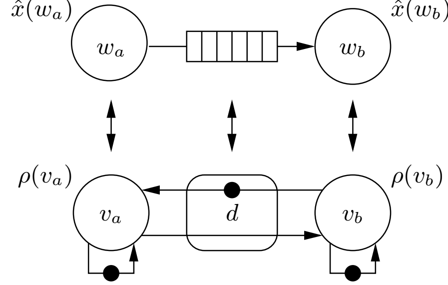


Figure 4.4: Task graph and single-rate dataflow graph.

flow model that lacks this dynamic behaviour conservatively models the temporal behaviour of these task graphs.

Consider task graph T from Figure 4.4. This task graph has two tasks, a data producing task w_a and a data consuming task w_b that communicate data values through containers of FIFO buffer b_{ab}. In every task execution, task w_a acquires one empty container from b_{ab} and releases one full container on b_{ab}. Similarly, in every task execution, task w_b acquires one full container from b_{ab} and releases one empty container on b_{ab}. Execution i of task w_a has execution time $x(w_a, i)$, and execution j of task w_b has execution time $x(w_b, j)$. Let $\hat{x}(w_a)$ be an upper bound on the execution time of any execution of task w_a, i.e. $\hat{x}(w_a) \geq x(w_a, i)$ for all i, and let $\hat{x}(w_b)$ be an upper bound on the execution time of any execution of task w_b, i.e. $\hat{x}(w_b) \geq x(w_b, j)$ for all j. Initially, all containers in buffer b_{ab} are empty. The number of containers in buffer b_{ab} is denoted by d. Both tasks execute on their private resources.

Figure 4.4 also shows single-rate dataflow graph G. This dataflow graph is temporally conservative to task graph T, because of the following. Buffer b_{ab} corresponds with two edges e_{ab} and e_{ba} in opposite direction, and for each (initially empty) container there is one token on e_{ba}, i.e. $\delta(e_{ab}) = 0$ and $\delta(e_{ba}) = d$. Task w_a corresponds with actor v_a and task w_b corresponds with actor v_b. The firing rule of actor v_a requires that one token on edge e_{ba} is present, which corresponds with the empty container acquired by w_a, and the firing rule of v_b requires that one token on edge e_{ab} is present, which corresponds with the full container acquired by w_b. The fact that executions of a single task cannot execute concurrently to themselves is modelled by the edges e_{aa} and e_{bb} that both have one initial token, thereby prohibiting auto-concurrent firing of actors v_a and v_b. Therefore, Property 4.2 holds for task graph T and dataflow graph G. Given that actor v_a has a firing duration of $\hat{x}(w_a)$, i.e. $\rho(v_a) \geq \hat{x}(w_a)$, and actor v_b has a firing duration of $\hat{x}(w_b)$, i.e. $\rho(v_b) \geq \hat{x}(w_b)$, dataflow graph G is temporally conservative to task graph T by Theorem 4.3.

Dataflow graph G has three simple cycles, $\langle e_{aa} \rangle$, $\langle e_{bb} \rangle$, and $\langle e_{ab}, e_{ba} \rangle$. The maximum cycle mean of G is, therefore, given by Equation (4.6).

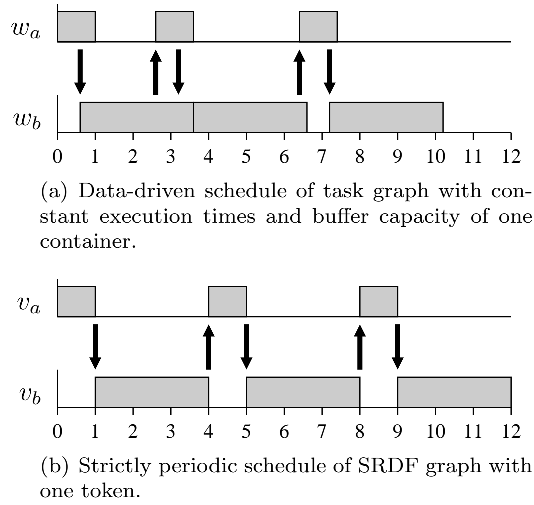
$$\mu(G) = \max \left(\left\{ \frac{\rho(v_a)}{1}, \frac{\rho(v_b)}{1}, \frac{\rho(v_a) + \rho(v_b)}{d} \right\} \right) \tag{4.6}$$

We will now show three examples that show three sources of dynamicity that is present in the task graph but not necessarily in the dataflow model, (1) variation in production and consumption times of containers, (2) data-driven schedule of task graph versus constructed schedule of dataflow graph, and (3) variation in execution times. The dataflow analysis provides conservative bounds on the arrival times of containers in the task graph, because of the required one-to-one relation between task graph and dataflow graph and because the dataflow graph has monotonic temporal behaviour.

Example 4.1 Let us have a constant execution time of one for task w_a, i.e. $\hat{x}(w_a) = 1$, a constant execution time of three for task w_b, i.e. $\hat{x}(w_b) = 3$, and a buffer capacity equal to one container, i.e. $d = 1$. Then, this leads to the situation that, in the data-driven schedule of the task graph, task w_b has a maximum difference between subsequent starts of four time units. The schedule of this task graph does not need to be periodic, even though we have constant execution times. This is because the time between the start of a task execution and the time at which containers are produced, i.e. released, can vary. An example data-driven schedule of this task graph is shown in Figure 4.5(a), in which the arrows denote container production times.

Given these execution times and buffer capacities, the maximum cycle mean of the SRDF graph, as given by Equation (4.6), is $\mu(G) = 4$. This implies that a strictly periodic schedule exists for SRDF graph G, where for every actor the difference between subsequent start times equals 4. This schedule is shown in Figure 4.5(b), in which the arrows denote token production times. Comparing the token production times in Figure 4.5(b) and the container production times in Figure 4.5(a), we see that, in this example, the token production times are conservative container production times. From the maximum cycle mean analysis, we learn that cycle $\langle e_{ab}, e_{ba} \rangle$ is the critical cycle of this graph.

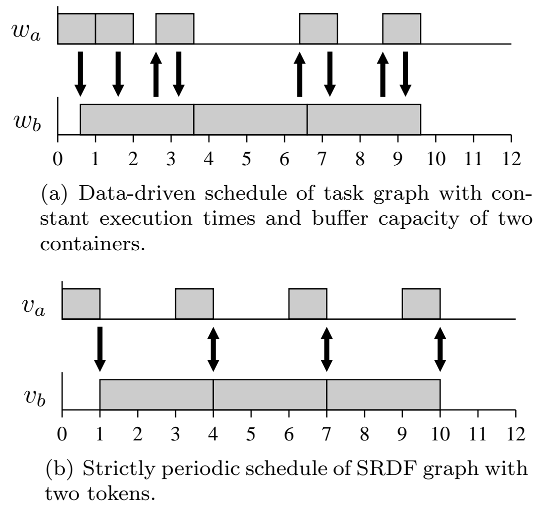
Example 4.2 Given the task graph and dataflow graph from Example 4.1, let us increase the buffer capacity from one container to two containers, i.e. $d = 2$. The data-driven schedule of the resulting task graph is shown in Figure 4.6(a). In this schedule, we see that task w_b can start a next task execution immediately after it has finished a task execution. Task w_b will, therefore, have a difference between subsequent starts equal to three time units.



(a) Data-driven schedule of task graph with constant execution times and buffer capacity of one container.

(b) Strictly periodic schedule of SRDF graph with one token.

Figure 4.5: Schedules of task graph and SRDF graph from Example 4.1.



(a) Data-driven schedule of task graph with constant execution times and buffer capacity of two containers.

(b) Strictly periodic schedule of SRDF graph with two tokens.

Figure 4.6: Schedules of task graph and SRDF graph from Example 4.2.

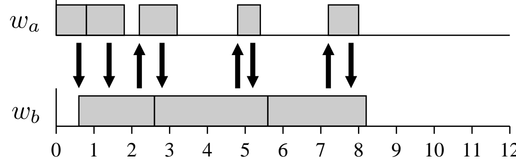


Figure 4.7: Data-driven schedule of task graph with varying execution times and buffer capacity of two containers from Example 4.3.

With $d = 2$, the maximum cycle mean of G is $\mu(G) = 3$. This implies that a strictly periodic schedule of actor firings exists, in which for every actor the difference between subsequent starts equals three time units. This schedule is shown for this SRDF graph in Figure 4.6(b). By comparing the token production times in Figure 4.6(b) with the corresponding container production times in Figure 4.6(a), we see that the schedule of G has token production times that are conservative to the corresponding container production times in the schedule of task graph T. From the maximum cycle mean analysis, we learn that the cycle $\langle e_{bb} \rangle$ is the critical cycle of this graph. This can be understood from the shown schedules, where the fact that actor v_b can only start after it has finished limits the number of firings per time interval of the SRDF graph. Therefore, a further increase in the number of tokens on the cycle $\langle e_{ab}, e_{ba} \rangle$ will not lead to a smaller maximum cycle mean and also not to a larger number of firings per time interval. This implies that, in this example, the worst-case number of task executions per time interval, i.e. the worst-case throughput, will not increase for any larger buffer capacity than the buffer capacity of two.

Example 4.3 Given the task graph of Example 4.2, let us in this example have that the tasks no longer have a constant execution time but instead a varying execution time, with $\hat{x}(w_a) = 1$ and $\hat{x}(w_b) = 3$ providing the upper bounds on the execution time of task w_a and task w_b, respectively. This can lead to the data-driven schedule as shown in Figure 4.7. When comparing the token production times in Figure 4.6(b) with the corresponding container production times in Figure 4.7, we see that, also in this example with varying task execution times, the schedule of G has token production times that are conservative to the corresponding container production times in the schedule of task graph T.

4.5.2 Modelling Sequences of Task Executions

In order to have a compact dataflow model that has a one-to-one relation between tasks and dataflow actors, we need to be able to model multiple non-blocking code-segments with a single actor firing. Any extension of the

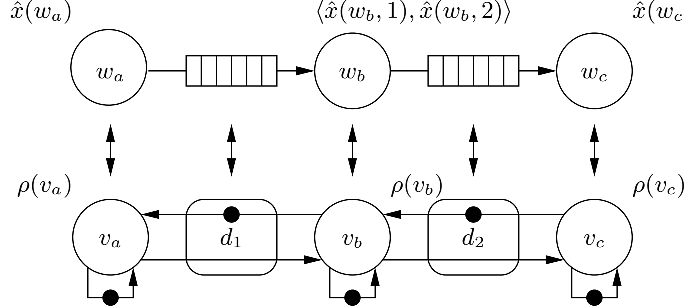
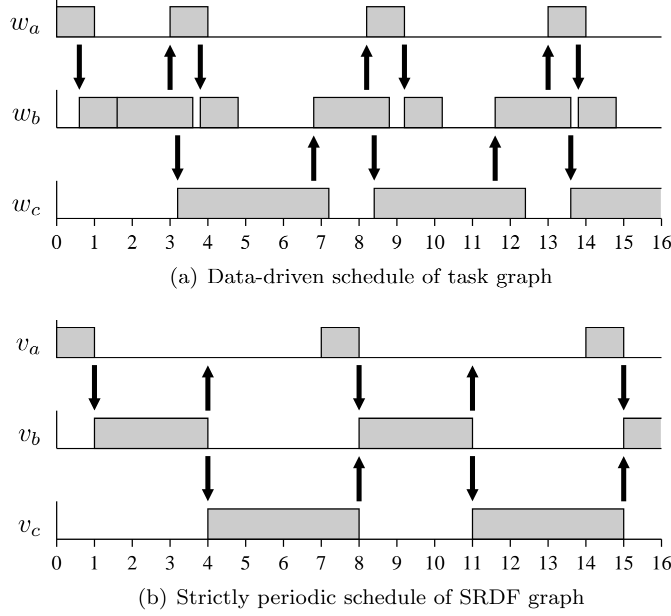


Figure 4.8: Example task graph, in which task w_b has two non-blocking code-segment that are modelled by a single actor firing of actor v_b.

task graph beyond the producer-consumer examples of the previous section includes a task that has multiple non-blocking code-segments. An example is given in Figure 4.8. In this task graph, task w_b reads data from the buffer from task w_a and writes data in the buffer to task w_c. Since the read and a write primitive both include one blocking acquisition primitive, and every blocking acquisition primitive starts a non-blocking code-segment, task w_b has two non-blocking code-segments. The execution time of execution i of task w_b is given $x(w_b, i)$. We let $\hat{x}(w_b, 1)$ denote the upper bound on the execution times of executions of the first non-blocking code-segment of w_b. Similarly, we let $\hat{x}(w_b, 2)$ denote the upper bound on the execution times of executions of the second non-blocking code-segment of w_b

Modelling multiple non-blocking code-segments by a single actor firing requires a reasoning that for instance explains why multiple blocking acquisition primitives can be modelled by a single firing rule. This reasoning is not made explicit in (Sriram and Bhattacharyya 2000). In (Moonen et al. 2007), this reasoning is provided for the so-called Cyclo-Static Dataflow (CSDF) model (Bilsen et al. 1996), which is a generalisation of SRDF. In this section, we first discuss a task graph that has a task with multiple non-blocking code-segments. This task graph will later be modelled by an SRDF graph. Subsequent to this example, a generalisation of the required relation between dataflow graph and task graph from Section 4.2 is provided that discusses how multiple non-blocking code-segments can be conservatively modelled.

Example 4.4 Given the task graph of Example 4.1, we extend this task graph with a second buffer b_{bc} that has one initially empty container and a third task w_c that has a constant execution time $\hat{x}(w_c) = 4$. In every task execution, task w_c acquires a full container from buffer b_{bc} and releases an empty container on buffer b_{bc}. Compared to the task graph of Figure 4.4, task w_b now has two non-blocking code-segments. This is because every ac-



(a) Data-driven schedule of task graph

(b) Strictly periodic schedule of SRDF graph

Figure 4.9: Schedules of task graph and SRDF graph from Example 4.4.

quisition primitive starts a non-blocking code-segment and task w_b has two acquisition primitives, an acquisition primitive that acquires one container from buffer b_{ab} and an acquisition primitive that acquires one container from b_{bc}. Furthermore, the non-blocking code-segment that acquires from b_{bc} includes two release primitives, a release primitive that releases one empty container on buffer b_{ab} and a release primitive that releases one full container on buffer b_{bc}. Task w_b executes the non-blocking code-segment that acquires from buffer b_{ab} and the non-blocking code-segment that acquires from buffer b_{bc} in an alternating fashion starting with the acquisition from b_{ab}. The first non-blocking code-segment that acquires from buffer b_{ab} has a constant execution time of $\hat{x}(w_b, 1) = 1$ and the second non-blocking code-segment that acquires from buffer b_{bc} has a constant execution time of $\hat{x}(w_b, 2) = 2$. This task graph is shown in Figure 4.8. A data-driven schedule of this task graph is shown in Figure 4.9(a).

To describe the required relation between task graph and dataflow graph, we need to extend Property 4.2 from a one-to-one relation between task executions and actor firings to a many-to-one relation between task executions and actor firings.

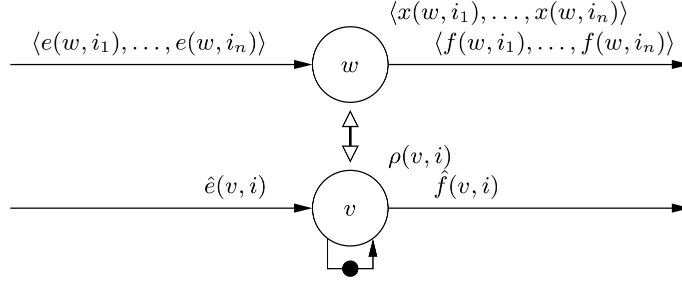


Figure 4.10: Relation between task graph and dataflow graph that satisfies Property 4.3.

Property 4.3 *For each task w in the task graph there is a unique actor v in the dataflow graph. For the relation between the input and output buffers of task w and the input and output queues of actor v, Property 4.1 holds. Furthermore, for a non-empty sequence of task executions $i = \langle i_1, \ldots, i_n \rangle$ there is a unique firing rule in the set of firing rules of actor v that is only satisfied if (1) the tokens that correspond with the containers consumed by i are present, and (2) the token that signals the finish of the previous actor firing is present. Furthermore, satisfaction of the firing rule that corresponds with the sequence of task executions i enables an actor firing that computes the same function as the composition of the functions computed by task executions $i_1, \ldots, i_n$ except that tokens are produced in an atomic action on queues instead of containers being produced on buffers.*

The generalisation of Theorem 4.3 to describe the required relation between external enabling times and between finish times is relatively straightforward and is based on the fact that task executions consume and produce containers in between start and finish, while actor firings consume and produce tokens at their start and finish. This leads to conservative consumption and production times of the actor firing relative to the task executions, which, by monotonicity, leads to a conservative dataflow model. The relation between task graph and dataflow graph that is considered in Theorem 4.4 is illustrated in Figure 4.10.

Theorem 4.4 *Given that Property 4.3 holds for task graph T and dataflow graph G. If Equation (4.7) holds for any task execution i_p that is part of the sequence of task executions i of a task w, then G is temporally conservative to T.*

$$e(w, i_p) \leq \hat{e}(v, i) \Rightarrow f(w, i_p) \leq \hat{f}(v, i) \tag{4.7}$$

Actor v corresponds to task w and actor firing i corresponds to the sequence of task executions i.

Proof. By Definition 4.14, dataflow graph G is temporally conservative to task graph T, if given a starting situation in which all token arrival times are conservative no actor firing can lead to token arrival times that are not conservative. In G, actor firings consume and produce the same amount of tokens as the cumulative number of containers consumed and produced by their corresponding sequence of task executions. Furthermore, task executions consume containers not before their start and produce containers not after their finish, while actor firings consume tokens at their start and produce tokens at their finish. This implies that if Equation (4.7) holds, then no actor firing produces its tokens earlier than the corresponding task execution produces its containers. This implies that given token arrival times that are conservative every actor firing leads to token arrival times that are again conservative, which implies that dataflow graph G is temporally conservative to task graph T. □

Example 4.4 (continued) Figure 4.8 also shows an SRDF graph for which, in relation to task graph T, Property 4.3 holds. This SRDF graph is temporally conservative to this task graph, given $\rho(v_a) = \hat{x}(w_a) = 1$, $\rho(v_b) = \hat{x}(w_b, 1) + \hat{x}(w_b, 2) = 1 + 2 = 3$, and $\rho(v_c) = \hat{x}(w_c) = 4$. Furthermore, the buffers have a capacity of one container, which is modelled by having $d_1 = 1$ and $d_2 = 1$. The maximum cycle mean of this SRDF graph is given by Equation (4.8).

$$\mu(G) = \max\left(\left\{\frac{1}{1}, \frac{3}{1}, \frac{4}{1}, \frac{4}{1}, \frac{7}{1}\right\}\right) = 7 \quad (4.8)$$

The strictly periodic schedule of this SRDF graph with its period equal to seven time units is shown in Figure 4.9(b). In Figure 4.9, we see that token production times, as shown in Figure 4.9(b), are conservative container production times, as shown in Figure 4.9(a).

4.6 Conclusion

In this chapter, we have discussed how guarantees on the satisfaction of timing constraints of a task graph can be obtained through dataflow simulation and dataflow analysis. We introduced functionally deterministic dataflow and showed that dataflow simulation is applicable for any functionally deterministic dataflow graph to obtain guarantees on the satisfaction of timing constraints for a single input stream of the task graph. Furthermore, we introduced Single-Rate Dataflow (SRDF) and showed that through dataflow analysis guarantees on the satisfaction of timing constraints can be obtained for any input stream of the task graph.

In this chapter, we modelled task graphs that execute on their private resources. Chapter 5 extends the dataflow simulation and analysis approaches to let them be applicable to model task graphs that execute on resources that are scheduled at run-time by budget schedulers.

Furthermore, the models presented in this chapter for which guarantees can be provided for all input streams do not allow to model task graphs with inter-task synchronisation behaviour that depends on the processed data stream. Functionally deterministic dataflow does allow to model these task graphs. However, only simulation is a known applicable technique for functionally deterministic dataflow implying that no guarantees for all input streams can be obtained. In Chapter 6, we present a new dataflow model that is more expressive than the models presented in this chapter for which analysis is applicable, but less expressive than functionally deterministic dataflow, for which we could only apply simulation.

Lastly, both the simulation and the analysis approach presented in this chapter provide guarantees on the satisfaction of the timing constraints for given buffer capacities. The presented dataflow analysis approach of maximum cycle mean analysis has exponential complexity in terms of multi-rate or cyclo-static dataflow graphs. In Chapter 6, we present an approach that computes buffer capacities that satisfy given timing constraints, where this approach has polynomial complexity for multi-rate and cyclo-static dataflow graphs.

Chapter 5

Modelling Run-Time Scheduling in Dataflow Graphs

Abstract – *This chapter shows that the effects of run-time scheduling can be included in the dataflow model of a functionally deterministic task graph, given that the run-time scheduler guarantees resource budgets to each task. Initially, we will model these effects by including response times in our dataflow model. This can, however, lead to inaccurate analysis results. We will show that a model with two parameters, latency and rate, results in accurate bounds.*

In this chapter, we extend our results from Chapter 4 to task graphs that are executed on resources that are scheduled at run-time by budget schedulers. More specifically, we show that the temporal behaviour of any functionally deterministic task graph that executes on resources that are scheduled by a budget scheduler can be conservatively modelled by a dataflow graph. We will first present a conservative dataflow model that uses response times to capture the worst case temporal effects imposed by run-time scheduling on a single task execution. Response times depend on the execution time and scheduler settings. However, as we will show with an intuitive example, modelling the temporal behaviour of run-time scheduled tasks with response times can lead to inaccurate bounds on the temporal behaviour of the task graph. We will show that a model that additionally takes the arrival time of containers into account results in more accurate

The material in this chapter is based on (Wiggers et al. 2007b).

bounds. This more accurate model has a latency and a rate parameter instead of the single response time parameter. In order to apply this more accurate model with a latency and rate parameter, we will need to generalise the required one-to-one correspondence between dataflow model and task graph.

The outline of this chapter is as follows. In Section 5.1, we will model the effects of run-time scheduling by budget schedulers with so-called response times, which only depend on the execution time and scheduler settings. We will show that the model is conservative, but we also provide an example in which this model is not accurate. Section 5.2 will present a more accurate model that additionally depends on the arrival times of containers. This more accurate model has a latency and a rate parameter. In order to include this in dataflow graphs that enable guarantees for all input streams, we generalise the required relation between dataflow model and task graph in Section 5.3. In Section 5.4, we evaluate the accuracy of dataflow simulation and analysis in case the effects of run-time scheduling are included with the latency and rate model. Finally, we conclude this chapter by a discussion of the presented concepts in Section 5.5.

5.1 Modelling Run-Time Scheduling with Response Times

In this section, we conservatively include the effects of run-time scheduling by budget schedulers in the dataflow model. We first derive an expression for the upper bound on the response time of a task execution that is valid for any budget scheduler. Subsequently, we show how the effects of run-time scheduling by budget schedulers are conservatively taken into account in the dataflow model. The conservativeness of the dataflow model rests on properties specific to budget schedulers. Furthermore, we will provide an example illustrating that the application of a model with response times can lead to inaccurate upper bounds on the temporal behaviour of the task graph. The next section will provide a model that is more accurate than response times.

5.1.1 Conservative Bound on Response Time

Based on the property that budget schedulers guarantee a budget R in any time interval of length Q, we derive an expression for a tight upper bound on the response time of task executions that are scheduled by a budget scheduler. This bound requires that budget R and interval Q, and the execution time are known. If, instead of the execution time, an (estimated) upper bound on the execution time is known, then this upper bound on the execution time can be used to compute a response time that is conservative

for the execution times that are smaller than or equal to this (estimated) upper bound.

We focus, in this section, on a task w, which is further omitted from the discussion for reasons of clarity.

Lemma 5.1 shows that an upper bound on the response time of task execution i is provided by Equation (5.1).

Lemma 5.1 *For every scheduler that guarantees every task a minimum amount of time R in every interval of time Q, an upper bound on the response time $r(w, i)$ of execution i of task w is given by*

$$r(w, i) \leq x(w, i) + (Q - R) \left\lceil \frac{x(w, i)}{R} \right\rceil \tag{5.1}$$

Proof. The worst-case response time of execution i occurs if previous executions have already depleted the allocated time budget. In this case, execution i needs to wait for maximally $Q - R$ time before it can start its execution. This results in a worst-case response time of as defined in Equation (5.1). □

Theorem 5.1 shows that taking the upper bound on the response time of task execution i as the firing duration of the corresponding actor firing i results in a temporally conservative dataflow model.

Theorem 5.1 *Given for any execution i of any task w in task graph T an execution time of $x(w, i)$, where task w has a guaranteed budget R in every time interval Q. Further given that Property 4.2 holds for dataflow graph G and task graph T. If the actor firing corresponding to task execution i has a firing duration that is equal to the upper bound on the response time of task execution i as given by Equation* (5.1)*, then dataflow graph G is temporally conservative to task graph T.*

Proof. We only deal with task w and actor v in this proof. To improve legibility both are omitted from the parameter list of functions referred to in this proof.

From the definition of the response time of task execution i as the time interval from the enabling time until the finish time of task execution i it follows that Equation (5.2) holds.

$$f(i) \leq \max(e(i), f(i - 1)) + x(i) + (Q - R) \left\lceil \frac{x(i)}{R} \right\rceil \tag{5.2}$$

The enabling time of the actor firing that corresponds to task execution i is given by $\max(\hat{e}(i), \hat{f}(i - 1))$. Therefore, associating the upper bound on the response time of execution i as given by Equation (5.1) results in a finish time of this actor firing that is given by Equation (5.3).

$$\hat{f}(i) = \max(\hat{e}(i), \hat{f}(i-1)) + x(i) + (Q - R)\left\lceil \frac{x(i)}{R} \right\rceil \tag{5.3}$$

We will show that $e(i) \leq \hat{e}(i) \Rightarrow f(i) \leq \hat{f}(i)$ holds by induction over executions i.

Base step With $f(-1) = 0$, we have that the finish time of execution 0 is given by Equation (5.4).

$$f(0) = e(0) + x(0) + (Q - R)\left\lceil \frac{x(0)}{R} \right\rceil \tag{5.4}$$

With $\hat{f}(-1) = 0$, we have for execution 0 that the finish time of the corresponding actor firing is given by Equation (5.4).

$$\hat{f}(0) = \hat{e}(0) + x(0) + (Q - R)\left\lceil \frac{x(0)}{R} \right\rceil \tag{5.5}$$

Therefore, we have that $e(0) \leq \hat{e}(0)$ implies that $f(0) \leq \hat{f}(0)$.

Induction step Given that $e(i-1) \leq \hat{e}(i-1) \Rightarrow f(i-1) \leq \hat{f}(i-1)$, we need to show that $e(i) \leq \hat{e}(i) \Rightarrow f(i) \leq \hat{f}(i)$, where $f(i)$ is given by Equation (5.2) and $\hat{f}(i)$ is given by Equation (5.3). Given that $e(i-1) \leq \hat{e}(i-1) \Rightarrow f(i-1) \leq \hat{f}(i-1)$, we have that Equation (5.6) holds.

$$f(i) \leq \max(e(i), \hat{f}(i-1)) + x(i) + (Q - R)\left\lceil \frac{x(i)}{R} \right\rceil \tag{5.6}$$

Given that $e(i) \leq e(i-1)$ holds, we have that Equation (5.7) holds.

$$f(i) \leq \max(\hat{e}(i), \hat{f}(i-1)) + x(i) + (Q - R)\left\lceil \frac{x(i)}{R} \right\rceil \tag{5.7}$$

Therefore, given that $e(i-1) \leq \hat{e}(i-1) \Rightarrow f(i-1) \leq \hat{f}(i-1)$, it holds that $e(i) \leq \hat{e}(i) \Rightarrow f(i) \leq \hat{f}(i)$.

This implies by Theorem 4.3 that dataflow graph G is temporally conservative to task graph T. □

If the finish time of firing i is computed with Equation (5.3), then it is clear that an earlier external enabling time of firing i does not lead to a later finish time of firing i. Since the bounds do not affect the behaviour of any other actor firing, we have that the resulting dataflow model has monotonic temporal behaviour. This result is quite peculiar, since it is well known that schedulers can have non-monotonic behaviour, i.e. scheduling anomalies (Graham 1969). Also a budget scheduler can have non-monotonic behaviour, i.e. an earlier enabling of one task can lead to a later start of another task. However, in our model we have taken into account the latest time at which every task obtains its budget. In this way, we have bounded

the non-monotonic effects of the scheduler. Furthermore, the scheduling of the dataflow graph does not include a bounded number of processing resources, which implies that it is not a multiprocessor scheduling problem.

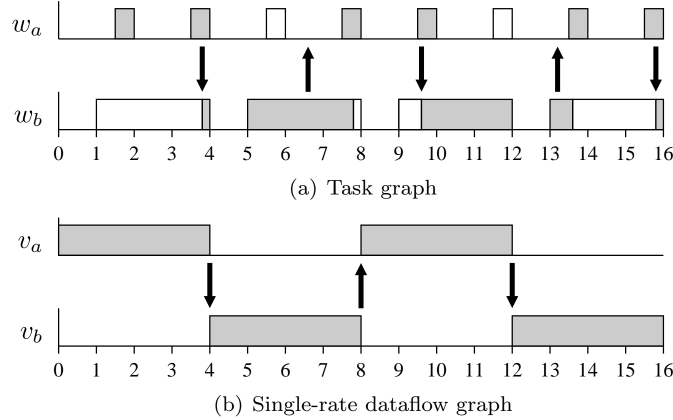
For the case without resource sharing, conservatism was introduced by letting consumptions be at the start and productions be at the finish. In case of resource sharing, we have that the arrival times are conservative for all initial states of the budget scheduler, e.g. independent of the current position in the time-division multiplex period when we start this application. This is because in case of a multiprocessor system in which the processors each have their individual clock, which are not synchronised with each other, then inherent – unknown – variation in the clocks leads to inherent – unknown – variation in the alignment of the time-division multiplex schedules, i.e. the alignment of the time-division multiplex schedules varies over time. In case the applied budget scheduler is a time-division multiplex scheduler, then with our dataflow model, we compute the worst-case arrival times for every possible alignment of the time-division multiplex schedules.

5.1.2 Dataflow Analysis with Response Times

In this section, we illustrate how the effects of run-time scheduling on the temporal behaviour of a task graph are included in a single-rate dataflow graph by using response times. We will first provide examples of a producer-consumer task graph for which the result of Theorem 5.1 that is based on a one-to-one relation between task executions and actor firings is sufficient. Subsequently, we will extend the result of Theorem 5.1 to hold for a one-to-one relation between sequences of task executions and single actor firings, which will allow us to model task graphs with a general topology. This extension rests on the property of budget schedulers that in every time interval Q at least R budget is available.

The first two examples present a task graph that has allocated budgets that are such that the model with response times provides an accurate upper bound on the container arrival times in these task graphs. The first example has a task graph for which we have a model with a one-to-one correspondence between non-blocking code-segments and actor firings. The second example has a task graph that includes a task with two non-blocking code-segments. In order to model this task graph, we will present a generalisation of Theorem 5.1 that is valid when multiple task executions are modelled by a single actor firing.

The third example presents a task graph that has allocated budgets that are such that the model with response times provides upper bounds on container arrival times that with an increasing number of task executions become increasingly less accurate. This is because the model with response times cannot include the fact that multiple task executions can execute immediately consecutive to each other within the allocated budget. We will extend our theory to allow for a more accurate model in later sections.



(a) Task graph

(b) Single-rate dataflow graph

Figure 5.1: data-driven schedule of task graph with constant execution times and buffer capacity of one container executed on TDM scheduled processors.

The first example shows a task graph with resource budget allocations that has a dataflow model with a one-to-one correspondence between task executions and actor firings. Furthermore, the upper bounds on container production times as provided by the token production times of this dataflow model are accurate.

Example 5.1 Given the task graph of Example 4.1, in this example we execute this task graph on two Time-Division Multiplex (TDM) scheduled processors. Task w_a has a budget of $R = 1/2$ in time interval $Q = 2$. Task w_b has a budget of $R = 3$ in time interval $Q = 4$. An example data-driven execution of this task graph is shown in Figure 5.1(a). In this figure, the rectangles that are not filled denote budget in which the task is polling for the arrival of sufficient containers, but not executing the task code.

According to Theorem 5.1, the SRDF graph from Figure 4.4, where actor v_a has a firing duration $\rho(v_a) = 4$ and actor v_b has a firing duration $\rho(v_b) = 4$ is temporally conservative to the task graph in this example. With these firing durations, the SRDF graph has a maximum cycle mean of four time units. A strictly periodic schedule with a period of four is shown in Figure 5.1(b). This dataflow model provides an accurate upper bound on container arrival times, because, in the shown data-driven schedule of the task graph, the TDM schedulers are started in their worst-case position. Furthermore, the task execution times are constant in this example. The model becomes more accurate if container production occurs later in the task execution.

In the following example, we have a task graph with a task that reads from one buffer and writes on another buffer. This means that there are two blocking acquisition primitives, and therefore two non-blocking code-segments in this task. In order to model a task with two non-blocking code-segments by a dataflow actor that has a single firing rule, firing duration and production rule, we need to model a sequence of two task executions with a single actor firing. The required correspondence between task graph and dataflow graph was discussed in Section 4.5.2. Subsequent to discussing the example task graph, we will extend Theorem 5.1 and present a sufficient condition on the relation between actor firings and sequences of task executions such that the dataflow graph is temporally conservative to the task graph.

Example 5.2 Given the task graph from Example 4.4 as shown in Figure 4.8, in this example we schedule this task graph on three TDM scheduled processors, where task w_a has budget $R = 1/2$ in time interval $Q = 2$, task w_b has budget $R = 3$ in time interval $Q = 4$, and task w_c has budget $R = 4$ in time interval $Q = 5$. A data-driven schedule of this task graph is shown in Figure 5.2(a).

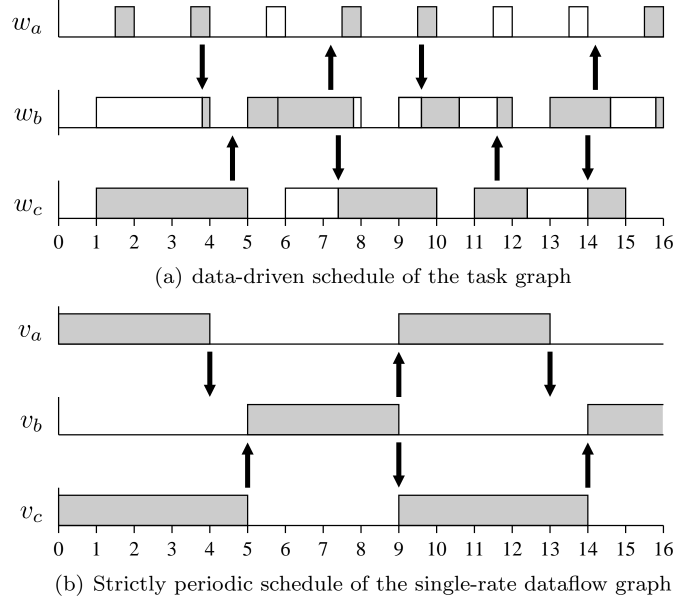
In order to present a dataflow graph of which we can show that it is temporally conservative to the task graph from Example 5.2, we need the following theorem that provides a sufficient condition on the relation between actor firings and sequences of task executions.

Theorem 5.2 *Given that Property 4.3 holds for dataflow graph G and task graph T. If actor firing i that corresponds to the sequence of task executions $i = \langle i_1, \ldots, i_n \rangle$ has a firing duration that is equal to $r(w, i)$ as given by Equation (5.1), with execution time equal to $\sum_{m=1}^{n} x(w, i_m)$, then dataflow graph G is temporally conservative to task graph T.*

Proof. We only deal with task w and actor v in this proof. To improve legibility both are omitted from the parameter list of functions referred to in this proof.

Let the sequence of task executions $i = \langle i_1, \ldots, i_n \rangle$ have external enabling times $e(i_1), \ldots, e(i_n)$. The external enabling time of actor firing i is the earliest time at which all corresponding required tokens are present, which is $\hat{e}(i) = \max(\{\hat{e}(i_1), \ldots, \hat{e}(i_n)\})$.

If we delay the external enabling time of any task execution i_p, with $1 \leq p \leq n$, that is part of the sequence of task executions i to become $e'(i_p) = \max(\{e(i_1), \ldots, e(i_n)\})$, then all task executions in the sequence i are externally enabled at time $\max(\{e(i_1), \ldots, e(i_n)\})$. This implies that with these delayed external enabling times of the task executions in i, we



(a) data-driven schedule of the task graph

(b) Strictly periodic schedule of the single-rate dataflow graph

Figure 5.2: Task graph from Figure 4.8 with constant execution times and buffer capacities of one container executed on TDM scheduled processors together with a conservative single-rate dataflow graph.

have that the sequence of task executions i can be seen as one task execution, since all acquisition primitives are guaranteed to succeed immediately. We say that this non-blocking code-segment has an external enabling time of $e'(i) = \max(\{e(i_1), \ldots, e(i_n)\})$ and a finish time $f'(i) = f'(i_n)$, where $f'(i_n)$ is the finish time of task execution i_n given external enabling time $e'(i_n)$. Furthermore, task execution i has an execution time equal to $\sum_{m=1}^{n} x(i_m)$. From Theorem 5.1, it follows that dataflow graph G is temporally conservative to the task graph T in which the external enabling times are delayed as just specified. This means that we have $e'(i) \leq \hat{e}(i) \Rightarrow f'(i) \leq \hat{f}(i)$.

We will now show that G is still temporally conservative to T if the external enabling is not delayed. Given the delayed external enabling time of the task executions i_p, an upper bound on the finish time of task execution i_p is given by Equation (5.8).

$$f'(i_p) \leq \max(e'(i), f(i-1)) + \sum_{m=1}^{p} x(i_m) + (Q - R) \left\lceil \frac{\sum_{m=1}^{p} x(i_m)}{R} \right\rceil \quad (5.8)$$

A budget scheduler provides a guaranteed minimum budget in a time interval. Therefore, the guaranteed minimum budget available to the task in the interval from $\max(e'(i), f(i-1))$ until $f'(i_p)$ does not change if task executions already use budget before $e'(i)$. Therefore, $f'(i_p)$ is a valid upper bound for any external enabling time of i_p that is earlier than or equal to $e'(i)$, i.e. $e(i_p) \leq e'(i) \Rightarrow f(i_p) \leq f'(i_p) \leq f'(i_n) = f'(i)$. Because $e'(i) \leq \hat{e}(i) \Rightarrow f'(i) \leq \hat{f}(i)$, we have that $e(i_p) \leq \hat{e}(i) \Rightarrow f(i_p) \leq \hat{f}(i)$. By Theorem 4.4, this implies that dataflow graph G is temporally conservative to task graph T. □

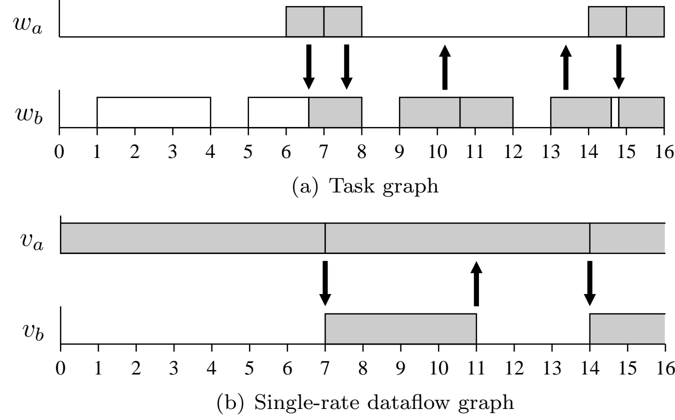
Having established Theorem 5.2, we can continue with Example 5.2 and present a dataflow graph that is temporally conservative to the task graph from this example.

Example 5.2 (continued) According to Theorem 5.2, the SRDF graph from Figure 4.8 is temporally conservative to the task graph from this example, with firing durations $\rho(v_a) = 4$, $\rho(v_b) = 4$, and $\rho(v_c) = 5$. This means that the maximum cycle mean of this graph is nine time units. A strictly periodic schedule with a period of nine is shown in Figure 5.2(b). As illustrated in Figure 5.2, the token production times, from Figure 5.2(b), are conservative container production times, as illustrated in Figure 5.2(a).

The bound on the response time of a task execution on a budget scheduler takes into account that the worst-case enabling time occurs just before the budget is depleted. However, depending on budget allocation and execution times it can very well be that the assumption of an enabling at the worst-case instant for every task execution results in an overly pessimistic bound on the temporal behaviour. The following example shows such a situation in which application of the bound on the response time results in a dataflow graph that has temporal behaviour that is diverging from the temporal behaviour of the task graph, resulting in a bound that is less accurate with an increasing number of task executions. This example illustrates the problem addressed in the next sections.

Example 5.3 In this example, we have the same task graph as in Example 5.1. The budget of task w_b is also the same with budget $R = 3$ in time interval $Q = 4$. The budget of task w_a is changed to a budget of $R = 2$ in time interval $Q = 8$. A data-driven schedule of this task graph is shown in Figure 5.3(a).

According to Theorem 5.2, the SRDF graph from Figure 4.4, where actor v_a has a firing time $\rho(v_a) = 7$ and actor v_b has a firing duration $\rho(v_b) = 4$ is temporally conservative to the task graph in this example. A strictly periodic schedule of this SRDF graph is shown in Figure 5.3(b). In Figure 5.3, we see that the first token production time in Figure 5.3(b) accurately bounds the container production time of the first task execution,



(a) Task graph

(b) Single-rate dataflow graph

Figure 5.3: data-driven schedule of task graph with constant execution times and buffer capacity of two containers executed on TDM scheduled processors.

as shown in Figure 5.3(a). However, the fact that the second task execution can execute immediately after the first task execution has finished is not captured in this dataflow model, which assumes that every execution needs to wait $Q - R$ time before it can start. Even though the presented dataflow model is temporally conservative to the task graph, with an increasing number of executions the upper bound on container production times becomes increasingly less accurate.

In the following sections, we will show that the effects of run-time scheduling on the temporal behaviour of a task can be more accurately modelled by two dataflow actors. To achieve this, we first relax our requirements on the relation between task graph and dataflow graph as specified in Section 4.2. Subsequently, we show that a model with two parameters, i.e. a latency and a rate parameter, is more accurate than the model from Section 5.1 that only has a single parameter, i.e. the response time.

5.2 Modelling Run-Time Scheduling with Latency and Rate

In this section, we derive a more accurate upper bound on the finish times of task executions given that the task is scheduled on a budget schedulers. This bound requires that budget R and interval Q are known, and,

furthermore, a conservative external enabling time of this execution, a conservative finish time of the previous execution, and the execution time are known. If, instead of the execution time, an (estimated) upper bound on the execution time is known, then this upper bound on the execution time can be used to compute a finish time that is conservative for the execution times that are smaller than or equal to this (estimated) upper bound. The presented bound on finish times obtains a tight bound over multiple task executions by taking into account the external enabling time of this task execution and the finish time of the previous task execution.

In this section, we focus on a task w, which is further omitted from the discussion for reasons of clarity. Let $e(i)$ be the external enabling time of task execution i, i.e. the time at which task execution i is enabled by sufficient containers on all adjacent buffers, let $f(i)$ be the finish time of task execution i, while $x(i)$ is the execution time of task execution i. Let $i-1$ denote the previous task execution. We will show that Equation (5.12) specifies an upper bound on the finish time that holds for all schedulers in the class of budget schedulers. A tighter upper bound can be found for specific schedulers. First we introduce the concept of consecutive executions, which is used in Lemma 5.2 to derive an upper bound on a sequence of consecutive executions. This upper bound depends on knowledge of the actual external enabling and finish times, which are unknown in the model. Lemma 5.3 linearises the bound of Lemma 5.2 to obtain a bound that depends on the upper bound on finish times. Lemma 5.4 provides an upper bound on finish times that is valid given known upper bounds on external enabling times. Theorem 5.3 shows that with the bound of Lemma 5.4 a conservative dataflow model is obtained.

Definition 5.1 (Consecutive execution) *Execution i of task w is part of a consecutive execution that starts with execution k of task w if and only if for all executions j of task w, with $k < j \leq i$, we have that $e(j) \leq f(j-1)$.*

Lemma 5.2 provides an upper bound on the finish time of a sequence of consecutive executions.

Lemma 5.2 *Given that execution i is part of a consecutive execution that starts with execution k. Then for every scheduler that guarantees a task a minimum amount of time R in every interval of time Q, an upper bound on the finish time of execution i is given by*

$$f(i) \leq f^w(i) = e(k) + \sum_{j=k}^{i} x(j) + (Q-R)\left\lceil \frac{\sum_{j=k}^{i} x(j)}{R} \right\rceil \tag{5.9}$$

Proof. If $e(k) > f(k-1)$, then at time $e(k)$ execution k has sufficient containers present on all adjacent buffers and execution $k-1$ has finished,

which implies that from $e(k)$ execution k can start its execution. The worst-case finish time of execution k occurs if previous executions have already depleted the allocated time budget. In this case, execution k needs to wait for maximally $Q - R$ time before it can start its execution. This results in a worst-case finish time of $g(k) = e(k) + x(k) + (Q - R)\lceil x(k)/R \rceil$. If for each execution j, with $k < j \leq i$, we have that $e(j) \leq f(j-1)$, then we can see executions k through i as a single execution. In this case, we have that an upper bound on the finish time of execution i is given by $f^w(i)$ as defined in Equation (5.9). □

Bound f^w is a tight bound. This is because bound f^w equals the actual finish time f if $e(k)$ occurs just at the moment that the budget R is depleted, where k is the first execution of a consecutive execution.

Bound f^w can be applied if the starts of consecutive executions can be determined. However, the condition that determines when a consecutive execution starts depends on the actual enabling and finish times, while bound f^w computes upper bounds on finish times and container arrival times. Determining the starts of consecutive executions based on the application of bound f^w is therefore problematic. The following theorem presents a bound that is an upper bound on f^w that does not depend on the knowledge of consecutive executions.

Lemma 5.3 *For every scheduler that guarantees a task a minimum amount of time R in every interval of time Q, we have that an upper bound on the finish time of execution i is given by*

$$f(i) \leq f^w(i) \leq \max(e(i) + Q - R, f^w(i-1)) + \frac{Q \cdot x(i)}{R} \tag{5.10}$$

Proof. We conservatively bound Equation (5.9).

$$f^w(j) \leq f^l(j) = e(k) + Q - R + \frac{Q \sum_{i=k}^{j} x(i)}{R} \tag{5.11}$$

For any execution i, we can have two cases. Either $e(i) > f(i-1)$ and i is the first execution in a sequence of consecutive executions, or $e(i) \leq f(i-1)$. In case $e(i) > f(i-1)$, then $f^l(i) = e(i) + Q - R + Q \cdot x(i)/R$. In case $e(i) \leq f(i-1)$, then we have that $f^l(i) - f^l(i-1) = Q \cdot x(i)/R$. Therefore for any i we have that Equation (5.10) holds. □

The upper bound on finish times as given by Equation (5.10) is still a tight bound, although compared to bound f^w it now additionally depends on the execution times. This is because $\lceil x \rceil$ was bounded by $x + 1$. Depending on the value of x this is a tight bound.

The bound on $f(i)$ as given by Equation (5.10) still depends on the actual external enabling times. Lemma 5.4 provides an upper bound on the finish times that only depends on upper bound on external enabling and finish times.

Lemma 5.4 *Equation* (5.12) *holds.*

$$e(i) \leq \hat{e}(i) \Rightarrow f^w(i) \leq \hat{f}(i) = \max(\hat{e}(i)+Q-R, \hat{f}(i-1)) + \frac{Q \cdot x(i)}{R} \quad (5.12)$$

Proof. We will show by induction over i that if $e(i) \leq \hat{e}(i)$, then $\hat{f}(i)$ as given by Equation (5.12) is an upper bound on $f^w(i)$ as given by Equation (5.10), i.e. $e(i) \leq \hat{e}(i) \Rightarrow f^w(i) \leq \hat{f}(i)$.

Base step. For $i = 0$, we have that $f^w(0) = \max(e(0)+Q-R, f^w(-1)) + \frac{Q \cdot x(0)}{R}$. With $f^w(-1) = 0$ this results in $f^w(0) = e(0) + Q + R + \frac{Q \cdot x(0)}{R}$, which by $e(0) \leq \hat{e}(0)$ results in $f^w(0) \leq \hat{e}(0) + Q + R + \frac{Q \cdot x(0)}{R}$. Since $\hat{f}(-1) = 0$, we have that $\hat{f}(0) = \hat{e}(0) + Q + R + \frac{Q \cdot x(0)}{R}$, which implies that $e(0) \leq \hat{e}(0) \Rightarrow f^w(0) \leq \hat{f}(0)$.

Induction step. We show that $e(i-1) \leq \hat{e}(i-1) \Rightarrow f^w(i-1) \leq \hat{f}(i-1)$ implies that $e(i) \leq \hat{e}(i) \Rightarrow f^w(i) \leq \hat{f}(i)$.

We have that $f^w(i) = \max(e(i)+Q-R, f^w(i-1)) + \frac{Q \cdot x(i)}{R}$, which given $e(i) \leq \hat{e}(i)$ results in $f^w(i) \leq \max(\hat{e}(i) + Q - R, f^w(i-1)) + \frac{Q \cdot x(i)}{R}$. With $f^w(i-1) \leq \hat{f}(i-1)$ this implies that $f^w(i) \leq \max(\hat{e}(i)+Q-R, \hat{f}(i-1)) + \frac{Q \cdot x(i)}{R} = \hat{f}(i)$. □

Theorem 5.3 shows that the results of Lemma 5.4 can be applied to obtain a temporally conservative dataflow model.

Theorem 5.3 *If finish times in dataflow graph G are computed according to Equation* (5.12)*, then G is temporally conservative to task graph T.*

Proof. Since Lemma 5.3 states that $f(i) \leq f^w(i)$ and Lemma 5.4 states $e(i) \leq \hat{e}(i) \Rightarrow f^w(i) \leq \hat{f}(i)$, we have that $e(i) \leq \hat{e}(i) \Rightarrow f(i) \leq \hat{f}(i)$. By Theorem 4.3, this implies that dataflow graph G is temporally conservative to task graph T. □

The bound on finish times as given by Equation (5.12) can be applied in dataflow simulation, where bounds on external enabling and finish times are available, but not immediately in dataflow analysis. The next section will generalise our requirements on the relation between dataflow graph and task graph to allow a task to be modelled by two actors. The relation between the finish time and external enabling time of this dataflow component consisting of two actors is equal to the bound provided by Equation (5.12).

5.3 Dataflow Analysis with Latency and Rate Model

The result of Theorem 5.3 can be applied in a dataflow simulation to obtain guarantees on the satisfaction of timing constraints for a single input stream. In this section, we generalise the required relation between dataflow graph and task graph and show how given this generalised relation the effects of run-time scheduling can be included in dataflow analysis with a latency and rate model. The resulting analysis is more accurate than the dataflow analysis that used response times to capture the effects of run-time scheduling.

5.3.1 Task Modelled by Two Actors

In this section, we relax the requirements from Section 4.2 on the relation between task graph and dataflow graph. As in Section 4.2, this section defines a one-to-one relation between task executions and actor firings. The difference with the required relation as defined in Section 4.2 is that this section defines a one-to-one relation between task and dataflow components. A dataflow component is a subset of actors. With the required relation as defined in this section, every task is modelled by one dataflow component consisting of two actors.

We first define a dataflow component, which is an element in the partitioned set of actors of a dataflow graph.

Definition 5.2 (Dataflow graph component set) *The component set K of a dataflow graph G is a partitioning of the set of actors V of dataflow graph G. This means that*

$$\forall k_x, k_y \in K \bullet k_x \neq k_y \Rightarrow k_x \cap k_y = \emptyset \wedge V = \bigcup K \tag{5.13}$$

Definition 5.3 (Dataflow graph component) *A dataflow graph component is an element in a dataflow graph component set.*

The following property requires that every task is modelled by one dataflow component that consists of two actors v_1 and v_2, where the firing rule of v_1 corresponds with the enabling condition of the task, and the produced tokens by firings of v_2 correspond with the containers produced by executions of the task. The correspondence is illustrated in Figure 5.4.

Property 5.1 *For every task in the task graph, there is a unique dataflow component in the dataflow graph. For the relation between the input and output buffers of a task and the input and output queues of the corresponding dataflow component, Property 4.1 holds. Furthermore, the component consists of two actors, v_1 and v_2, and queues e_{12} and e_{22}, with $\mathbf{src}(e_{12}) = v_1$, $\mathbf{dst}(e_{12}) = v_2$, and $\delta(e_{12}) = 0$, and $\mathbf{src}(e_{22}) = v_2$, $\mathbf{dst}(e_{22}) = v_2$, and*

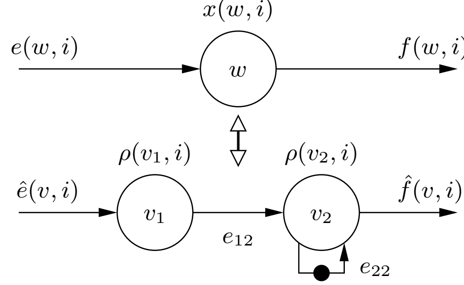


Figure 5.4: Required relation between task and dataflow component, given a one-to-one relation between task executions and component firings.

$\delta(e_{22}) = 1$. *Actor* v_1 *has a set of firing rules such that for any task execution* i *there is a unique firing rule in this set of firing rules that is satisfied iff the tokens that correspond with the containers consumed by task execution* i *are present. Furthermore,* v_1 *produces only on queue* e_{12} *on which it produces every firing a single token. Actor* v_2 *has a firing rule that is satisfied iff on both queue* e_{12} *and on queue* e_{22} *a token is present. Firing* i *of actor* v_2 *produces tokens on queues such that there is a one-to-one correspondence with the containers that are produced on buffers by execution* i *of the corresponding task.*

Property 5.1 allows us to define a component firing as a set of actor firings, which in turn allows us to have an intuitive definition of the external enabling time and finish time of a component firing.

Definition 5.4 (Component firing) *Given that Property 5.1 holds for task graph* T *and dataflow graph* G*. We define firing* i *of a component* k_x *of* G *as the two-element set of actor firings given by firing* i *of actor* v_1 *of* k_x *together with firing* i *of actor* v_2 *of* k_x*.*

Definition 5.5 (External enabling time component firing) *The external enabling time of firing* i *of component* k_x *is the external enabling time of firing* i *of actor* v_1 *of* k_x*.*

Definition 5.6 (Finish time component firing) *The finish time of firing* i *of component* k_x *is the finish time of firing* i *of actor* v_2 *of* k_x*.*

Given the just defined external enabling time and finish time of a component firing, Theorem 4.3 can be relatively straightforwardly generalised from a one-to-one relation between tasks and actors to a one-to-one relation between tasks and dataflow components.

Theorem 5.4 *Given that Property 5.1 holds for task graph T and dataflow graph G. If Equation* (5.14) *holds for any task execution i of a task w, then G is temporally conservative to T.*

$$e(w,i) \leq \hat{e}(k,i) \Rightarrow f(w,i) \leq \hat{f}(k,i) \tag{5.14}$$

Component k corresponds to task w and component firing i corresponds to task execution i.

Proof. By definition 4.14, dataflow graph G is temporally conservative to task graph T, if given a starting situation in which all token arrival times are conservative no firing can lead to token arrival times that are not conservative. In G, component firings consume and produce the same amount of tokens as the number of containers consumed and produced by their corresponding task execution. Furthermore, task executions consume containers not before their start and produce containers not after their finish, while component firings consume tokens at their start and produce tokens at their finish. This implies that if Equation (5.14) holds, then no component firing produces its tokens earlier than the corresponding task execution produces its containers. This implies that given token arrival times that are conservative every component firing leads to token arrival times that are again conservative, which implies that dataflow graph G is temporally conservative to task graph T. □

5.3.2 Applying Dataflow Analysis with Latency and Rate

In this section, we provide a sufficient condition on the firing times of the two actors in a dataflow component to let the dataflow graph be temporally conservative to the task graph. Subsequently, we apply this result on the example from Section 5.1 that showed that a model with response times can lead to inaccurate bounds, and show that a model with latency and rate results in a more accurate model.

Theorem 5.5 *Given that Property 5.1 holds for task graph T and dataflow graph G. Furthermore, given that task w corresponds with component k. If any firing i of component k has a firing duration $Q-R$ for actor v_1 and has a firing duration $\frac{Q \cdot x(w,i)}{R}$ for actor v_2, then G is temporally conservative to T.*

Proof. The finish time of firing i of actor v_2 of component k that models task w is $\max(a(i,e_{12}), a(i-1,e_{22})) + \frac{Q \cdot x(w,i)}{R}$, where $a(i,e_{12})$ is the arrival time of the i-th token on queue e_{12} and $a(i-1,e_{22})$ is the arrival time of the $i-1$-th token on queue e_{22}. We have that $a(i,e_{12})$ equals $\hat{e}(i)+Q-R$, and that $a(i-1,e_{22}) = \hat{f}(i-1)$. This implies that the finish time of component firing i is given by Equation (5.12). By Theorems 5.3 and 5.4 this implies that the dataflow graph G is temporally conservative to task graph T. □

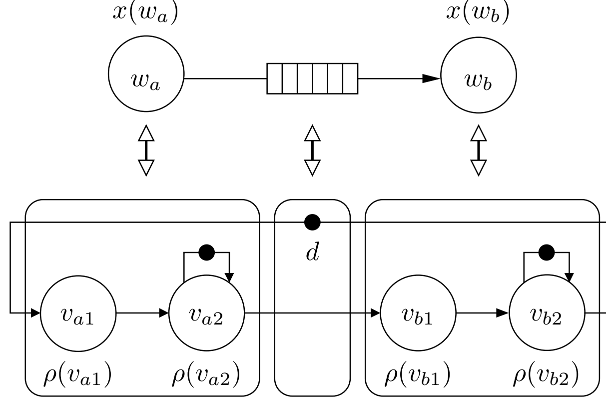


Figure 5.5: Task graph with corresponding SRDF graph.

Theorem 5.5 provides a sufficient condition on the SRDF graph such that tasks can be conservatively modelled with two actors. We revisit the task graph from Example 5.3 for which the SRDF graph with response times did not provide accurate upper bounds on container arrival times. The SRDF graph that we will now construct is shown in Figure 5.5 and has a latency and a rate parameter instead of only a response time. The rate parameter captures that the budget allows for a specific number of task executions in a given time interval. The latency parameter captures the discretisation effects, i.e. the fact that we do not have a fluid system in which budget is distributed completely homogeneously over time, but instead a discrete system in which budget is available in bursts. A budget of $R = 1/2$ every time interval of $Q = 2$, therefore, has the same rate as a budget of $R = 2$ every time interval of $Q = 8$. However, since in the latter case the burstiness with which budget can come available is increased, the latency is higher.

Example 5.4 Given the task graph with two tasks w_a and w_b from Example 5.3, as shown in Figure 5.5. Task w_a has a constant execution time of one time unit, and a budget $R = 2$ in time interval $Q = 8$. Task w_b has a constant execution time of three time units, and a budget $R = 3$ in time interval $Q = 4$. The capacity of the buffer is two containers. A data-driven schedule of this task graph, where both tasks are scheduled on different TDM scheduled processors is shown in Figure 5.3(a).

According to Theorem 5.3, the SRDF graph from Figure 5.5, with firing durations $\rho(v_{a1}) = 6$, $\rho(v_{a2}) = 4$, $\rho(v_{b1}) = 1$, and $\rho(v_{b2}) = 4$, is temporally conservative to the task graph in this example. From maximum cycle mean analysis we obtain that a strictly periodic schedule exists of this SRDF graph with a period of $6+4+1+4/2 = 15/2 = 7\,1/2$. A strictly periodic

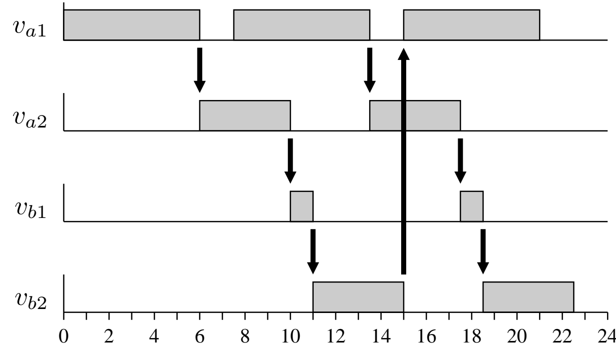


Figure 5.6: Strictly periodic schedule of SRDF graph that models a task graph that has a buffer with a capacity of two containers.

schedule with a period of seven and a half time units is shown in Figure 5.6.

Example 5.4 showed that the model with a latency and a rate parameter is not immediately more accurate than the previous model with response times, the maximum cycle mean is in fact increased. However, we see in the strictly periodic schedule of Figure 5.3(b) that actor firings are immediately subsequent to each other. A larger number of tokens in this SRDF model with response times will, therefore, not lead to a reduced maximum cycle mean. In the strictly periodic schedule of the SRDF graph with a latency and a rate parameter, as shown in Figure 5.6, we see that the maximum cycle mean can be decreased by a larger number of tokens on the cycle through all four actors. The next example makes this more concrete.

Example 5.5 Given the same task graph and SRDF graph as in Example 5.4. However, let us, in this example, assume a buffer with a capacity of four containers. Then this results in a corresponding SRDF graph, with four tokens on the queue from v_{b2} to v_{a1}, which has a maximum cycle mean of four time units. A periodic schedule of this SRDF graph with a period of four is shown in Figure 5.7.

Example 5.5 illustrates the novel aspect of the introduced dataflow component, which is that actor v_1 of the dataflow component can fire concurrently to itself. In case of a single actor with a response time both the time from enabling until finish as well as the time between subsequent finishes needs to be conservative to the modelled task, and both these aspects are captured with that single response time. In case of the presented dataflow

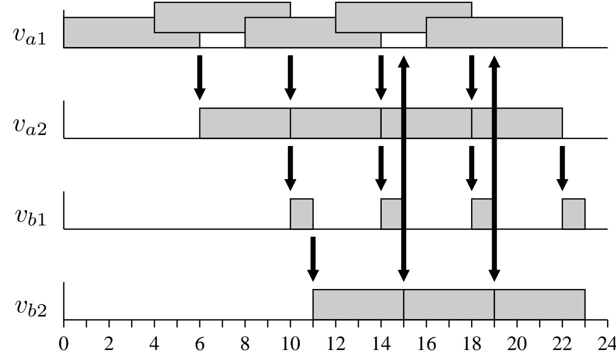


Figure 5.7: Strictly periodic schedule of SRDF graph that models a task graph that has a buffer with a capacity of four containers.

model the sum of the response times of the two actors needs to be conservative to the time between enabling and finish of the task, while the response time of actor v_2 needs to be conservative to the time between subsequent task finishes. Example 5.5 illustrates that the model with dataflow components can guarantee an accurate upper bound on container arrival times. In this example, this more accurate model requires a larger buffer than the buffer that the data-driven schedule of the task graph suggests to be sufficient. However, the presented model is tight, and the accuracy of the model depends on the specific execution times.

5.3.3 Modelling a Sequence of Task Executions with a Dataflow Component Firing

In Section 4.5.2, we generalised the required relation between task graph and dataflow graph from the one-to-one correspondence between task executions and actor firings to a one-to-one correspondence between a sequence of task executions and an actor firing. Similarly, we, in this section, generalise from a one-to-one correspondence between task executions and component firings to a one-to-one correspondence between a sequence of task executions and a component firing. This generalisation is required to model a task graph with a general topology.

Property 5.2 allows a component firing to model a sequence of task executions, as illustrated in Figure 5.8. Property 5.2 generalises Property 5.1, which allows a component firing to model a single task execution.

Property 5.2 *For every task in the task graph, there is a unique dataflow component in the dataflow graph. For the relation between the input and output buffers of a task and the input and output queues of the correspond-*

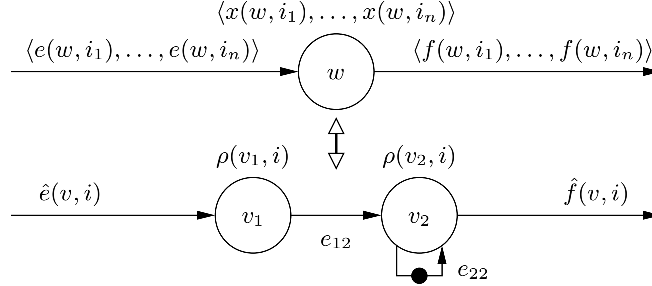


Figure 5.8: Required relation between task and dataflow component, given a one-to-one relation between sequences of task executions and component firings.

ing dataflow component, Property 4.1 holds. Furthermore, the component consists of two actors, v_1 *and* v_2*, and queues* e_{12} *and* e_{22}*, with* $\mathbf{src}(e_{12}) = v_1$*,* $\mathbf{dst}(e_{12}) = v_2$*, and* $\delta(e_{12}) = 0$*, and* $\mathbf{src}(e_{22}) = v_2$*,* $\mathbf{dst}(e_{22}) = v_2$*, and* $\delta(e_{22}) = 1$*. Actor* v_1 *has a set of firing rules such that for the sequence of task executions* $i = \langle i_1, \ldots, i_n\rangle$ *there is a unique firing rule in this set of firing rules that is satisfied iff the tokens that correspond with the containers consumed by the sequence of task executions* i *are present. Furthermore,* v_1 *produces only on queue* e_{12} *on which it produces every firing a single token. Actor* v_2 *has a firing rule that is satisfied iff on both queue* e_{12} *and on queue* e_{22} *a token is present. Firing* i *of actor* v_2 *produces tokens on queues such that there is a one-to-one correspondence with the containers that are produced on buffers by the sequence of executions* i *of the corresponding task.*

Because a component firing is defined as a set of two actor firings, Theorem 4.4 can be relatively straightforwardly generalised to obtain Theorem 5.6, which holds for component firings instead of actor firings. Further, while Theorem 5.4 holds given a one-to-one relation between task executions and component firings, Theorem 5.6 holds given a one-to-one relation between sequences of task executions and component firings.

Theorem 5.6 *Given that Property 5.2 holds for task graph* T *and dataflow graph* G*. If Equation* (5.15) *holds for any task execution* i_p *that is part of the sequence of task executions* i *of a task* w*, then* G *is temporally conservative to* T*.*

$$e(w, i_p) \leq \hat{e}(k, i) \Rightarrow f(w, i_p) \leq \hat{f}(k, i) \tag{5.15}$$

Component k *corresponds to task* w *and component firing* i *corresponds to the sequence of task executions* i*.*

Proof. By Definition 4.14, dataflow graph G is temporally conservative to task graph T, if given a starting situation in which all token arrival times are conservative no firing can lead to token arrival times that are not conservative. In G, component firings consume and produce the same amount of tokens as the cumulative number of containers consumed and produced by their corresponding sequence of task executions. Furthermore, task executions consume containers not before their start and produce containers not after their finish, while component firings consume tokens at their start and produce tokens at their finish. This implies that if Equation (5.15) holds, then no component firing produces its tokens earlier than the corresponding task execution produces its containers. This implies that given token arrival times that are conservative every component firing leads to token arrival times that are again conservative, which implies that dataflow graph G is temporally conservative to task graph T. □

For the same reasons as in Section 5.1.2, a sequence of task executions can be modelled by substitution of the execution time for a sum of execution times in the firing duration expressions of the dataflow model.

Theorem 5.7 *Given that Property 5.2 holds for dataflow graph G and task graph T. If component firing i that corresponds to the sequence of task executions $i = \langle i_1, \ldots, i_n \rangle$ has a firing duration of firing i of actor v_1 that is equal to $Q - R$ and a firing duration of firing i of v_2 that is equal to $\frac{Q \cdot \sum_{m=1}^{n} x(w, i_m)}{R}$, then dataflow graph G is temporally conservative to task graph T.*

Proof. The proof is completely analogous to the proof of Theorem 5.2, which in summary is as follows. We first assume that the external enabling times are delayed until a time that we know that sufficient containers are present to let the sequence of task executions be viewed as a single task execution, with an execution time equal to the sum of execution times of this sequence. For these delayed external enabling times the model is conservative by Theorem 5.3. Because the task is scheduled on a budget scheduler, an earlier external enabling and use of budget before the delayed external enabling does not decrease the guaranteed minimum budget that is assumed by the model, therefore the model is also conservative for earlier external enabling times of the task executions. □

To emphasise that the dataflow component with two actors has one actor that is the destination of all queues from other components and has one actor that is the source of all queues to other components, the SRDF graph of a task graph with three tasks is shown in Figure 5.9. See Example 4.4 for an example description of this task graph. In this dataflow graph, actor v_{b1} is the destination of the queues from actor v_{a2} and from actor v_{c2}, and v_{b2} is the source of the queues to actor v_{a1} and actor v_{c1}.

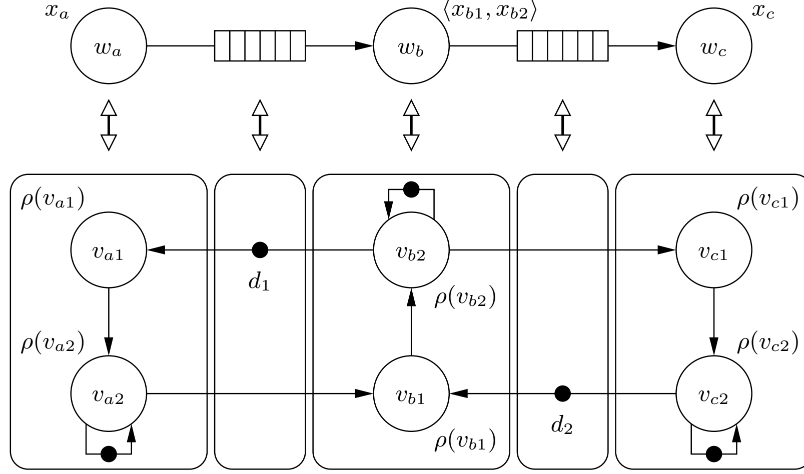


Figure 5.9: Correspondence between task graph and dataflow graph with dataflow components.

The relation between task graph and dataflow graph can be further generalised to let one task be modelled by more than two actors. The requirement is that these actors that are additional to the two actors of the dataflow component defined in Property 5.1 are not the source or destination of any queue from another component. With this requirement the concept of a component firing is unchanged and the dataflow model is temporally conservative if Equation (5.14) holds for all tasks and corresponding components.

We have now defined a dataflow graph to be temporally conservative to a task graph if every container arrival time is pessimistically bounded by the corresponding token arrival time. In principle, this definition could be generalised to only hold for container arrivals in specific buffers. Given this generalised definition of a conservative dataflow model, we believe that the presented theory can be further generalised to model m tasks by n dataflow actors.

5.4 Accuracy Evaluation

In this section, we discuss three experiments that are set-up in such a way that the results can be intuitively understood. In the first experiment, we have an application with as little as possible variation, i.e. jitter, in its temporal behaviour. In this experiment, the execution times and the number of communicated containers is kept constant over task executions and the run-time arbitration can be seen as the only source of variation, i.e. jitter.

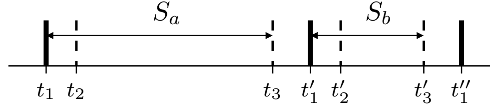


Figure 5.10: Typical sequence of actions in TDM scheduler.

In the second experiment, we introduce variation by letting the execution time of one task alternate between two values. In the third experiment, we introduce another source of variation, which is a task that communicates a variable number of containers. Communication of a variable number of containers mimics behaviour as for instance found in video decoders, where a to be decoded frame contains a variable number of blocks.

We have implemented a small scheduling kernel that implements time-division multiplex (TDM) scheduling. With TDM scheduling there is a fixed sequence in which tasks are allocated their time slices within a period. This implies that the interval of time over which tasks are guaranteed a minimum budget is the same for all tasks and equals the TDM period. A typical sequence of events is shown in Figure 5.10. At time t_1, the scheduler sets the timer to the size of slice S_a of the next task, i.e. task a. From time t_1 to time t_2 the scheduler restores the context of task a after which task a can continue execution from time t_2. At time t_3, the processor receives an interrupt that signals that the time slice S_a is completed. From time t_3 to time t'_1 the scheduler stores the context of task a, after which at time t'_1 the timer is set with the slice S_b of task b. This sequence is repeated for all slices in a period and is the same in every period.

In all experiments, we consider an architecture with 2 ARM7 processors (SWARM 2003) that are directly connected to a double-ported memory. All instructions and (shared) data are in this memory. The advantage of this reduced set-up is that the limited number of sources of variation allows for a clean discussion of the observed differences between the simulations at different levels of abstraction. At the cost of a more elaborate model, the effects on the temporal behaviour of the application as for instance caused by resource sharing in the memory hierarchy can be included in the dataflow model (Hansson et al. 2009b). On this architecture, we have observed an upper bound on $t_2 - t_1$ of 98 cycles and an upper bound on $t'_1 - t_3$ of 249 cycles. This implies that with n time slices the TDM period equals the sum of the slices plus n times 249 cycles, while the budget allocated to a task equals the slice size minus 98 cycles. This is really a guaranteed budget since apart from the timer interrupts no other interrupts are received by this processor.

5.4.1 Experiment 1

In this first experiment we have a task graph consisting of a data producing task and a data consuming task. The data producing task produces one container in every execution and the data consuming task consumes one container in every execution. Both tasks iterate through a loop in which they first block on the arrival of a container, then do some processing, subsequently copy the result of this processing in the container, and at the end of the iteration release the container. The tasks wait on a container by polling the buffer administration. As soon as the consuming task releases a container, i.e. finishes an execution, we trigger a monitor in the simulator that prints the current time. The execution time of an execution of a task is the time between successive finishes in case the polls always succeed and this task is the only task on the processor, i.e. no TDM arbitration overhead is included. We have constructed this experiment such that we have a minimal variation in the execution times of these tasks, an upper bound on the execution time of the data producing task is 360794 cycles and an upper bound on the execution time of the data consuming task is 360796 cycles. These two tasks have a time slice of 2 Mcycles and execute on different processors. On both processors, there is one additional task that also has a time slice of 2 Mcycles.

In Figure 5.11, the first 20 finish times of the data consuming task as observed in our cycle-true simulator are shown for a buffer capacity of 3 containers. The simulation results named 'tdm-1' have been obtained by placing the data producing task as the second task that is allocated its slice by its TDM scheduler, and placing the data consuming task as the first task that is allocated its slice by its TDM scheduler. The results named 'tdm-2' have been obtained by placing the data producing task as the first task that is allocated its slice by its TDM scheduler, and placing the data consuming task as the second task that is allocated its slice by its TDM scheduler. The results named 'tdm-3' have been obtained by placing the data producing task as the first task that is allocated its slice by its TDM scheduler, and placing the data consuming task as the first task that is allocated its slice by its TDM scheduler. The results named 'dataflow' have been obtained in our dataflow simulator that together with execution of the tasks, evaluates Equation (5.12) in order to determine the finish time of the corresponding actor firing, which equals the token production time. Basically, this simulates the SRDF graph from Figure 5.5. It is known that the self-timed execution of an SRDF graph results in a periodic regime of start times of the actors (Sriram and Bhattacharyya 2000). This period is a multiple of the maximum cycle mean, where this multiple equals the number of firings of an actor within this period. In this experiment, the period is three times the mcm.

The bound on finish times and the actual finish times diverge. The reason is that with these time slices and this buffer capacity the throughput

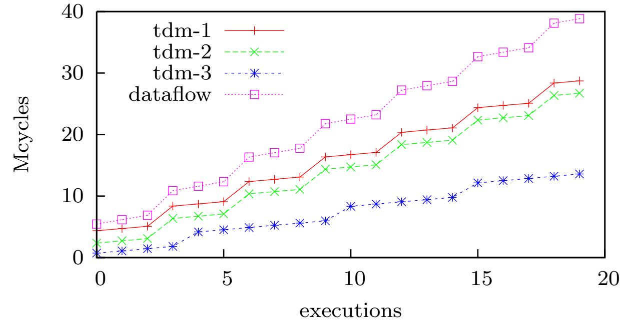


Figure 5.11: Finish times of consumer for buffer capacity of 3.

of this task graph is limited by the buffer capacity, the tasks can execute at least five times in their slice, while the buffer has a capacity of three. The consequence is that the conservative production times by the data consuming task in our dataflow simulator lead to conservative start times of the data producing task which again lead to conservative production times of the data producing task, etc. In short, since the buffer capacity determines the throughput an over-estimation of the task finish times results in a lower throughput estimate in our dataflow simulations. Furthermore, the finish times of the different TDM configurations diverge. This is because, in situation 'tdm-3', the slices of the two tasks occur at the same time, allowing for more than 3 executions per slice.

In Figure 5.12, the first 20 finish times of the data consuming task are shown for the same set-up except that now the buffer has a capacity of 8 containers. In this case the rate of the start times in the dataflow simulations closely follows the actual rate. Note that typically a strictly periodically executing sink or source task determines the throughput of stream processing applications, which implies that typically the situation depicted in Figure 5.12 occurs. In this data-driven execution, the start times are strictly periodic and the period equals the maximum cycle mean.

When comparing the results for these two buffer capacities it becomes clear that the application of polling instead of interrupts does not necessarily lead to low processor utilisation. While the data consuming task only used 50% of its budget for the trace shown in Figure 5.11, where we had a buffer capacity of three, virtually the complete budget was used by the data consuming task for the trace shown in Figure 5.12, where we increased the buffer capacity to eight. Since the budgets of both tasks are equal, the budget not used by the data consuming task, in case of a buffer capacity of eight, is due to the difference in execution time with the data producing task. Alternatively, but not shown, we could have decreased the

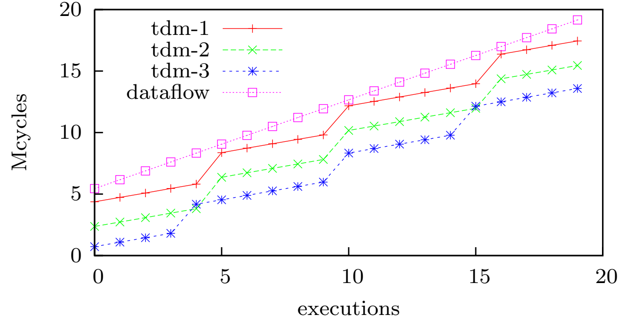


Figure 5.12: Finish times of consumer for buffer capacity of 8.

time slices instead of increasing the buffer capacity to increase the processor utilisation. This means that a suitable selection of buffer capacities and scheduler settings can result in an external enabling time of the next execution of a task that is before the finish time of the current execution. In this situation, only a single polling action per task execution is required and a high processor utilisation can be obtained.

5.4.2 Experiment 2

In this experiment, we introduced variation in the execution time of the data producing task. In an alternating fashion, subsequent executions of the data producing task have an upper bound on their execution time of 2860779 and 360803 cycles. The buffer capacity in this experiment is eight containers. In Figure 5.13, the first 20 finish times of the data consuming task are plotted as observed in our cycle-true and dataflow simulation environments. In the dataflow simulator, we have used the just described sequence of worst-case execution times when computing the finish times with Equation (5.12). Basically, the resulting dataflow simulator computes finish times of a cyclo-static dataflow model (Bilsen et al. 1996). It is known that the data-driven execution of a cyclo-static dataflow model results in a periodic regime as again confirmed by the results from our dataflow simulation. This period is a multiple of the maximum cycle mean of the corresponding single-rate dataflow graph. These finish times could have also been computed in an analytic fashion (Wiggers et al. 2007b,c; Stuijk et al. 2007).

5.4.3 Experiment 3

In this third experiment, we again changed the data producing task. Instead of producing one container in every execution, in this experiment the

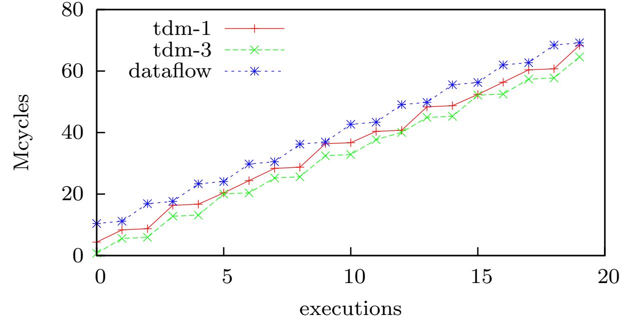


Figure 5.13: Finish times for cyclo-static execution times.

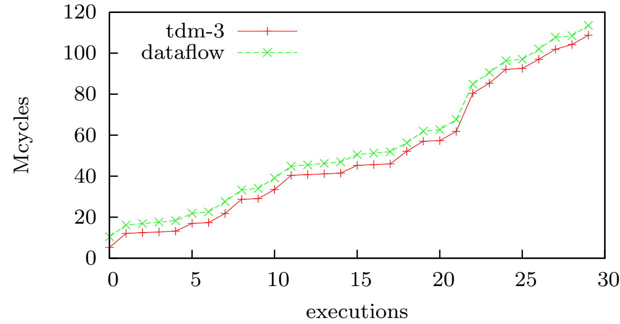


Figure 5.14: Finish times for variable production quanta.

data producing task produced between zero and five containers in every execution. We increased the execution time of the data producing task to have an upper bound of 2861527 cycles, while still having very little variation.

In Figure 5.14, the first 30 finish times of the data consuming task are shown. The simulation results named 'tdm-3' are for the case that the slices of both tasks occur at the same time. Again upper bounds on the finish times of the data consuming task are computed in our dataflow simulator using Equation (5.12) and shown in this figure under the name 'dataflow'. The finish times for the cases 'tdm-1' and 'tdm-2' are not shown but these are upper bounded by the simulation results named 'dataflow'.

This third experiment shows that the combination of budget scheduler and polling can immediately handle tasks that have aperiodic activations. Furthermore, we are still able to give tight bounds on the finish times of a task that executes aperiodically. Cyclo-static dataflow is the most ex-

pressive known dataflow model for which we can construct a corresponding single-rate dataflow graph of which the maximum cycle mean can be derived. However, the aperiodic variation in the production quanta of the data producing task cannot be modelled by a cyclo-static dataflow graph. Chapter 6 presents a dataflow model that can capture this variation in the production quanta.

5.4.4 Discussion

In these experiments, the dataflow simulator finished in a fraction of a second, while the cycle-accurate model required a couple of minutes to finish. A run of a thousand executions of the tasks of experiment 1 with a buffer capacity of eight required tens of milliseconds in our dataflow simulator and about an hour on the cycle-accurate model. However, note that we first need to determine execution times in a cycle-accurate simulator before we can use the dataflow simulator to explore various buffer capacities and scheduler settings.

5.5 Conclusion

In this chapter, we have extended the dataflow simulation and analysis approaches from Chapter 4 to include the effects of run-time scheduling by budget schedulers. First, we modelled every task by one actor, where the effects of run-time scheduling on the temporal behaviour of task executions was captured by a response time. By an intuitive example, we showed that this model can result in inaccurate bounds on the temporal behaviour of the task graph. This is because this model assumes that every task execution can start at the worst-case instant, while in fact a budget scheduler allows for task executions that are immediately consecutive. A more accurate model that additionally depends on the arrival times of containers, called latency and rate model, was introduced. However, to include this model in dataflow graphs that are amenable to dataflow analysis, we had to generalise the relation between task graph and dataflow graph to let a task be modelled by a dataflow component consisting of two actors. Experiments show that a high accuracy is obtained with a dataflow model that captures the effects of run-time scheduling with a latency and rate model.

The last experiment presented in this chapter included a task that had an aperiodic variation in the number of containers produced per task execution. While we could simulate this behaviour to obtain conservative container arrival times, no analysis technique is available to guarantee satisfaction of the timing constraints for all input streams. The next chapter presents a new dataflow model together with a technique that computes buffer capacities that satisfy given timing constraints, where this new dataflow model can capture this aperiodic variation in the container production quantum.

Chapter 6

Constraint-Driven Computation of Aperiodic Multiprocessor Schedules

Abstract – *This chapter defines an algorithm that uses a new dataflow model, variable-rate phased dataflow, to compute buffer capacities that satisfy timing and resource constraints for task graphs that have inter-task synchronisation behaviour that is dependent on the processed data stream and that execute on run-time scheduled resources.*

In the two previous chapters, we discussed dataflow simulation of functionally deterministic dataflow graphs and dataflow analysis of SRDF graphs. For any functionally deterministic task graph, dataflow simulation of functionally deterministic dataflow graphs can provide guarantees on the satisfaction of timing constraints, however, only for a single input stream. On the other hand, for task graphs with inter-task synchronisation behaviour that is independent of the processed data stream, dataflow analysis of SRDF graphs can provide guarantees on the satisfaction of timing constraints. Dataflow analysis of SRDF graphs provides guarantees on the satisfaction of timing constraints for all input streams of this task graph. Buffer capacities in the task graph are reflected by initial tokens in the dataflow graph. For both dataflow simulation as well as dataflow analysis

The material in this chapter is based on (Wiggers et al. 2006, 2007a,c, 2008a,b, 2009).

the number of initial tokens is an input. This means that these techniques are applicable for given buffer capacities.

In this chapter, we present our approach to compute buffer capacities using an algorithm that applies a new dataflow model called variable-rate phased dataflow (VPDF) (Wiggers et al. 2009). Our approach allows to compute buffer capacities that satisfy timing and resource constraints for task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream. Task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream have task execution rates that are dependent on the processed data stream, which are, therefore, aperiodic. The buffer capacity computation approach presented in this chapter is the last step in a sequence of approaches presented at various conferences that started from more restrictive dataflow models.

The outline of this chapter is as follows. Section 6.1 discusses the relation between the approaches for increasingly expressive dataflow models. Subsequently, in Section 6.2, we introduce the variable-rate phased dataflow model. Section 6.3 presents the algorithm to compute buffer capacities and shows that these buffer capacities are indeed sufficient. While we restrict ourselves to parameters with an upper bound in Section 6.3, Section 6.4 extends this approach in order to include parameters with no upper bound. A more specific extension to the graph definition and buffer capacity algorithm is made in Section 6.5 to enable that the effects of run-time scheduling are captured with a model that has a latency and a rate parameter. Two example applications are modelled with our new dataflow model in Section 6.6. We conclude with Section 6.7.

6.1 Introduction

In this section, we first provide an overview of the relation between the expressiveness of the dataflow models used in this thesis. Subsequently, we explain the approach behind the buffer capacity computation for a restrictive dataflow model and stepwise extend this approach to more expressive dataflow models in order to arrive at the approach that is presented in detail in the remainder of this chapter.

6.1.1 Expressiveness of Dataflow Models

The ordering of the dataflow models that are discussed in this thesis is shown in Figure 6.1 as a Hasse diagram, where every instance of a dataflow model shown lower in the Hasse diagram is a valid instance of dataflow models shown higher in the Hasse diagram. This ordering is valid given that every actor has a self-edge with a single token. In SRDF (Reiter 1968), every actor firing consumes and produces a single token on each adjacent queue, while in MRDF (Lee and Messerschmitt 1987) every actor firing consumes and produces a fixed positive quantum of tokens on each

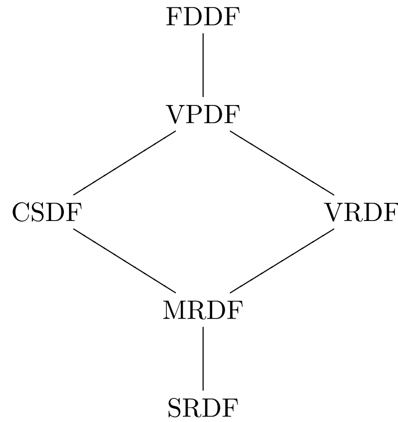


Figure 6.1: Hasse diagram of dataflow models discussed in this thesis.

adjacent queue. CSDF (Bilsen et al. 1996) has actors that have a fixed length sequence of phases. Each phase is fired once and this sequence of phases is repeated infinitely often. In CSDF, every actor firing consumes and produces a fixed non-negative quantum of tokens on each adjacent queue. VRDF (Wiggers et al. 2008b) is a generalisation of MRDF, in which every actor firing consumes and produces a quantum of tokens on each adjacent queue, where this quantum is from a given interval of non-negative values. This quantum is allowed to change from actor firing to actor firing. VPDF is a generalisation of CSDF and introduced in (Wiggers et al. 2009). VPDF also has actors with a fixed length sequence of phases, however the number of times a phase is fired is parameterised and attains a value from a given interval of non-negative values. Furthermore, as in VRDF, each actor firing in VPDF consumes and produces a parameterised quantum of tokens on each adjacent queue, where this parameter attains a value from a given interval of non-negative values. The values that are associated with these parameters of a VPDF actor can change once in every iteration through all phases of this actor. Functionally deterministic dataflow (FDDF) is defined in Chapter 4 and only requires that firing rules are sequential and that firings are functional. Every VPDF graph is a FDDF graph and FDDF is the most expressive model that is considered in this thesis.

6.1.2 Buffer Capacity Computation Approach

The approach to compute buffer capacities with VPDF graphs as presented in this chapter is the result of work that started with an approach to compute buffer capacities with MRDF graphs (Wiggers et al. 2006) followed

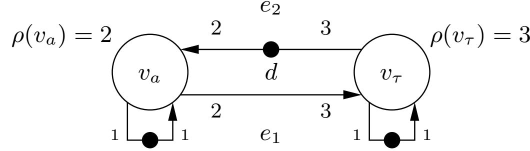


Figure 6.2: Example MRDF graph.

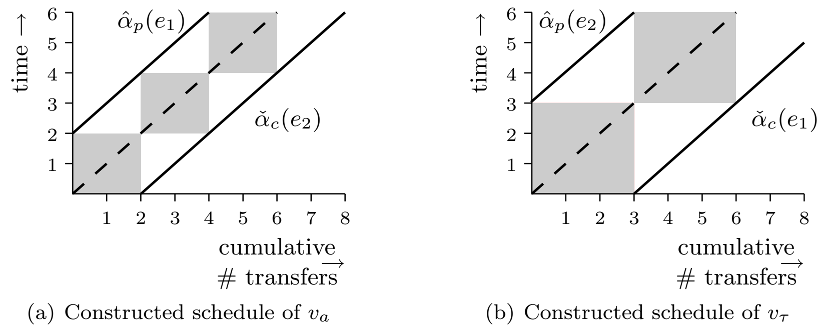
by generalisations to CSDF (Wiggers et al. 2007a,c), VRDF (Wiggers et al. 2008a,b), and finally VPDF (Wiggers et al. 2009). We will first discuss the basic idea of the approach for MRDF graphs and subsequently discuss how this approach is extended to the other dataflow models.

Multi-Rate Dataflow We illustrate the approach to compute buffer capacities with MRDF graphs from (Wiggers et al. 2006) with the MRDF graph of Figure 6.2. Assume that this MRDF graph models a task graph with a data producing task w_a, modelled by v_a, and a data consuming task w_τ, modelled by v_τ, where w_τ is required to execute strictly periodically with a period of three time units.

For this MRDF graph, we have that for every two firings of actor v_τ there are three firings of actor v_a required to return all tokens to their initial queues. If we construct for actor v_a a periodic schedule with three firings in every six time units and for actor v_τ a periodic schedule with two firings in every six time units, then we have constructed a schedule for this graph that satisfies the timing constraint. Furthermore, a bounded number of tokens is sufficient to sustain this schedule indefinitely. Given these schedules per actor, we derive, on each adjacent queue, a linear upper bound on the production time of tokens, $\hat{\alpha}_p$, and a linear lower bound on the consumed number of tokens, $\check{\alpha}_c$, as shown in Figures 6.3(a) and 6.3(b). In this graph, e_1 is the queue from actor v_a to actor v_τ and e_2 is the queue from actor v_τ to actor v_a.

Given these bounds on token production and consumption times, we derive a sufficient minimum start time of the first firing of actor v_τ relative to the start time of the first firing of actor v_a such that tokens are not consumed before they are produced, given that both actors sustain the constructed periodic schedule. As illustrated in Figure 6.4, a start time of actor v_τ that is five time units later than the start time of actor v_a is sufficient to let actor v_τ consume tokens after they are produced by actor v_a. This start time of five time units follows when we let the upper bound on token production times on queue e_1 equal the lower bound on token consumption times on this queue e_1.

Given the constructed schedules and the start times of these schedules, a sufficient number of tokens on the queue from actor v_τ to actor v_a can



(a) Constructed schedule of v_a

(b) Constructed schedule of v_τ

Figure 6.3: Constructed schedules of firing shapes with their bounds.

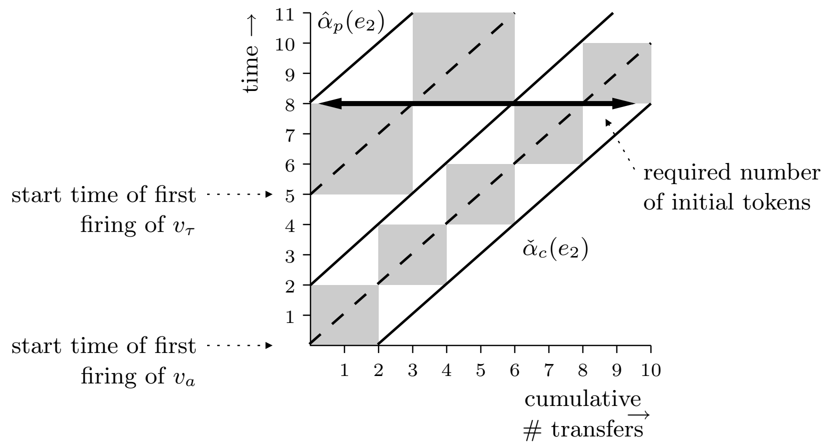


Figure 6.4: Constructed schedule of MRDF graph, with the horizontal double-headed arrow indicating the required ten tokens.

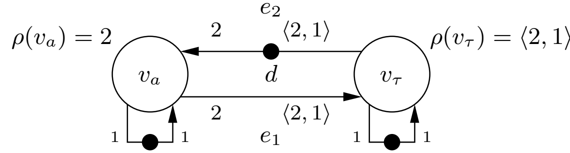


Figure 6.5: Example CSDF graph.

be determined. This number of tokens equals the number of tokens that are consumed but not yet produced on this queue, given the constructed schedules. As illustrated in Figure 6.4, ten initial tokens are sufficient to sustain the constructed schedules that satisfy the timing constraints. This number of tokens is a sufficient buffer capacity. This is because of the required one-to-one correspondence between task graph and MRDF graph and because MRDF graphs are temporally monotonic in the start times and firing durations. Temporal monotonicity in the start and firing durations means that earlier starts and smaller firing durations cannot lead to later token production times, because of the required one-to-one correspondence this implies that containers cannot arrive later than the token arrival times in the constructed schedules that satisfy the timing constraints.

The described algorithm constructs for each actor a strictly periodic schedule. This has the benefit of a low computational complexity, at the cost of sub-optimality. Optimal schedules for MRDF graphs have multi-dimensional periodic schedules per actor (Verhaegh et al. 2001). The sub-optimality of the approach implies that our algorithm might not find a schedule that satisfies the constraints on throughput and buffer capacities, while there does exist a multi-dimensional schedule that satisfies these constraints. An evaluation of the accuracy of the presented approach is provided both in (Wiggers et al. 2006) as well as in Chapter 7.

Cyclo-Static Dataflow If task w_τ of the task graph modelled by the MRDF graph of Figure 6.2 infinitely often has the sequence of first acquiring two containers and then acquiring one container, then this task is more accurately modelled by the CSDF graph of Figure 6.5.

The buffer capacity computation for CSDF graphs from (Wiggers et al. 2007a,c) is an extension of the algorithm for MRDF graphs. CSDF actors have a sequence of phases. The algorithm for CSDF graphs constructs a schedule per actor in which this sequence of phases starts strictly periodically such that the timing constraint is satisfied. Within this period the firings of the phases are scheduled. The constructed schedule for actor v_τ of the CSDF graph of Figure 6.5 is shown in Figure 6.6. Compared to the MRDF model, the CSDF model results in a smaller difference between the upper bound on production times on e_2, which is the queue from v_τ to v_a,

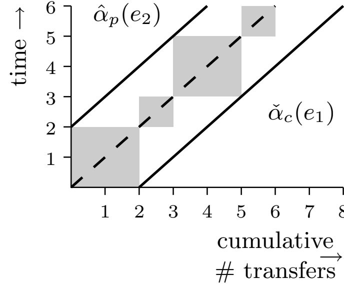


Figure 6.6: Constructed schedule of CSDF actor v_τ.

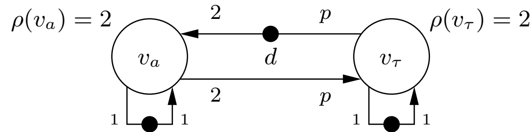


Figure 6.7: Example VRDF graph, with $p \in \{1, 2\}$.

and the lower bound on consumption times on e_1, which is the queue from v_a to v_τ. This results in a smaller number of required tokens. For reasons of brevity, we do not show how the schedules and bounds on token transfer times from Figure 6.4 are changed. However, in this example, eight tokens are sufficient to sustain the schedules as shown in Figure 6.3(a) and 6.6.

In the constructed schedules, the sequence of phases starts strictly periodically. If the timing constraint results in a period that is larger than the sum of firing durations of the phases, then there is scheduling freedom. In (Wiggers et al. 2007a,c) two heuristics are presented to schedule the firings of the individual phases to use this scheduling freedom to obtain further reduced buffer capacities.

Variable-Rate Dataflow Task w_τ that subsequently acquires two and one container is best modelled by a CSDF actor. However, to illustrate the approach to compute buffer capacities with VRDF graphs, we model this task by the VRDF actor v_τ that is shown in Figure 6.7. Because we cannot model an alternating sequence of phases with VRDF, we need to relax the timing constraint to a period of four for task w_τ. In the model this results in periodic firings of actor v_τ with a period of two.

In (Wiggers et al. 2008a,b) it is shown that a sufficient number of tokens can be determined for the VRDF graph from Figure 6.7 by taking the maximum token production and consumption quanta for v_τ and constructing the schedule as shown in Figure 6.8(a). It is shown that the

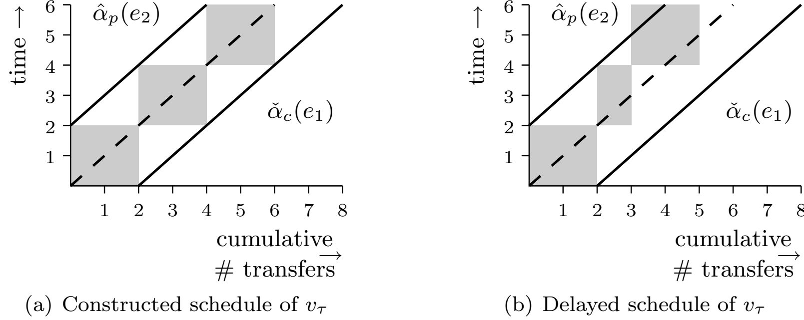


Figure 6.8: Constructed schedules of firing shapes with their bounds.

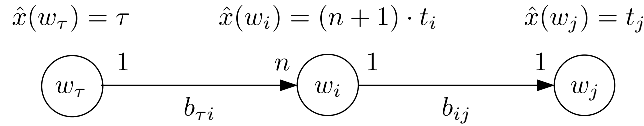


Figure 6.9: Task graph that can be modelled by Variable-Rate Dataflow.

timing constraint is satisfied for every sequence of token transfer quanta even though the smaller quantum of one token will lead to a schedule that has token production times that are no longer conservatively bounded by the bounds from Figure 6.8(a). The timing constraints are satisfied for every sequence, because VRDF graphs have linear temporal behaviour. VRDF graphs have linear temporal behaviour, because VRDF graphs are functionally deterministic dataflow graphs and Theorem 4.1 states that functionally deterministic dataflow graphs have linear temporal behaviour. As defined in Definition 4.12, linear temporal behaviour implies that if a token production time is delayed by Δ, then this cannot result in token arrival times that are delayed by more than Δ. In Figure 6.8(b), it is illustrated that tokens will not only be produced Δ later as a consequence of the smaller token transfer quantum, but that tokens will also be consumed Δ later as a consequence of the smaller token transfer quantum. Therefore, if tokens arrive on time to sustain the schedule from Figure 6.8(a), then by linearity tokens will arrive on time to also sustain the delayed schedule from Figure 6.8(b).

VRDF can model task graphs that cannot be modelled by CSDF graphs. An example task graph for which buffer capacities can be determined using VRDF is shown in Figure 6.9. For this task graph, buffer capacities should be determined such that task w_τ can execute strictly periodically with

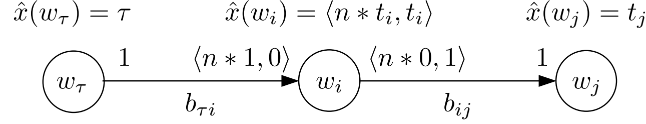


Figure 6.10: Task graph that can be modelled by Variable-Rate Phased Dataflow.

period τ. In this task graph, task w_i has a loop with n iterations that each read one container from buffer $b_{\tau i}$. After the loop a container is written on buffer b_{ij}. Since n is not a constant value, this task graph cannot be modelled by MRDF or CSDF. Furthermore, variation in token transfer quanta has peculiar aspects. For example, the graph from Figure 6.2 can execute deadlock-free with $d = 4$ tokens, while if v_a would transfer 3 instead of 2 tokens only $d = 3$ tokens are required for deadlock-free execution of the graph. This means that maximising the token transfer quanta does not result in a buffer capacity that guarantees deadlock-free execution.

The VRDF model is constructed such that it allows an efficient computation of buffer capacities that are sufficient to guarantee satisfaction of the constraints. The requirement for VRDF is that task w_i from Figure 6.9 waits on n containers on buffer $b_{\tau i}$ and on one container on buffer b_{ij} before it starts the loop. However, the number of loop iterations is not always known, for instance if task w_i is a variable-length decoder then the number of iterations depends on the processed stream and is only determined during execution of the loop. Furthermore, the required buffer capacity on buffer $b_{\tau i}$ grows with the maximum value of n, and the buffer capacity is unbounded if n is unbounded.

Variable-Rate Phased Dataflow In this chapter, we discuss a new dataflow model that is an extension of VRDF called Variable-Rate Phased Dataflow (VPDF) (Wiggers et al. 2009) that has phases of execution, where these phases can fire a variable number of times. This implies that we can now distinguish the individual iterations of a loop, for example of task w_i as shown in Figure 6.10.

In the task graph of Figure 6.10, task w_i reads n times a single container, denoted by $n * 1$, and can exit the loop based on the data in this container, i.e. the value of n does not need to be known before the loop is started. Furthermore, the capacity of buffer $b_{\tau i}$ is independent of n, i.e. a bounded buffer capacity exists even if n is unbounded. An additional benefit is that the worst-case time between consumption and production of containers in Figure 6.10 is t_i instead of $(n + 1) \cdot t_i$ as in the VRDF graph of Figure 6.9. This enables the derivation of smaller buffer capacities and end-to-end latency.

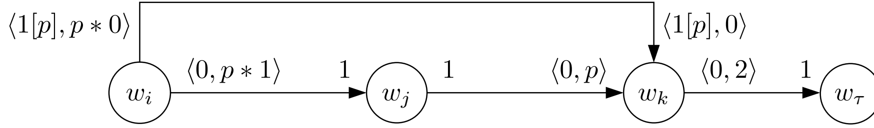


Figure 6.11: Task graph with conditional execution of task w_j, because p can attain zero.

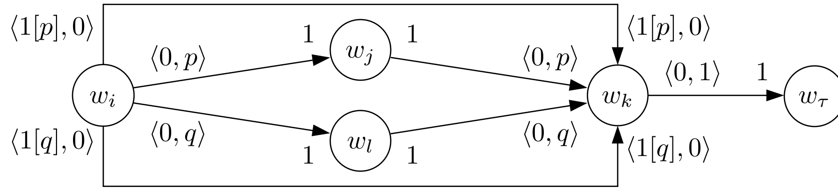
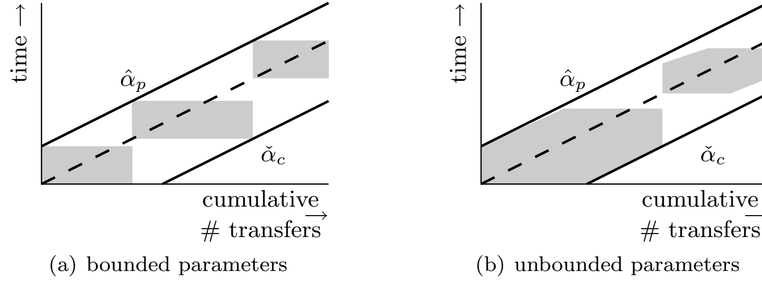


Figure 6.12: With $p, q \in \{0, 1\}$, tasks w_i and w_k model switch and select tasks.

VPDF can model conditional execution of tasks, as also VRDF can model conditional execution of tasks. An example task graph is shown in Figure 6.11, where task w_i first produces a single container holding the value of p, denoted by $1[p]$, after which it produces p times a single container, denoted by $p * 1$. Task w_k first consumes the value of p before it consumes p containers. Since we allow p to have the value zero, this leads to conditional execution of task w_j. VPDF can also model switch and select tasks, as exemplified by Figure 6.12. The analysis will compute buffer capacities that are sufficient to guarantee satisfaction of the timing constraints for all combinations of parameter values, which in this case are four combinations. Note that this is conservative, if, in the application code modelled by Figure 6.12, the values of parameters p and q are always each others complement, i.e. behave like an if-then-else, and only two combinations of parameter values are actually possible.

The VPDF dataflow model is defined such that for every task graph, which can be modelled by a valid VPDF dataflow graph, buffer capacities can be computed that satisfy a throughput constraint and any constraints on maximum buffer capacities. This is shown in Theorem 6.5. Our algorithm is conservative, which means that it can occur that the algorithm rejects VPDF graphs that have a set of constraints which actually can be satisfied. VPDF is defined such that validity of VPDF graphs is decidable and can be efficiently checked. As defined in Section 6.2, the validity of a VPDF graph is determined by a consistency check on the firing rates and a check on the graph topology.



(a) bounded parameters (b) unbounded parameters

Figure 6.13: Aggregate firing shapes.

The algorithm to compute sufficient buffer capacities with VPDF is a generalisation of the algorithm for VRDF, and has as important contribution the concept of an aggregate firing. Compared to the algorithm for VRDF graphs, the concept of an aggregate firing introduces a third level of scheduling. For VRDF graphs, for each individual actor a schedule of firings was determined that satisfied the throughput constraint. Subsequently, for each actor the start time of the first firing was determined such that on each queue tokens arrive before they are consumed in the constructed schedules. For VPDF graphs, firings are first scheduled per iteration through all phases to form an aggregate firing. Then these aggregate firings perform the role that firings have in the algorithm for VRDF graphs. This concept of an aggregate firing can also be applied on dataflow models that have fixed sequences of firings that each produce and consume a fixed number of tokens. For example, firings of MRDF actor v_τ from Figure 6.2 are aggregate firings of the firings of actor v_τ from Figure 6.5. While aggregate firings of CSDF actors have firing durations and token transfer quanta that are fixed over all aggregate firings, aggregate firings of VPDF actors have firing durations and token transfer quanta that vary over different aggregate firings.

Figure 6.13 shows schedules of aggregate firings together with a linear upper bound on token production times $\hat{\alpha}_p$ and a linear lower bound on token consumption times $\check{\alpha}_c$. These bounds are used to compute buffer capacities. Schedules of aggregate firings are constructed such that the left-bottom corner of the aggregate firing shape is on the dashed line. As a result, the schedule of aggregate firing shapes remains between the linear bounds. The aggregate firing shapes bound the production and consumption times of the actor firings within the aggregate firings. In Section 6.3, we introduce aggregate firings that have a rectangular shape as in Figure 6.13(a). However, the size of the rectangular firing shape grows as the number of firings per iteration through all phases grows. Therefore, with rectangular aggregate firing shapes, a growing number of firings results in a

growing difference between these bounds, which in turn results in growing computed buffer capacities. This means that these aggregate firings are not suitable in case of an unbounded number of firings per phase. Therefore, in Section 6.4, we introduce aggregate firings that have an aggregate firing shape that grows within the linear bounds along the required transfer rate, as the number of firings increases. This is shown in Figure 6.13(b). Because the linear bounds are not affected by an increase in firings, the computed buffer capacity does not increase with a growing number of firings. The next sections first define VPDF and compute buffer capacities for the case that the number of firings per phase is bounded.

6.2 Graph Definition

In this section, we first define variable-rate dataflow that only allows parameters that are local to tasks and does not support communication of parameter values between actors. Subsequently, we extend variable-rate phased dataflow to allow actors to communicate the values of a parameter. This allows to model for example task w_i in Figure 6.11 that communicates the number of loop iterations p to task w_k. This section concludes with the required relation between task graphs and variable-rate phased dataflow graphs.

6.2.1 Variable-Rate Phased Dataflow

A VPDF graph G is a directed multi-graph that is given by the tuple $G = (V, E, P, \delta, \rho, \phi, \pi, \gamma, \theta, \chi)$, and consists of a finite set of actors V and a finite set of labeled queues E.

Each actor has a finite sequence of phases. The number of phases of actor v_i is given by $\theta(v_i)$, with $\theta : V \to \mathbb{N}^*$. The number of times phase h of actor v_i is fired is parameterised and given by $\chi(v_i, h)$. The set of parameters is given by P. We define $\chi : V \times \mathbb{N}^* \to P$. Actors fire their phases in a cyclo-static fashion. The transition to the next phase, i.e. phase $(h \mod \theta(v_i)) + 1$, only occurs after phase h has fired $\chi(v_i, h)$ times.

A firing of an actor is enabled when on all input queues of the actor sufficient tokens are present. The number of tokens that are required on a queue $e \in E$ in phase h is parameterised and given by $\gamma(e, h)$. The function γ is defined as $\gamma : E \times \mathbb{N}^* \to P$. Similarly, the parameterised number of tokens produced in phase h, i.e. the parameterised token production quantum, is given by $\pi(e, h)$, where we define $\pi : E \times \mathbb{N}^* \to P$. We require that for every phase, the number of firings of this phase is parameterised in a different parameter than the parameters in which the token transfer quanta of this phase are parameterised. With each parameter $p \in P$ a set of integer values is associated, which is given by $\phi(p)$. We define $\phi : P \to \mathcal{P}(\mathbb{N})$, where $\mathcal{P}(\mathbb{N})$ denotes the set of all subsets of $\mathbb{N}$ excluding the empty subset. Every parameter $p \in P$ is only associated with a single

phase of a single actor v_i. Further, every parameter p is only allowed to be assigned a value once per iteration through all $\theta(v_i)$ phases. Furthermore, the value of a parameter is required to be a function of previously consumed tokens.

The number of initial tokens on queue e is given by $\delta(e)$, with $\delta : E \to \mathbb{N}$, while the firing duration of an actor $v \in V$ in phase h is given by $\rho(v, h)$, with $\rho : V \times \mathbb{N}^* \to \mathbb{R}^+$. In any phase h of actor v, tokens are consumed in an atomic action at the start of a firing and tokens are produced in an atomic action $\rho(v, h)$ later at the finish of the firing. An actor does not start a firing before every previous firing has finished.

We define the following convenience functions. For an actor v_i, the function $E(v_i)$ provides the set of queues adjacent to v_i. We define the parameterised cumulative token production quantum on queue e_{ij} as $\Pi(e_{ij}) = \sum_{h=1}^{\theta(v_i)} \chi(v_i, h) \cdot \pi(e_{ij}, h)$, and the parameterised cumulative token consumption quantum on e_{ij} is $\Gamma(e_{ij}) = \sum_{h=1}^{\theta(v_j)} \chi(v_j, h) \cdot \gamma(e_{ij}, h)$. For an actor v_i, the function $P(v_i)$ provides the set of parameters in which the cumulative token production quanta on queues from v_i and the cumulative token consumption quanta on queues to v_i are parameterised.

We require that on each queue $e \in E$ there is a parameter valuation such that $\Pi(e) > 0$ and there is a parameter valuation such that $\Gamma(e) > 0$. Furthermore, we only consider strongly connected VPDF graphs.

We will now discuss consistency of dataflow graphs and present a check on the consistency of VPDF graphs. For inconsistent VPDF graphs, the throughput constraint cannot be satisfied in bounded memory.

Consistency On each queue of a VPDF graph the parameterised production and consumption quanta specify a relation between the execution rates of the two adjacent actors. If there are two directed paths that connect a given pair of actors and that specify inconsistent relations between the execution rates of this pair of actors, then either for any finite number of initial tokens this sub-graph will deadlock or there is an unbounded accumulation of tokens. Therefore, in order to verify whether there exists a bounded number of initial tokens that is sufficient to satisfy the throughput constraint, we need to check whether the relation between the execution rates of each pair of actors is strongly consistent (Bhattacharya and Bhattacharyya 2002; Lee 1991) over all paths between these two actors. We define strong consistency as the requirement that the constraints on execution rates are consistent for every valuation of the token transfer parameters.

On a queue e_{ij} from actor v_i to actor v_j, we have the requirement that it should hold that $z_i \cdot \Pi(e_{ij}) = z_j \cdot \Gamma(e_{ij})$, where actor v_i executes its $\theta(v_i)$ phases proportionally z_i times and actor v_j executes its $\theta(v_j)$ phases proportionally z_j times. Similarly to (Lee 1991; Bilsen et al. 1996), we collect all these requirements in matrix notation, i.e. we require a non-

trivial (symbolic) solution to exist for $\Psi\mathbf{z} = \mathbf{0}$, to verify whether a VPDF graph is strongly consistent. The matrix Ψ is an $|E| \times |V|$ matrix, where

$$\Psi_{ij} = \begin{cases} \Pi(e_i) & \text{if } e_i = (v_j, v_k) \\ -\Gamma(e_i) & \text{if } e_i = (v_k, v_j) \\ \Pi(e_i) - \Gamma(e_i) & \text{if } e_i = (v_j, v_j) \\ 0 & \text{otherwise} \end{cases}$$

In the matrix Ψ, each parameter p that can only attain a single value, i.e. $|\phi(p)| = 1$, is substituted by this value. The smallest positive integer $\mathbf{z}$ is called the (symbolic) repetition vector of the VPDF graph, and z_i is an element of this repetition vector that denotes the (symbolic) repetition rate of v_i. In the remainder of this chapter, we only consider strongly consistent VPDF graphs.

For example, if we substitute an actor v_k that consumes and produces a single token in every firing on all its adjacent queues for the subgraph G'_p in Figure 6.14, then we obtain the following topology matrix and system of linear equations for this VPDF graph.

$$\begin{bmatrix} 1 & -2 & 0 & 0 \\ -1 & 2 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \\ 0 & p & -1 & 0 \\ 0 & -p & 1 & 0 \\ 0 & 0 & 1 & -p \\ 0 & 0 & -1 & p \end{bmatrix} \begin{bmatrix} z_\tau \\ z_i \\ z_k \\ z_j \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The repetition vector is given by the repetition rates $z_\tau = 2$, $z_i = 1$, $z_k = p$, and $z_j = 1$.

The functional behaviour of a VPDF graph is deterministic in the sense that it is schedule independent, because the firing rule equals the token consumption quanta in that firing and the production rule of a firing equals the token production quanta in that firing. Because the parameters that specify the token consumption and production quanta are a function of previously consumed tokens, we have that the firing rules are sequential (Lee and Parks 1995) and that the firings are functional. These are sufficient conditions for the VPDF graph to be functionally deterministic. As shown in Chapter 4, functionally deterministic dataflow graphs have monotonic and linear temporal behaviour. Therefore, also VPDF graphs have monotonic and linear temporal behaviour.

6.2.2 Parameter Distribution

In Section 6.2.1, we required that every parameter $p \in P$ is only associated with a single actor v_i. In this section, we relax this constraint and allow for

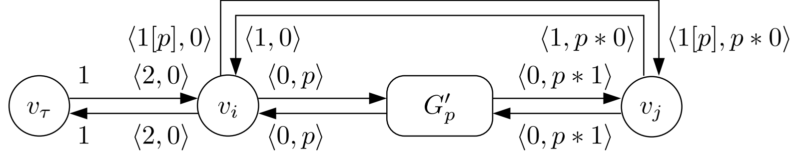


Figure 6.14: VPDF graph with shared parameter p. Actors v_i and v_j have two phases, where the second phase of v_j is fired p times. The value of p is determined by v_i and send to v_j.

every parameter p one queue e_p to exist over which the values of this parameter are communicated. Let e_p be from v_i to v_j, then the values of this parameter p are determined in firings of v_i. Actor v_i is required to produce a value on e_p in a phase previous to the phases that have their number of firings or token transfer quanta parameterised in p. Further, actor v_j is required to consume a value from e_p in a phase previous to the phases that have their number of firings or token transfer quanta parameterised in p. Furthermore, we require that on this queue e_p, we have $\delta(e_p) = 0$ and that $\Pi(e_p) = \Gamma(e_p) = 1$. We extend the dataflow graph with the function φ, which returns the parameter value that is communicated over a queue. The function φ is defined as $\varphi : E \rightarrow P \cup \{\epsilon\}$, where ϵ is an undefined parameter value. In the dataflow graph of Figure 6.14, the annotation at the end-points of a queue denote the parameterised token transfer quanta per phase. In case the value of a parameter p is transferred, then this is denoted by $1[p]$.

Next to the already mentioned restrictions, we have three additional restrictions. The first additional restriction is a restriction on the topology of the graph and results in a scoping of this parameter p. For example for the VPDF graph from Figure 6.14, this restriction does not allow v_i to use the parameter p on edges towards v_τ. Let $\Pi(e_1) \approx p$ mean that $\Pi(e_1)$ is parameterised in p, which means, with v_i producing tokens on e_1, that v_i has a phase h such that $\chi(v_i, h) = p$ or a phase k such that $\pi(e_1, k) = p$. Let $\Gamma(e_2) \approx p$ have the analogous meaning. Let parameter p be communicated from v_i to v_j. Then, intuitively, we require that every simple directed path that starts with an output queue e_1 of v_i, with $\Pi(e_1) \approx p$, includes an input queue e_2 of v_j, with $\Gamma(e_2) \approx p$, and vice versa from v_j to v_i. This can be verified as follows. Given a VPDF graph G, we create graph G_p^- by removing all output queues e_o of actors v_i and v_j for which $\Pi(e_o)$ is not parameterised in p and by removing all input queues e_i of actors v_i and v_j for which $\Gamma(e_i)$ is not parameterised in p. We require that in G_p^- the same set of actors V'_p is reachable from v_i as is reachable from v_j.

We define G_p to be the sub-graph formed by the set of actors V'_p together with the actors v_i and v_j and by all queues from G that have both

source and destination actor in G_p. The second additional restriction is that there is a positive integer repetition vector $\mathbf{z}$ of G_p in which we have that the repetition rates of v_i and v_j are equal to one, i.e. $z_i = z_j = 1$. This restriction implies that for every value of p communicated from v_i to v_j there is one iteration of G_p. Therefore, this restriction implies that a strongly consistent VPDF graph executes in bounded memory, while in general a strongly consistent dataflow graph does not need to execute in bounded memory (Buck 1993).

The third additional restriction is that actor v_τ, which models the task on which the throughput constraint is specified, is not allowed to be part of a sub-graph G_p. This means that there does not exist a parameter p that is communicated from v_i to v_j for which v_τ is on a simple directed path from v_i to v_j that starts and ends with cumulative transfer quanta that are parameterised in p. Furthermore, it is required that $v_\tau \neq v_i$ and $v_\tau \neq v_j$.

6.2.3 Required Relation to Task Graph

We require that when modelling task graphs with VPDF graphs, Property 4.3 should hold. The more general relations between task graphs and dataflow graphs as defined in Chapter 5 are not applicable, because VPDF actors implicitly have a self-edge with one token. Further, we require that if actor firing i corresponds to the sequence of task executions $i = \langle i_1, \ldots, i_n \rangle$, then actor firing i is required to have a firing duration that is equal to $r(i)$ as given by Equation (5.1), with execution time equal to $\sum_{m=1}^{n} x(i_m)$. With this requirement it follows from Theorem 5.2 that the VPDF graph is temporally conservative to the task graph.

An example task that can be modelled with a VPDF actor is task w_i from Listing 3.1, also shown in Figure 6.15. This task contains both a loop with a number of iterations that is determined before the loop is executed, i.e. the shown for-loop, and a loop with a number of iterations that is determined during execution of the loop, i.e. the shown while-loop. Important to note is that values of parameters that are shared between two actors need to be first communicated before they are used. This implies a restriction on the sharing of parameters by tasks that can be modelled by VPDF. If tasks share a parameter that indicates the number of loop iterations, then the loop is required to have a number of iterations that is determined before execution of the loop, such as the for-loop in Listing 3.1.

As discussed in Section 3.4, the requirement that containers arrive in time at the time-triggered interfaces imposes a throughput constraint. In the analysis as presented in this thesis, the time-triggered interfaces are considered as data-driven tasks. In case a VPDF model is used, we require that the throughput constraint is given by specifying a single task w_τ, which is a task that either does not have any input buffers, a source, or does not have any output buffers, a sink, and which is required to execute *wait-free.*

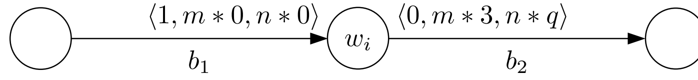


Figure 6.15: Task w_i from Listing 3.1.

A task executes wait-free, if every task execution of this task is enabled no later than the worst-case finish time of its previous task execution. The requirement to execute wait-free is a generalisation of the requirement to execute strictly periodically.

Even though task executions are enabled by the arrival of containers, and not time-triggered, we can consider a time-triggered interface as a task if we require that this task can execute wait-free. This is because the interface would be considered as a task with a single non-blocking code-segment with a constant execution time that executes on a non-shared resource. This means that this task will have a constant response time. Therefore, satisfaction of the requirement that this task should execute wait-free implies that we guarantee that strictly periodically the required number of containers is present, which implies that we can not distinguish between an event- or time-based triggering of this task w_τ.

6.3 Buffer Capacities

In this section, we use the VPDF graph to compute a number of initial tokens that directly corresponds with sufficient capacities for the buffers in the task graph. Our approach is explained through various examples and can be understood independently of the formalised reasoning, which is included to show that indeed our approach is correct for all cases.

The required number of initial tokens depends on the firing durations of the actors. Firings of dataflow actors have a constant firing duration. At the end of this section, Theorem 6.5 shows that the computed number of tokens is a sufficient buffer capacity using temporal monotonicity of VPDF and the fact that we required that the VPDF graph is temporally conservative to the task graph by Theorem 5.2. In this section, i.e. Section 6.3, we restrict ourselves to parameters with which a finite set of parameter values is associated, i.e. for which a maximum value exists. In Section 6.4, we remove this restriction and discuss our adaptations to the approach that we now discuss.

6.3.1 Outline of the Approach

The computation of the required number of initial tokens occurs in four steps, (1) computation of the maximum transfer rate per queue, (2) deriva-

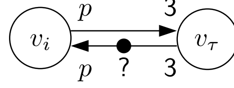
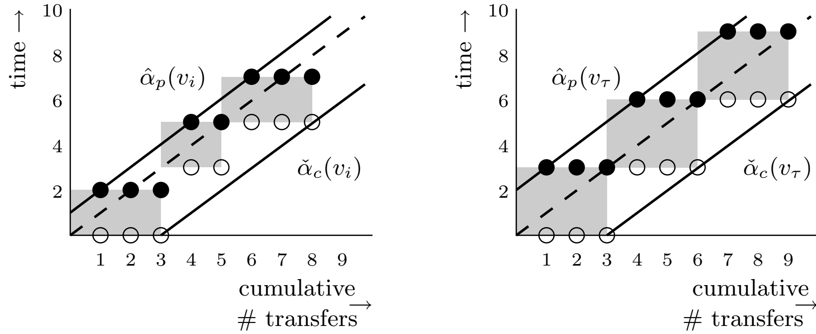


Figure 6.16: Example VPDF graph, with $\phi(p) = \{2, 3\}$, v_i has firing duration of 2 and v_τ has a firing duration of 3.



(a) Example derived schedule actor v_i. (b) Example derived schedule actor v_τ.

Figure 6.17: Schedules derived for example sequences of parameter values together with lower bound on consumption times $\check{\alpha}_c$ and upper bound on production times $\hat{\alpha}_p$.

tion of actor schedules per queue, (3) derivation of minimum distances between the start times of adjacent actors, and (4) deriving the required number of tokens.

Step 1 Given that actor v_τ is required to execute wait-free, we compute the maximally required token transfer rate on each queue of the dataflow graph. This token transfer rate is maximum in the sense that no possible parameter value can require a larger rate in order to let v_τ execute wait-free. This step is discussed in Section 6.3.3. In the VPDF graph of Figure 6.16, the maximum required transfer rate is one token per time unit.

Step 2 On each queue, we define a linear upper bound on token production times, $\hat{\alpha}_p$ and a linear lower bound on token consumption times, $\check{\alpha}_c$. The inverse of the maximum required rate on a queue is taken as the slope of linear bounds on the token production and consumption bounds on this queue. We show the existence of a schedule of actor firings for every sequence of parameter values such that these bounds are valid. This schedule is shown to exist by creating aggregate firings. Instead of a parameterised

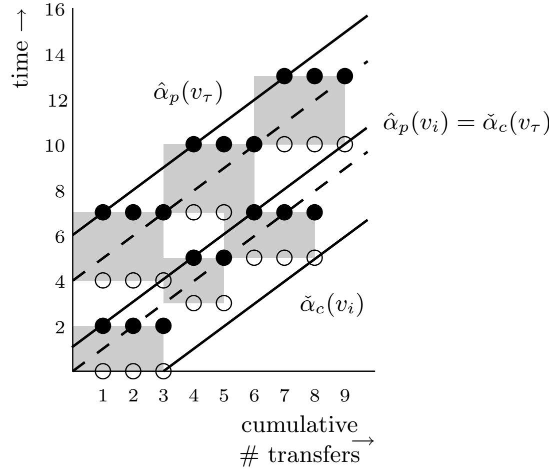


Figure 6.18: Resulting example schedule for graph of Figure 6.16.

number of firings per phase and a parameterised number of transferred tokens per firing, an aggregate firing only has one phase. An aggregate firing has both a parameterised firing duration and parameterised token transfer quanta. It is again monotonicity that tells us that if a schedule of aggregate firings exists, the schedule of actor firings will not lead to later token production times, since actor firings can only start earlier. A sufficient offset of the linear bounds relative to the start time of the first firing in the schedule of aggregate firings is determined such that the bound is conservative. This step is discussed in Section 6.3.4. The schedules derived for actors v_i and v_τ, from Figure 6.16, for particular sequences of parameter values are shown in Figures 6.17(a) and 6.17(b), respectively. In these figures, the dots denote token transfer times and the shaded rectangles are aggregate firing shapes. Every aggregate firing produces tokens at its finish, denoted by filled dots, and consumes tokens at its start, denoted by open dots. In these figures, the feasibility of an actor corresponds with the requirement that for every possible parameter value the right-top corner of the aggregate firing shape is under the dashed line. If the right-top corner of the aggregate firing shape is under the dashed line, then we can always delay the start of the next aggregate firing such that the next aggregate firing starts on the dashed line. This is our procedure to construct a schedule per queue that satisfies the maximum required transfer rate.

Step 3 On every queue, the slopes of the linear bounds on productions and consumptions are equal. On each queue, we have determined a difference between the first start time of the actors and their linear bounds

on token transfer times. Since tokens can only be consumed after they are produced, this leads to a constraint on the minimum distance between the first start times of these two actors. If there is a constraint on the maximum buffer capacity, which implies a constraint on the maximum number of initial tokens on a queue, then this leads to a constraint on the maximum distance between the first start times of the consumer and the producer. This is modelled by a constraint on the minimum distance in the opposite direction, i.e. between the first start times of the producer and the consumer. Together with the objective to minimise the distances between adjacent actors, the set of constraints on minimum distances between start times leads to a network flow problem that can be solved to obtain start times. These start times satisfy both the constraint that token consumption can only take place after token production and the constraint on maximum number of initial tokens. This step is discussed in Section 6.3.5. For our example VPDF graph from Figure 6.16, we have, with maximally zero initial tokens on the queue from v_i to v_τ and no constraint on the maximum number of initial tokens on the queue from v_τ to v_i, that v_τ should start $1^2/_3$ later than v_i. The resulting situation is shown in Figure 6.18. In a VPDF graph as shown in Figure 6.24, there are multiple paths from v_i to v_j and the minimum difference in start times between these two actors is the maximum over all these paths.

Step 4 The just derived start times together with the bounds on number of token transfers enable the derivation of the required number of initial tokens. This is because on every queue the linear bounds have the same slope and the difference between the upper bound on number of consumed tokens and the lower bound on number of produced tokens is an upper bound on the number of initial tokens that is required to enable the schedules discussed in Step 3. This step is discussed in Section 6.3.6. The required number of initial tokens on the queue from v_τ to v_i for the graph from Figure 6.16 can be derived from Figure 6.18. We have that $\hat{\alpha}(v_\tau) - \check{\alpha}(v_i) = 9$ and that the slope equals 1, which means that the maximum number of tokens consumed by v_i but not yet produced by v_τ according to these schedules equals 9 tokens. For a graph as shown in Figure 6.14, the analysis will take into account that the number of tokens on the queue from v_j to v_i needs to compensate for any additional latency of the paths through sub-graph G'_p.

The constraints on minimum distance between start times are determined using schedules for actors that are defined per queue in isolation. In Section 6.3.8, we show that the derived number of tokens is still sufficient if these schedules per queue are coupled to obtain a schedule per actor in isolation.

Furthermore, the required transfer rate on a queue depends on the parameter values. We show that for every sequence of parameter values,

the derived number of initial tokens is sufficient to let v_τ fire wait-free. There are two cases to be considered, (1) parameters that are used by only one actor and (2) parameters that are used by two actors. For the first type of parameter, we will show that other values can only lead to delayed start times of actors that are further away from actor v_τ. The fact that a firing of v_τ cannot start before every previous firing has finished leads to a delay of the third firing, and also of subsequent firings, in the schedule of v_τ shown in Figure 6.17(b). This means that this third firing will produce its tokens later on the queue to v_i, thereby potentially delaying v_i. However, since also subsequent firings of v_τ are delayed, tokens will still arrive on time from v_i. This is discussed in detail for any VPDF graph for which each parameter is only used by a single actor in Section 6.3.8.1.

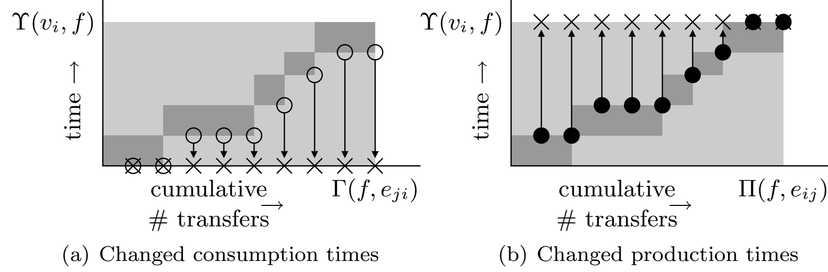
For parameters that are shared by two actors, other values also do not affect the schedule of actor v_τ. This is because by construction of VPDF graphs there is always an actor on the path to v_τ of which the schedule is unaffected by the change in parameter value. This is discussed in detail in Section 6.3.8.2.

We start this section by introducing aggregate firings. Aggregating firings allow for a closed expression for the schedules that we construct per queue in isolation. This drastically reduces the complexity of comparing schedules and reasoning about which schedule has later start times.

6.3.2 Aggregate Firings

In the reasoning in the following sections, the following additional conservative approximation is made. For each actor v_i, we aggregate all phases to a single phase. On any output queue e_{ij} of v_i this aggregate phase has a parameterised token production quantum $\Pi(e_{ij})$, and on any input queue e_{hi} of v_i this aggregate phase has a parameterised token consumption quantum $\Gamma(e_{hi})$. The parameterised firing duration of this aggregate phase is equal to the cumulative firing duration $\Upsilon(v_i) = \sum_{h=1}^{\theta(v_i)} \chi(v_i, h) \cdot \rho(v_i, h)$. We call a firing of this aggregated phase an aggregated firing. In aggregate firing f the token production quantum on e_{ij} is given by $\Pi(f, e_{ij})$, the token consumption quantum on e_{hi} is given by $\Gamma(f, e_{hi})$, and the firing duration is given by $\Upsilon(v_i, f)$. Furthermore, we use $\hat{\Gamma}(e_{ij})$ to denote the maximum consumption quantum of an aggregate firing on queue e_{ij} and $\hat{\Upsilon}(v_i)$ to denote the maximum firing duration of an aggregate firing. Both these maximum values are obtained by taking the maximum values of the parameters on which $\Gamma(e_{ij})$ and $\Upsilon(v_i)$ depend.

In Figure 6.19, an example schedule of firings is denoted by the darker firing shapes. The corresponding aggregate firing shape is denoted by the lighter colour. This aggregate firing consumes all $\Gamma(f, e_{ji})$ tokens at its start, see Figure 6.19(a), and produces all $\Pi(f, e_{ij})$ tokens at its finish, which is $\Upsilon(v_i, f)$ later than its start, see Figure 6.19(b). As illustrated in



(a) Changed consumption times (b) Changed production times

Figure 6.19: An aggregate firing is a conservative approximation of its firings.

Figure 6.19, an aggregate firing requires the same number of tokens to be present earlier than a normal firing, and delays token production times. This abstraction of multiple firings to a single aggregate firing leads to a reduced accuracy of the analysis, see Chapter 7. If there exists a schedule of aggregate firings, then this implies the existence of a schedule of normal firings. This is because normal firings can only start earlier and produce their tokens earlier than the aggregate firings. By monotonicity of VPDF graphs, we know that earlier start times and earlier token production times cannot lead to later token production times anywhere in the VPDF graph. This implies that if the number of initial tokens allows v_τ to fire wait-free in case of aggregate firings by all actors, then v_τ can still fire wait-free in case all actors have normal firings.

6.3.3 Maximum Token Transfer Rates

In this section we derive the maximum required token transfer rate of each queue that is required to guarantee that given a bounded number of initial tokens still sufficient tokens are available to let actor v_τ fire wait-free. This means that for each queue we need to find the values for all parameters of the VPDF graph that maximise the number of tokens transferred on this queue, given that v_τ fires wait-free. This section explains Step 1 of the approach that is outlined in Section 6.3.1.

The maximum required token transfer rate on queue e_{ij} is defined by Equation (6.1) that specifies that $\hat{r}(e_{ij})$ is the maximum over all possible parameter values of a certain ratio. This ratio includes z_i/z_τ, which is the ratio of the repetition rate of v_i and the repetition rate of v_τ. If each actor v_x in a graph fires z_x times, then on each queue of that graph the number of produced tokens equals the number of consumed tokens. This implies that z_i/z_τ specifies how often v_i needs to fire relative to a firing of v_τ to satisfy the requirement of a bounded number of initial tokens.

Given that v_τ fires every τ, this implies that v_i is required to fire ${}^{z_i}/_{z_\tau}$ times every τ time units, i.e. with firing rate ${}^{z_i}/_{z_\tau \cdot \tau}$. The parameterised token production rate on e_{ij} is given by the parameterised cumulative token production quantum, i.e. $\Pi(e_{ij})$, times the just derived firing rate, which equals ${}^{z_i}/_{z_\tau \cdot \tau}$. Equation (6.1) defines the maximum required token transfer rate per queue. This is a maximum rate per queue, because we only consider graphs that can execute in bounded memory, which implies that the maximum required consumption rate on a queue equals the maximum required production rate on that queue.

Definition 6.1 *The maximum required token transfer rate on queue e_{ij} is given by the maximum value that Equation* (6.1) *can attain, where $\phi(P)$ denotes the set of assignments of values to the parameters of G*

$$\hat{r}(e_{ij}) = \max_{\phi(P)} \frac{\Pi(e_{ij}) \cdot z_i}{z_\tau \cdot \tau} \tag{6.1}$$

Determining the maximum required transfer rate on a queue requires to find the parameter values that maximise the ratio ${}^{\Pi(e_{ij}) \cdot z_i}/_{z_\tau \cdot \tau}$. Theorem 6.1 will show that to determine these parameter values, instead of considering all possible combinations of parameter values, only all combinations of the extreme values that the parameters can attain need to be considered. This is because Lemma 6.2 shows this ratio is monotone in every parameter. Lemma 6.1 is used in the proof of Lemma 6.2 to rewrite this ratio.

Lemma 6.1 *Given a directed path of length n from an actor v_i to an actor v_z that consists of queues e_1 through e_n, then*

$$\frac{z_i}{z_z} = \frac{\Gamma(e_1) \cdot \ \dots \ \cdot \Gamma(e_n)}{\Pi(e_1) \cdot \ \dots \ \cdot \Pi(e_n)} \tag{6.2}$$

Proof. The proof is by induction over the queues of such a path.

Base step. Let e_{ij} be the first queue on the path. We have that $z_i \cdot \Pi(e_{ij}) = z_j \cdot \Gamma(e_{ij})$, which implies ${}^{z_i}/_{z_j} = {}^{\Gamma(e_{ij})}/_{\Pi(e_{ij})}$.

Induction step. Given that for queues e_1 to e_k, we have that $\frac{z_i}{z_x} = \frac{\Gamma(e_1) \cdot \dots \cdot \Gamma(e_k)}{\Pi(e_1) \cdot \dots \cdot \Pi(e_k)}$. Let e_{k+1} be queue e_{xy}. Then for this queue, we have that ${}^{z_x}/_{z_y} = {}^{\Gamma(e_{k+1})}/_{\Pi(e_{k+1})}$. This implies that $\frac{z_i}{z_y} = \frac{\Gamma(e_1) \cdot \dots \cdot \Gamma(e_{k+1})}{\Pi(e_1) \cdot \dots \cdot \Pi(e_{k+1})}$. □

Lemma 6.2 shows that the parameterised ratio in Equation (6.1) is monotone in every parameter. This enables a straightforward procedure as presented in Theorem 6.1 to find the parameter values that lead to the maximal value that this ratio can attain.

Lemma 6.2 *For any parameter $p \in P$ the ratio as given by Equation* (6.3) *is monotone in p.*

$$\frac{\Pi(e_{ij}) \cdot z_i}{z_\tau \cdot \tau} \tag{6.3}$$

Proof. From Lemma 6.1, we know that if there is a directed path of length n from v_i to v_τ consisting of queues e_1 through e_n, then Equation (6.3) can be rewritten into Equation (6.4). There is always such a directed path, because we only consider strongly connected VPDF graphs.

$$\frac{\Pi(e_{ij}) \cdot \Gamma(e_1) \cdot \ldots \cdot \Gamma(e_n)}{\Pi(e_1) \cdot \ldots \cdot \Pi(e_n) \cdot \tau} \tag{6.4}$$

This expression can be factored into Equation (6.5), where each factor is a ratio of cumulative transfer quanta of a single actor on this path from v_i to v_τ.

$$\frac{\Pi(e_{ij})}{\Pi(e_1)} \cdot \frac{\Gamma(e_1)}{\Pi(e_2)} \cdot \ldots \frac{\Gamma(e_{n-1})}{\Pi(e_n)} \cdot \frac{\Gamma(e_n)}{\tau} \tag{6.5}$$

A cumulative token transfer quantum is a sum of products, where the number of summands equals the number of phases and each product has as factors a firing parameter and a transfer parameter. A cumulative firing duration such as τ is also a sum of products, where again the number of summands equals the number of phases and each product now has as factors a firing parameter and a firing duration.

Let us first consider the case of parameters that are not shared between two actors. These parameters can only be present in one factor of Equation (6.5). Say that x/y is that factor from Equation (6.5) in case of parameter p. Here x stands for the numerator of this factor, which is either a cumulative token production or consumption quantum, and y stands for the denominator of this factor, which is either a cumulative token production or consumption quantum or a cumulative firing duration. Because p is only allowed to be used in a single phase of an actor, we can find α and q such that $x = \alpha p + q$ and we can find β and r such that $y = \beta p + r$, where q and r are expressions that do not include p. Because it is not allowed that a parameter is used both to parameterise the number of firings of a phase as well as a token transfer quantum of that same phase, also α and β can be found that are independent of p. This implies that the numerator x and the denominator y are linear in p. Consequently, the derivative of this factor x/y with respect to p is $\alpha\beta p + \alpha r - \alpha\beta p - \beta q / (\beta p + r)^2 = \alpha r - \beta q / (\beta p + r)^2$. In this derivative the numerator is independent of p, while the denominator is always positive. This implies that the sign of the derivative is independent of p and therefore that this factor is monotone in p.

For the case that parameter p is shared by actors v_a and v_b, we have that $z_a/z_b = 1$. This implies by Lemma 6.1 that if a directed path from v_a to v_b is part of the directed path from v_i to v_τ that the cumulative token transfer quanta that include p cancel each other out. In other words, Equation (6.5) will not include parameters that are shared by two actors that are on a path from v_i to v_τ, and every parameter will only occur in one factor of Equation (6.5). This implies that the previous reasoning for parameters that are not shared between two actors covers all parameters that are relevant for this proof. ☐

Lemma 6.2 tells that Equation (6.3) is monotone in every parameter. This means that when considering a single parameter and taking constant values for all other parameters, then this expression is monotonically increasing or decreasing in this parameter. Note, however, that whether the expression is increasing or decreasing in this parameter depends on the values chosen for the other parameters. This means that to determine the maximum required transfer rate all combinations of extreme values of the parameters need to be considered.

Theorem 6.1 *For a queue e_{ij}, the maximum required rate $\hat{r}(e_{ij})$ is found by considering all combinations of the extreme values of the parameters in which the right-hand side of Equation* (6.1) *is parameterised.*

Proof. This follows immediately from Lemma 6.2 that says that the ratio over which is maximised is monotone in every parameter. □

Theorem 6.2 establishes a necessary and sufficient condition on the cumulative firing duration of an actor v_i in order for a schedule of v_i to exist that has the maximum required rate.

Theorem 6.2 *For every actor $v_i \in V$ there exists a schedule for all parameter values such that this schedule has the maximum required rate if Equation* (6.6) *holds.*

$$\max_{\phi(P)} \frac{\Upsilon(v_i) \cdot z_i}{\tau \cdot z_\tau} \leq 1 \tag{6.6}$$

Proof. If all actors v_j in the graph fire z_j times, then on each queue the number of tokens produced equals the number of tokens consumed. The left part of Equation (6.6) finds the maximum time that it takes v_i to fire z_i times relative to the time it takes v_τ to fire z_τ times. Since v_τ should fire wait-free it is required that for all parameter values v_i requires not more time to fire z_i times than v_τ requires to fire z_τ times. □

The cumulative firing duration is a sum of products, where the number of summands equals the number of phases. Each product has as factors a firing duration and a parameter that specifies the number of firings of this phase. The cumulative production quantum on a queue is an expression with the same structure, but has transfer parameters instead of firing durations. Therefore, since Equation (6.3) is monotone in every parameter, also the expression over which is maximised in Equation (6.6) is monotone in every parameter, and the necessary condition for feasibility of the graph is verified by considering all combinations of extreme parameter values.

For example, consider the VPDF graph as shown in Figure 6.20, where v_i has a firing duration of $4t$, v_j has a firing duration of t for both phases,

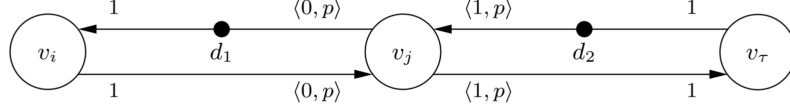


Figure 6.20: Example VPDF graph, with $p \in \{1, 2, 3\}$.

and v_τ has a firing duration of $3t$. The repetition rates of the actors in this graph are $z_i = p$, $z_j = 1$, and $z_\tau = p + 1$. The feasibility check for v_i amounts to determining the value of p that results in the maximum value of $(4t \cdot p)/3t \cdot (p+1)$. For $p = 1$, we have $(4t \cdot p)/3t \cdot (p+1) = 4/6$, while for $p = 3$, we have $(4t \cdot p)/3t \cdot (p+1) = 1$. Therefore, $p = 3$ maximises this ratio and we conclude that the firing durations of v_i allow to satisfy the throughput constraint of the graph. For v_j, we maximise $(2t \cdot 1)/3t \cdot (p+1)$. For $p = 1$, we have $(2t \cdot 1)/3t \cdot (p+1) = 2/6$, while for $p = 3$, we have $(2t \cdot 1)/3t \cdot (p+1) = 2/12$. Therefore, $p = 1$ maximises this ratio and we conclude that the firing durations of v_j allow to satisfy the throughput constraint of the graph. The firing durations of actor v_τ impose the throughput constraint on the graph.

In the VPDF graph as shown in Figure 6.20, we have that the maximal value of p determines whether v_i can satisfy the throughput constraint, while the minimal value of p determines whether v_j can satisfy the throughput constraint. This is different in a VRDF graph (Wiggers et al. 2008b), in which we have that for every parameter there is a single extreme value that triggers the worst-case behaviour.

6.3.4 Schedules per Queue

In this section we show how to compute a sufficient difference between a linear upper bound on production times and a linear lower bound on consumption times such that for every sequence of parameter values there exists a schedule between these bounds. This section explains Step 2 of the approach that is outlined in Section 6.3.1.

Given two actors v_i and v_j, and a queue e_{ij}. The linear upper bound on the production time of token x on e_{ij} under schedule $\sigma(v_i, e_{ij})$ is given by $\hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij}))$, while $\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij}))$ is the linear lower bound on the consumption time of token x on e_{ij} under schedule $\sigma(v_j, e_{ij})$. The schedules considered in this section are schedules of aggregate firings.

Let $\Xi(f, e_{ij})$ be the cumulative number of tokens produced on queue e_{ij} in aggregate firings one up to and including firing f, with $f \in \mathbb{N}^*$ and $\Xi(0, e_{ij}) = \delta(e_{ij})$, where $\delta(e_{ij})$ is the number of initial tokens on e_{ij}. The start time of the first aggregate firing of actor v_i is denoted by $s(v_i)$. We define the start time of aggregate firing f in schedule $\sigma(v_i, e_{ij})$ for v_i on

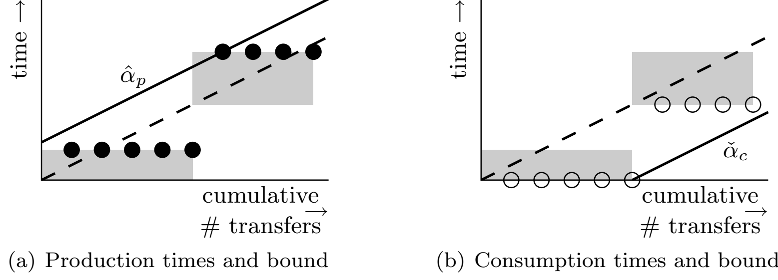


Figure 6.21: Schedules of aggregate firings on output and input queue.

queue e_{ij} by

$$s(f, \sigma(v_i, e_{ij})) = s(v_i) + \frac{\Xi(f-1, e_{ij}) - \delta(e_{ij})}{\hat{r}(e_{ij})} \tag{6.7}$$

We require that all schedules defined for an actor have an equal start time for the first aggregate firing of this actor.

This schedule implies that token $\Xi(f-1, e_{ij}) + 1$ up to and including token $\Xi(f, e_{ij})$ are produced at $s(f, \sigma(v_i, e_{ij})) + \Upsilon(v_i, f)$, i.e. at the finish time of the aggregate firing. With $s(f, \sigma(v_i, e_{ij}))$ given by Equation (6.7) and $\Upsilon(v_i, f) \leq \hat{\Upsilon}(v_i)$, an upper bound on the finish time of aggregate firing f is given by Equation 6.8.

$$s(f, \sigma(v_i, e_{ij})) + \Upsilon(v_i, f) \leq s(v_i) + \hat{\Upsilon}(v_i) + \frac{\Xi(f-1, e_{ij}) - \delta(e_{ij})}{\hat{r}(e_{ij})} \tag{6.8}$$

Let token x, with $x \in \mathbb{N}^*$, be produced by aggregate firing f, then $x = \Xi(f-1, e_{ij}) + y$, with $1 \leq y \leq \Xi(f, e_{ij}) - \Xi(f-1, e_{ij})$. Substitution of $x - y$ for $\Xi(f-1, e_{ij})$ in Equation (6.8) results in Equation (6.9).

$$s(f, \sigma(v_i, e_{ij})) + \Upsilon(v_i, f) \leq s(v_i) + \hat{\Upsilon}(v_i) + \frac{x - y - \delta(e_{ij})}{\hat{r}(e_{ij})} \tag{6.9}$$

Because $y \geq 1$, Equation (6.10) is an upper bound on the right-hand side of Equation (6.9). A linear upper bound on the production time of token x, on queue e_{ij} with schedule $\sigma(v_i, e_{ij})$ is therefore given by Equation (6.10).

$$\hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij})) = s(v_i) + \hat{\Upsilon}(v_i) + \frac{x - \delta(e_{ij}) - 1}{\hat{r}(e_{ij})} \tag{6.10}$$

An example schedule of token production times of aggregate firings is shown in Figure 6.21(a). The start time of every aggregate firing is such

that the left-bottom corner of the firing shape is on the dashed line. This means that if the first aggregate firing of Figure 6.21(a) produces more tokens, then the start time of the second aggregate firing is delayed more. As illustrated in this figure, given this schedule of aggregate firings, the upper bound on token transfer times is determined by the aggregate firing that has the largest cumulative firing duration.

Let $\Lambda(f, e_{ij})$ be the cumulative number of tokens consumed from queue e_{ij} in aggregate firings one up to and including firing f, with $f \in \mathbb{N}^*$ and $\Lambda(0, e_{ij}) = 0$. We define the start time of aggregate firing f in schedule $\sigma(v_j, e_{ij})$ for v_j on queue e_{ij} by

$$s(f, \sigma(v_j, e_{ij})) = s(v_j) + \frac{\Lambda(f-1, e_{ij})}{\hat{r}(e_{ij})} \tag{6.11}$$

This means that token $\Lambda(f-1, e_{ij}) + 1$ up to and including token $\Lambda(f, e_{ij})$ are consumed at $s(f, \sigma(v_i, e_{ij}))$. Let token x, with $x \in \mathbb{N}^*$, be consumed by aggregate firing f, then $x = \Lambda(f-1, e_{ij}) + y$, with $1 \leq y \leq \Lambda(f, e_{ij}) - \Lambda(f-1, e_{ij})$. Substitution of $x - y$ for $\Lambda(f-1, e_{ij})$ in Equation (6.11) results in Equation (6.12).

$$s(f, \sigma(v_j, e_{ij})) = s(v_j) + \frac{x - y}{\hat{r}(e_{ij})} \tag{6.12}$$

Because $\Lambda(f, e_{ij}) - \Lambda(f-1, e_{ij}) \leq \hat{\Gamma}(e_{ij})$, Equation (6.13) is a lower bound on Equation (6.12). A linear lower bound on the consumption time of token x from queue e_{ij} with schedule $\sigma(v_j, e_{ij})$ is therefore given by Equation (6.13).

$$\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij})) = s(v_j) + \frac{x - \hat{\Gamma}(e_{ij})}{\hat{r}(e_{ij})} \tag{6.13}$$

An example schedule of token consumption times of aggregate firings is shown in Figure 6.21(b). Similarly to the schedule created for a token producing actor, also in the schedule of a token consuming actor, aggregate firings have a start time such that the left-bottom corner of the firing shape is on the dashed line. This implies that a larger token consumption quantum leads to a larger delay of the next aggregate firing. Given this schedule, the lower bound on token consumption times is determined by the aggregate firing that has the largest cumulative consumption quantum.

Since, on any queue, tokens can only be consumed after they have been produced, we have that for every token x, $\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij})) \geq \hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij}))$ should hold. After substitution of Equations (6.10) and (6.13) in $\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij})) \geq \hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij}))$, we obtain Equation (6.14).

$$s(v_j) - s(v_i) \geq \frac{\hat{\Gamma}(e_{ij}) - \delta(e_{ij}) - 1}{\hat{r}(e_{ij})} + \hat{\Upsilon}(v_i) \tag{6.14}$$

6.3.5 Computation of Start Times of First Firings

The previous section derived for each queue a constraint on the minimum difference between start times of first firings of the actors adjacent to this queue. This constraint on the minimum difference between these start times is such that on this queue no tokens are consumed before they are produced, given the schedules that are determined for this queue in isolation. This section discusses how to compute the start time of the first firing of each actor given these constraints on the minimum difference between start times, this explains Step 3 of the approach outlined in Section 6.3.1.

Similar to (Wiggers et al. 2007c), we interpret the predefined number of initial tokens on a queue e, i.e. $\delta(e)$, as a constraint. This constraint on the maximum number of tokens implies a constraint on the maximum difference in start times of the adjacent actors. A constraint on the maximum difference between the start times of v_j and v_i can be reformulated as a constraint on the minimum difference between v_i and v_j, i.e. $s(v_j) - s(v_i) \leq 10$ is equivalent to $s(v_i) - s(v_j) \geq -10$. This reformulation of a constraint on the maximum difference in start times to a constraint on the minimum difference in start times is taken into account by the way the number of initial tokens affects the upper bound on token production times, leading to inclusion of $\delta(e)$ with a negative sign in Equation (6.14). These minimum differences form the constraints in the network flow problem in Algorithm 6.1 that minimises the differences between all start times subject to the mentioned constraints. This is done by introducing a dummy-actor v_0 and minimising all start times relative to the start time of a dummy-actor v_0. If there is a solution that satisfies the constraints, then solving this network flow problem results in the start times of the first firings of each actor.

The start times of the first firings of each actor are computed as follows. We construct a graph G_0 from a VPDF graph G as follows. We extend the VPDF graph with an additional actor v_0, $V_0 = V \cup \{v_0\}$, and extend the set E with queues from v_0 to every actor in V to obtain the set of edges E_0. We define the valuation function $\beta : E \to \mathbb{R}$ that with each queue e_{ij} associates the constraint on the minimum difference in start times as expressed by Equation (6.14), i.e. for queue $e_{ij} \in E$ we define

$$\beta(e_{ij}) = \frac{\hat{\Gamma}(e_{ij}) - \delta(e_{ij}) - 1}{\hat{r}(e_{ij})} + \hat{\Upsilon}(v_i) \tag{6.15}$$

while for every queue e adjacent to v_0 we have $\beta(e) = 0$. Subsequently, we solve the linear programming problem in Algorithm 6.1. By negating the start times and constraints, the problem in Algorithm 6.1 can be rewritten to a standard form of the dual of the uncapacitated network flow problem. Even more specifically it is such that it is a shortest path problem, which can be solved with Bellman-Ford. See (Bertsimas and Tsitsiklis 1997) for background information on linear programming, the dual of network flow

$$\min \sum_{v_i \in V} s(v_i)$$

subject to

$$s(v_j) - s(v_i) \geq \beta(e_{ij}) \qquad \forall e_{ij} \in E_0$$
$$s(v_0) = 0$$

Algorithm 6.1: Start time computation

problems and their relation to shortest path problems. Detection of a negative cycle by Bellman-Ford implies infeasibility of the specified timing and resource constraints.

6.3.6 Buffer Capacity Computation

This section discusses how the number of initial tokens on each queue is derived. This explains Step 4 of the approach that is outlined in Section 6.3.1. Given the start times of the first firings and the linear bounds on token production and consumption times, which are defined in these start times, a sufficient number of initial tokens is computed on each queue. Since start times are only computed when all constraints are satisfied, the resulting number of initial tokens on each queue is smaller than or equal to the constraint on maximum number of initial tokens on that queue. The computation of the number of initial tokens is is done by rewriting Equation (6.14) to Equation (6.16). We take as number of initial tokens the smallest integer value that satisfies the constraint in Equation (6.16).

$$\delta(e_{ij}) \geq \hat{\Gamma}(e_{ij}) - 1 + \hat{r}(e_{ij})\big(\hat{\Upsilon}(v_i) + s(v_i) - s(v_j)\big) \tag{6.16}$$

6.3.7 Computational Complexity

Overall our approach to compute the required number of tokens, which we show to correspond directly to the required buffer capacities, has a computational complexity that is exponential in the number of parameters and polynomial in the graph size.

More in detail, our approach has the following steps. We first have a number of checks on the validity of the VPDF graph, (1) strongly consistent, (2) strongly connected, (3) scoping of shared parameters, (4) repetition rates within scope of shared parameters, and (5) v_τ not in scope of shared parameter. The computational complexities of these checks is as follows, see (Cormen et al. 2001). To check for strong consistency, we solve a system of linear equations and have $O(E^3)$. To check that the graph is strongly connected we have $O(V + E)$. To check for proper scoping

of shared parameters and whether v_τ is not within the scope of a shared parameter, we have $O(V + E)$. To check for the repetition rates within scope of shared parameters, we solve a system of linear equation for each shared parameter, which results in $O(PE^3)$. Subsequently to these checks, (i) the maximum required transfer rates are determined, (ii) firing durations are checked, (iii) constraints on minimum differences in start times are determined, (iv) start times are computed, and (v) a sufficient number of tokens is determined. Determining the maximum required transfer rates and the check on firing durations both are exponential in the number of parameters, with complexities of $O(E2^P)$ and $O(V2^P)$, respectively. Determining the constraints requires determining the maximum cumulative consumption quantum and the maximum cumulative firing duration, which implies a complexity of $O(EK)$, with $K = \max(\{\theta(v_i)|v_i \in V\})$. Determining the number of tokens is linear in the number of edges, i.e. $O(E)$. Computing the start times can be done with Bellman-Ford, which has a complexity $O(VE)$. Therefore, the complexity of our complete algorithm is $O(V + VE + V2^P + EK + E^3 + PE^3 + E2^P)$.

6.3.8 Sufficiency of Buffer Capacities

In this section, we show that the computed number of initial tokens allows actor v_τ to fire with period τ for all possible parameter values. This is shown in two steps. To explain these steps, we define all parameters to be *local* except when parameter values are communicated to another actor. Parameters of which values are communicated to another actor are called *shared*. In the first step, we assume that all parameters that are shared by two actors, see Figure 6.24, are constant, i.e. can only attain a single value. With that assumption, Lemmas 6.5, 6.6, and 6.7 will help in establishing Theorem 6.3, which states that v_τ can execute wait-free for all possible values of local parameters. Subsequently, we show that this assumption can be removed. Lemmas 6.10, 6.11 help in establishing Theorem 6.4 that states that v_τ can execute wait-free, even if parameters are shared by two actors, given the number of initial tokens as computed in Section 6.3.6.

To compute the number of initial tokens, we have delayed start times of aggregate firings and used worst-case response times of code-segments. Theorem 6.5 shows that this is all allowed and that indeed the number of tokens computed for the VPDF graph as computed in Section 6.3.6 is a sufficient buffer capacity in the task graph that guarantees that the throughput constraint is satisfied. This is shown by using the result of Theorem 6.4, the one-to-one correspondence between tokens and containers in the dataflow graph and the task graph, and the fact that VPDF graphs have monotonic temporal behaviour.

We now start by introducing a number of concepts required in the later proofs. The number of tokens is computed with schedules of aggregate firings per queue. However, it is important to note that an actor can only

have a single schedule and not one schedule for each adjacent queue. The following definitions relate the individual schedules of the queues adjacent to an actor to the resulting schedule of that actor. Subsequently, we define shared parameters and define the assumption that shared parameters have constant values. This enables us to show in Section 6.3.8.1 that the number of initial tokens is sufficient in case there are no shared parameters, after which we show in Section 6.3.8.2 that also with shared parameters the computed number of initial tokens is sufficient for all parameter values.

Definition 6.2 *A queue-schedule is a function that given an aggregate firing f, with $f \in \mathbb{N}^*$, an actor v_i, and a queue e specifies the start time of the f-th aggregate firing of actor v_i when considering this queue with adjacent actors in isolation.*

Definition 6.3 *The start time of an aggregate firing in queue-schedule $\sigma(v_i, e)$ of actor v_i on queue e is given by Equation* (6.7) *in case e is an output queue of v_i and given by Equation* (6.11) *in case e is an input queue of v_i.*

Queue-schedule $\sigma(v_i, e)$ can be linearly bounded and the start time of the first firing is computed in Algorithm 6.1.

Definition 6.4 *For actor v_j, queue-schedule $\sigma(v_j, e_{ij})$ on input queue e_{ij} is valid, denoted by $\mathsf{valid}(\sigma(v_j, e_{ij}))$, if for every token x that is transferred over e_{ij} the production time of x resulting from queue-schedule $\sigma(v_i, e_{ij})$ is earlier than or equal to the consumption time of x resulting from queue-schedule $\sigma(v_j, e_{ij})$.*

Definition 6.5 *For two queue-schedules σ_1 and σ_2 that determine start times of an actor v_i on queue e, we have $\sigma_1 \leq \sigma_2$ if the start time of every firing f in σ_1 is not later than the start time of f in σ_2, i.e.*

$$\sigma_1 \leq \sigma_2 \Leftrightarrow \forall f \in \mathbb{N}^* \bullet s(f, \sigma_1) \leq s(f, \sigma_2) \tag{6.17}$$

In this work, the symbol $\bullet$ is used to separate the quantifiers from the proposition.

Definition 6.6 *An actor-schedule is a function that given an aggregate firing f, with $f \in \mathbb{N}^*$, of actor v_i specifies the start time of the f-th aggregate firing of actor v_i.*

Definition 6.7 *For every actor v_i, actor-schedule $\sigma(v_i)$ is given by taking for every firing the latest start time of v_i in the queue-schedules constructed on the queues adjacent to actor v_i.*

$$\sigma(v_i) = \max(\{\sigma(v_i, e) | e \in E(v_i)\}) \tag{6.18}$$

The schedule that is defined by Equation (6.18) is a schedule that is constructed independently of token arrival times. Definition 6.8 defines when this constructed schedule is valid with respect to token arrival times.

Definition 6.8 *For actor v_j, schedule $\sigma(v_j)$ is valid, denoted $\mathsf{valid}(\sigma(v_j))$, if we have that $\forall e_{ij} \in E(v_j) \bullet \mathsf{valid}(\sigma(v_j, e_{ij}))$.*

In Definition 6.4, a constructed schedule on an input queue is defined valid if token consumption times are not earlier than token production times. Definition 6.9, defines the schedule on an output queue of an actor to be valid if (1) the constructed schedule of the actor equals the schedule on this output queue and (2) on all input queues of this actor tokens arrive in time to enable the constructed schedule of this actor.

Definition 6.9 *For actor v_i and output queue e_{ij}, schedule $\sigma(v_i, e_{ij})$ is valid, denoted by $\mathsf{valid}(\sigma(v_i, e_{ij}))$, if $\mathsf{valid}(\sigma(v_i))$ and $\sigma(v_i) = \sigma(v_i, e_{ij})$.*

Lemmas 6.3 and 6.4 derive a difference between subsequent start times in the queue-schedules of the producer and consumer on a queue. These results are later used to compare queue-schedules and reason about equivalence of queue-schedules.

Lemma 6.3 *The difference between the start time of aggregate firing $f+1$ and the start time of aggregate firing f of actor v_i in queue-schedule $\sigma(v_i, e_{ij})$ is given by the ratio of the token production quantum of aggregate firing f and the maximum required rate on queue e_{ij}, i.e.*

$$s(f+1, \sigma(v_i, e_{ij})) - s(f, \sigma(v_i, e_{ij})) = \frac{\Pi(f, e_{ij})}{\hat{r}(e_{ij})} \tag{6.19}$$

Proof. Equation (6.7) says that

$$s(f+1, \sigma(v_i, e_{ij})) = s(v_i) + \frac{\Xi(f, e_{ij}) - \delta(e_{ij})}{\hat{r}(e_{ij})} \tag{6.20}$$

and

$$s(f, \sigma(v_i, e_{ij})) = s(v_i) + \frac{\Xi(f-1, e_{ij}) - \delta(e_{ij})}{\hat{r}(e_{ij})} \tag{6.21}$$

Subtracting Equation (6.21) from Equation (6.20) together with $\Xi(f, e_{ij}) - \Xi(f-1, e_{ij}) = \Pi(f, e_{ij})$ implies that this lemma holds. □

Lemma 6.4 *The difference between the start time of aggregate firing $f+1$ and the start time of aggregate firing f of actor v_j in queue-schedule $\sigma(v_j, e_{ij})$ is given by the ratio of the token consumption quantum of aggregate firing f and the maximum required rate on queue e_{ij}, i.e.*

$$s(f+1, \sigma(v_j, e_{ij})) - s(f, \sigma(v_j, e_{ij})) = \frac{\Gamma(f, e_{ij})}{\hat{r}(e_{ij})} \tag{6.22}$$

Proof. This proof is analogous to the proof of Lemma 6.3, however now starting from Equation (6.11). □

The set of parameters shared by two actors is given by Definition 6.10. This definition is used in Assumption 6.1 to be able to specify that shared parameters are assumed constant.

Definition 6.10 *A shared parameter is a parameter that is used by two actors. The set of shared parameters is given by*

$$S = \{p | \exists v_a, v_b \in V \bullet v_a \neq v_b \wedge p \in P(v_a) \cap P(v_b)\} \tag{6.23}$$

Assumption 6.1 *Any parameter p that is shared by two actors has a constant value $\hat{p}$ that equals the maximum value that p can attain, i.e. $\hat{p} = \max(\phi(p))$.*

$$\forall p \in S \bullet p = \hat{p} = \max(\phi(p)) \tag{6.24}$$

Under Assumption 6.1, only parameters that are local to an actor can attain different values. This means that for shared parameters as shown in Figure 6.24 it is assumed that they can only attain a single value, i.e. shared parameters are assumed to be constants. In the next section, we show that under this assumption the computed buffer capacities are sufficient. Subsequently, in Section 6.3.8.2, we show that Assumption 6.1 can be removed and that then still the computed buffer capacities are sufficient to guarantee that actor v_τ can execute wait-free.

6.3.8.1 Local Parameters

At the end of this section, Theorem 6.3 concludes that the number of initial tokens as computed by Equation (6.16) is sufficient given Assumption 6.1. Intuitively, our reasoning shows that schedules of actors that are closer to v_τ dominate schedules of actors that are further away from v_τ. These schedules dominate each other in the sense that delay is only pushed away from v_τ and that no actor can delay the schedule of an actor closer to v_τ. This in the end implies that no actor can delay the schedule of actor v_τ. This is proven by showing that every actor produces tokens on time on queues towards v_τ given that tokens arrive on time on queues away from v_τ. Lemma 6.5 shows that given that tokens arrive on time on all input queues for the queue-schedules, that tokens arrive on time for the actor-schedule. Lemma 6.6 shows that the actor-schedule equals the queue-schedule on queues towards v_τ. Given that tokens arrive on time, Lemmas 6.5 together with Lemma 6.6 says that on queues towards v_τ the queue-schedule of the token producing actor is valid. Lemma 6.7 shows that for any queue a valid queue-schedule by the token producing actor implies a valid queue-schedule by the token consuming actor. By repeated application of these lemmas, Theorem 6.3 is proven.

More in specific, for the example VPDF graph as shown in Figure 6.22 the reasoning is as follows. Note that it does not matter whether v_τ models an input or output interface. We first assume that on all queues away

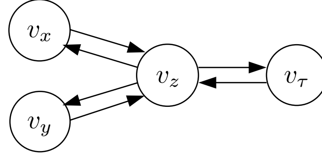


Figure 6.22: Example VPDF graph that is used to explain the outline of Section 6.3.8.1.

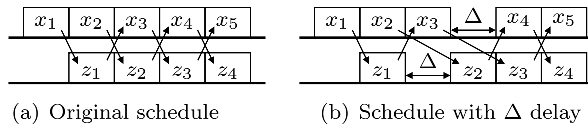


Figure 6.23: Illustration of linearity.

from actor v_τ tokens arrive on time. Given this assumption, we have that actors v_x and v_y produce their tokens on time. This implies that tokens arrive on time on all inputs of actor v_z, which implies that tokens arrive on time at actor v_τ. This is because the schedules constructed on the queues to and from actors v_x and v_y have such a transfer rate that they never lead to delayed start times of actor v_z. Depending on the consumption rate of v_τ, actor v_z can have delayed start times, which causes delayed start times of v_x and v_y. Because of linearity of VPDF graphs, actors v_x and v_y will not in turn delay v_z and v_z will not in turn delay v_τ. Therefore, a delay caused by v_τ will not delay v_τ.

In Figure 6.23, linearity and its consequences are illustrated. In Figure 6.23(a), we show an example schedule with five firings of actor v_x and four firings of actor v_z, where the arrows denote dependencies between firings. In Figure 6.23(b), the second firing of actor v_z is delayed by $\Delta \geq 0$. In this example, this causes a delay of Δ for the fourth firing of actor v_x. Even though the fourth firing of actor v_z depends on the fourth firing of actor v_x, which is delayed by Δ, we have that the fourth firing of actor v_z is not delayed by firings of actor v_x, because v_z already delayed its own firings by Δ.

Lemma 6.5 states that under Assumption 6.1, we have that for any actor that if tokens arrive in time for all its queue-schedules, then tokens also arrive in time for its actor-schedule. This is a non-trivial result, because the actor-schedule has later start times than the queue-schedules, which implies that it produces tokens later on its output queues. Lemma 6.5 shows that, even though the dataflow graph is strongly connected and thus has cycles, a $\Delta \geq 0$ later start time leads to a Δ later token production, which leads to

token arrival times on the input queues that are maximally delayed by Δ and therefore arrive on time for the subsequent firings of this actor, which are delayed by Δ. More specifically, in the graph of Figure 6.22, we have that the actor-schedule of v_z has later start times than its queue-schedules. We will later show that in fact the queue-schedule on queue $e_{z\tau}$ will delay the other queue-schedules. This lemma shows that, if on queue e_{xz} tokens arrive on time to sustain the non-delayed queue-schedule, then a delay Δ in the production time on e_{zx} will maximally lead to a delay Δ in production times on e_{xz}, leading to arrival times on e_{xz} that are on time for the queue-schedule of v_z on queue e_{xz}, which is also delayed by the actor-schedule of v_z, as illustrated by Figure 6.23.

Lemma 6.5 *Given Assumption 6.1, the following holds:*

$$\forall v_j \in V \bullet \big(\forall e_{ij} \in E(v_j) \bullet \mathsf{valid}(\sigma(v_j, e_{ij}))\big) \implies \mathsf{valid}(\sigma(v_j)) \qquad (6.25)$$

Proof. Equation (6.18) states that $\forall e \in E(v_j) \bullet \sigma(v_j) \geq \sigma(v_j, e)$, i.e. actor-schedule $\sigma(v_j)$ has token production times that are later than or equal to any of its queue-schedules $\sigma(v_j, e)$.

We will first show that every path from v_j back to itself starts with a queue e_1 and ends with a queue e_2 that have the same queue-schedule, i.e. $\sigma(v_j, e_1) = \sigma(v_j, e_2)$. Subsequently, we will show that a delay in token productions on e_1 cannot cause tokens to arrive too late on e_2, because both queue-schedules are delayed by the same amount.

Consistency tells us that every simple cycle has an even number of occurrences of a non-constant parameter p. Given Assumption 6.1, this means that every directed path from v_j to v_j that starts with a queue $e_1 \in E(p)$ ends with a queue $e_2 \in E(p)$ such that the ratio $\Pi(e_1)/\Gamma(e_2)$ is constant for all parameter values, i.e. $\exists \lambda \in \mathbb{Q} \bullet \Pi(e_1)/\Gamma(e_2) = \lambda$. We have that $\hat{r}(e_2) = \max_{\phi(P)}(\Gamma(e_2)\cdot z_j / z_\tau \cdot \tau)$ and $\hat{r}(e_1) = \max_{\phi(P)}(\Pi(e_1)\cdot z_j / z_\tau \cdot \tau)$. Substitution of $\lambda \cdot \Gamma(e_2)$ for $\Pi(e_1)$ in $\hat{r}(e_1)$ results in $\hat{r}(e_1) = \max_{\phi(P)}(\lambda \cdot \Gamma(e_2)\cdot z_j / z_\tau \cdot \tau) = \lambda \max_{\phi(P)}(\Gamma(e_2)\cdot z_j / z_\tau \cdot \tau) = \lambda \hat{r}(e_2)$. All schedules constructed for an actor have an equal start time for the first firing of this actor. Because, $\Pi(e_1) = \lambda \cdot \Gamma(e_2)$ and $\hat{r}(e_1) = \lambda \cdot \hat{r}(e_2)$, we have $\Pi(e_1)/\hat{r}(e_1) = \Gamma(e_2)/\hat{r}(e_2)$. Lemmas 6.3 and 6.4 tell that $\Pi(e_1)/\hat{r}(e_1) = \Gamma(e_2)/\hat{r}(e_2)$ means that in $\sigma(v_j, e_1)$ and $\sigma(v_j, e_2)$ the difference between subsequent firings of j is the same. Because the start time of the first firing in both schedules is the same, we have that both queue-schedules are the same, i.e. $\sigma(v_j, e_1) = \sigma(v_j, e_2)$. This also holds if p has a constant value.

Since a VPDF graph has linear temporal behaviour, a delay $\Delta = s(f, \sigma(v_j)) - s(f, \sigma(v_j, e_1)) \geq 0$ in token production times of firing f of v_j on queue e_1 leads to token arrival times on any corresponding queue e_2 that are maximally delayed by Δ. According to Definition 6.4, if $\mathsf{valid}(\sigma(v_j, e_2))$, then tokens arrive before the token consumption times that follow from schedule $\sigma(v_j, e_2)$. Because $\sigma(v_j, e_1) = \sigma(v_j, e_2)$, token production times following from schedule $\sigma(v_j, e_1)$ and token consumption times following

from schedule $\sigma(v_j, e_2)$ are delayed by Δ in the same firing. Since a delay Δ in $\sigma(v_j, e_1)$ causes tokens to arrive on e_2 maximally Δ later where $\sigma(v_j, e_2)$ is already equally delayed by Δ, tokens arrive in time to enable schedule $\sigma(v_j)$. Since this holds for all pairs e_1 and e_2 this implies that Equation (6.25) is true. □

Let $d(v_i)$ be the distance from actor v_i to actor v_τ be defined as the number of queues on the simple path from v_i to v_τ in the breadth-first tree as created by a breadth-first search from v_τ (Cormen et al. 2001). Lemma 6.6 establishes that queue-schedules on the queues adjacent to an actor v_i that are towards actors v_j and have $d(v_j) \leq d(v_i)$ determine the actor-schedule of v_i, in other words these queue-schedules dominate the other queue-schedules of an actor. For the example VPDF graph of Figure 6.22 this implies that the queue-schedule of v_z on $e_{z\tau}$ determines the actor-schedule of v_z.

Lemma 6.6 *Given Assumption 6.1, the following holds:*

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e_{ij} \in E(v_i) \bullet v_i \neq v_j \wedge d(v_j) \leq d(v_i) \implies \sigma(v_i) = \sigma(v_i, e_{ij}) \tag{6.26}$$

Proof. This follows from the proof of Lemma 6.10 that is presented in the next section, which includes shared parameters. □

Lemma 6.7 states that if the queue-schedule of v_i on a queue e_{ij} equals the actor-schedule of v_i, then on queue e_{ij} tokens arrive before they are needed by v_j according to the queue-schedule of v_j on queue e_{ij}. More specifically, in Figure 6.22, we have that $\mathsf{valid}(\sigma(v_z, e_{z\tau}))$ if $\mathsf{valid}(\sigma(v_z, e_{xz}))$ and $\mathsf{valid}(\sigma(v_z, e_{yz}))$ and $\mathsf{valid}(\sigma(v_z, e_{\tau z}))$ and $\sigma(v_z) = \sigma(v_z, e_{z\tau})$. This means that the schedule on the queue from v_z to v_τ is called valid, if tokens arrive in time on all input queues and the schedule on this output queue determines the actor-schedule of v_z. If this is true, then we know that on queue $e_{z\tau}$ the linear upper bound on token production times is a conservative bound.

Lemma 6.7 *The following holds.*

$$\forall e_{ij} \in E \bullet \mathit{valid}(\sigma(v_i, e_{ij})) \implies \mathit{valid}(\sigma(v_j, e_{ij})) \tag{6.27}$$

Proof. According to Definition 6.9, we have that $\mathsf{valid}(\sigma(v_i, e_{ij}))$ is true iff $\mathsf{valid}(\sigma(v_i))$ and $\sigma(v_i, e_{ij}) = \sigma(v_i)$. This means that $\mathsf{valid}(\sigma(v_i, e_{ij}))$ is true if on all input queues of v_i tokens arrive before they are consumed, and that the actor-schedule of actor v_i is equal to the queue-schedule of v_i on queue e_{ij}. This implies that token production times on e_{ij} are conservatively bounded by the linear upper bound on production times. Furthermore token consumption times in schedule $\sigma(v_j, e_{ij})$ are conservatively

bounded by the linear lower bound on consumption times. Since these bounds are taken into account when computing start times for schedules $\sigma(v_i, e_{ij})$ and $\sigma(v_j, e_{ij})$ in Algorithm 6.1, Equation (6.27) holds. □

In Theorem 6.3 below, the graph is traversed from actors that are furthest away from v_τ to actors that are closer to v_τ. For any actor v_j, we show that if tokens arrive on time on all input queues of v_j from actors that are closer to v_τ, i.e. Assumption 6.2 below holds, and tokens arrive on time on all input queues of v_j from actors further away from v_τ, then this actor produces its tokens on time on all output queues to actors closer to v_τ. As we traverse the graph, Assumption 6.2 applies to ever fewer actors until it only applies to v_τ. The proof is concluded by showing that in fact this assumption holds for v_τ. Assumption 6.2 enables the proof of Theorem 6.3 to be a proof by induction over the distance to v_τ, i.e. Theorem 6.3 is true independent of the truth of Assumption 6.2.

Assumption 6.2 *For actor v_j holds that on all input queues e_{ij} with $d(v_i) \leq d(v_j)$ holds that $\mathsf{valid}(\sigma(v_j, e_{ij}))$.*

Theorem 6.3 *Given Assumption 6.1, then the number of initial tokens as computed by Equation* (6.16) *is sufficient to let v_τ execute wait-free.*

Proof. The proof is by structural induction over a breadth-first tree that has actor v_τ as its root.

Base step. Let actor v_i be a leaf of this tree, then given Assumption 6.2 it holds that $\sigma(v_i)$ is valid. This implies that, on any output queue e of actor v_i, we have $\mathsf{valid}(\sigma(v_i, e))$. This is because Lemma 6.6 states $\sigma(v_i) = \sigma(v_i, e)$.

Induction step. For any actor v_j, if (1) on all queues e_{kj} from actors v_k, with $d(v_k) > d(v_j)$, it holds that $\mathsf{valid}(\sigma(v_j, e_{kj}))$, and (2) on all queues e_{hj} from actors v_h, with $d(v_h) \leq d(v_j)$, it holds that $\mathsf{valid}(\sigma(v_h, e_{hj}))$, then $\mathsf{valid}(\sigma(v_j))$ holds. With $\mathsf{valid}(\sigma(v_j))$ true, we have, by Lemma 6.6, that on all queues e_{jl}, with $d(v_l) \leq d(v_j)$, $\mathsf{valid}(\sigma(v_j, e_{jl}))$.

Together with Lemma 6.7, this means that starting from the leaves of the breadth-first tree, we can traverse the tree in a breadth-first manner back to v_τ to reach the conclusion that $\mathsf{valid}(\sigma(v_\tau))$ given Assumption 6.2.

However, for actor v_τ Assumption 6.2 holds by construction. This is because there are no actors with a smaller than or equal distance, which implies that we only need to check queues from v_τ to itself. We have that $\forall e \in E(v_\tau) \bullet \sigma(v_\tau, e) \leq \sigma(v_\tau)$. Any queue $e_{\tau\tau}$ from v_τ to itself needs, by consistency, to have the same token production and consumption quantum. This implies that the schedule of the token producer, i.e. $\sigma(v_\tau, e_{\tau\tau})$ and the schedule of the token consumer, i.e. $\sigma(v_\tau, e_{\tau\tau})$, are delayed by the same delay $\Delta \geq 0$, which implies that tokens arrive in time. □

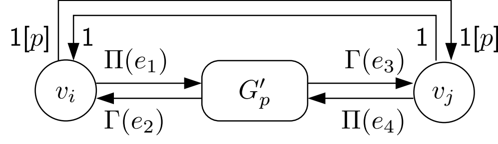


Figure 6.24: VPDF graph with shared parameter p, where $\Pi(e_1)$, $\Gamma(e_2)$, $\Gamma(e_3)$, and $\Pi(e_4)$ are parameterised in p.

6.3.8.2 Shared Parameters

In this section, Theorem 6.4 shows that Theorem 6.3 also holds without Assumption 6.1. This is done by adding a clause to the premise of Lemma 6.6 enabling us to remove Assumption 6.1, resulting in Lemma 6.10. The proof of Lemma 6.10 is based on the results of Lemmas 6.8 and 6.9, which will also support the proof of Lemma 6.14. Subsequently, Lemma 6.11 shows that given a valid actor-schedule the removal of Assumption 6.1 does not affect the validity of this actor-schedule. These two lemmas are used in the proof of Theorem 6.4 to show that parts of a VPDF graph cannot influence the schedule of v_τ. We show that for example, for the graph shown in Figure 6.24, variation in the value of parameter p does not influence the actor-schedules of actors v_i and v_j and since, by construction of VPDF graphs, actor v_τ is not in G'_p variation in p does not influence the schedule of v_τ. This implies that a VPDF graph can be reduced, in Figure 6.24 through removal of G'_p, to a VPDF graph for which Assumption 6.1 holds, implying that Theorem 6.3 holds.

Lemmas 6.8 and 6.9 enable us to show in Lemma 6.14 that the queue-schedule on queues from an actor v_i that satisfy the following property determine the actor-schedule of this actor v_i.

Definition 6.11 *The property $D(e_{ij})$ is true if and only if queue e_{ij} is from v_i to v_j where these actors are different and actor v_j has a smaller than or equal distance to v_τ than actor v_i and the parameterised cumulative production quantum on e_{ij} is not parameterised in a shared parameter $p \in S$.*

$$D(e_{ij}) \Leftrightarrow v_i \neq v_j \wedge d(v_j) \leq d(v_i) \wedge (\nexists p \in S \bullet \Pi(e_{ij}) \approx p)$$

Lemma 6.8 *The following holds:*

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e_{ij}, e_{ik} \in E(v_i) \bullet D(e_{ij}) \implies \frac{\hat{r}(e_{ik})}{\hat{r}(e_{ij})} = \max_{\phi(P)} \left(\frac{\Pi(e_{ik})}{\Pi(e_{ij})} \right) \tag{6.28}$$

where e_{ik} is an output queue of v_i.

Proof. By definition, $\hat{r}(e_{ik}) = \max_{\phi(P)} \left({}^{\Pi(e_{ik})\cdot z_i}/{}_{z_\tau \cdot \tau}\right)$. By Lemma 6.1, we have ${}^{z_i}/{}_{z_j} = {}^{\Gamma(e_{ij})}/{}_{\Pi(e_{ij})}$. After substitution of ${}^{\Gamma(e_{ij})\cdot z_j}/{}_{\Pi(e_{ij})}$ for z_i in $\max_{\phi(P)} \left({}^{\Pi(e_{ik})\cdot z_i}/{}_{z_\tau \cdot \tau}\right)$ we have $\hat{r}(e_{ik}) = \max_{\phi(P)} \left({}^{\Pi(e_{ik})\cdot \Gamma(e_{ij})\cdot z_j}/{}_{\Pi(e_{ij})\cdot z_\tau \cdot \tau}\right)$.

We have that $\Pi(e_{ik})$ and $\Pi(e_{ij})$ are parameterised in different parameters than ${}^{z_j}/{}_{z_\tau}$. This is because, by Lemma 6.1, ${}^{z_j}/{}_{z_\tau}$ is equal to a ratio of cumulative transfer quanta on a directed simple path from v_j to v_τ. Because $v_i \neq v_j$ and because of the restrictions on sharing of parameters, none of these cumulative transfer quanta is parameterised in a parameter of $\Pi(e_{ik})$ or $\Pi(e_{ij})$.

Furthermore, we have that $\Pi(e_{ik})$ and $\Pi(e_{ij})$ are parameterised in different parameters than τ, because v_τ is not adjacent to a queue that communicates a parameter value.

Also $\Pi(e_{ik})$ and $\Pi(e_{ij})$ do not share a parameter with $\Gamma(e_{ij})$. For $\Pi(e_{ij})$ this is by definition of this queue. For $\Pi(e_{ik})$, suppose that $\Pi(e_{ik})$ shares a parameter with $\Gamma(e_{ij})$. Then, by the restrictions on sharing of parameters, $z_i = z_j$, and because $\Pi(e_{ij}) \cdot z_i = \Gamma(e_{ij}) \cdot z_j$ holds by consistency of the graph, we now have that $\Pi(e_{ij}) = \Gamma(e_{ij})$. However, by definition of e_{ij}, $\Pi(e_{ij})$ is not parameterised in a shared parameter and therefore $\Gamma(e_{ij})$ cannot share a parameter with $\Pi(e_{ik})$.

Because $\Pi(e_{ij})$ and $\Pi(e_{ik})$ are parameterised in different parameters than ${}^{\Gamma(e_{ij})\cdot z_j}/{}_{z_\tau \cdot \tau}$, we have $\hat{r}(e_{ik}) = \max_{\phi(P)} \left({}^{\Pi(e_{ik})\cdot \Gamma(e_{ij})\cdot z_j}/{}_{\Pi(e_{ij})\cdot z_\tau \cdot \tau}\right) = \max_{\phi(P)} \left({}^{\Pi(e_{ik})}/{}_{\Pi(e_{ij})}\right) \cdot \max_{\phi(P)} \left({}^{\Gamma(e_{ij})\cdot z_j}/{}_{z_\tau \cdot \tau}\right)$, because we have $\hat{r}(e_{ij}) = \max_{\phi(P)} \left({}^{\Gamma(e_{ij})\cdot z_j}/{}_{z_\tau \cdot \tau}\right)$, this equals $\max_{\phi(P)} \left({}^{\Pi(e_{ik})}/{}_{\Pi(e_{ij})}\right) \cdot \hat{r}(e_{ij})$. □

Lemma 6.9 *The following holds:*

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e_{ij}, e_{hi} \in E(v_i) \bullet D(e_{ij}) \implies \frac{\hat{r}(e_{hi})}{\hat{r}(e_{ij})} = \max_{\phi(P)} \left(\frac{\Gamma(e_{hi})}{\Pi(e_{ij})} \right) \tag{6.29}$$

where e_{hi} is an input queue of v_i.

Proof. This proof is analogous to the proof of Lemma 6.8. □

Lemma 6.10 states that the queue-schedules on the queues towards actor v_τ that are without shared parameters dominate the other queue-schedules of an actor.

Lemma 6.10 *The following holds:*

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e_{ij} \in E(v_i) \bullet D(e_{ij}) \implies \sigma(v_i) = \sigma(v_i, e_{ij}) \tag{6.30}$$

Proof. Let e_{ik} be an output queue of v_i and let e_{hi} be an input queue of v_i. In order to show that the queue-schedule of e_{ij} determines the actor-schedule of v_i, we need to show that the difference between subsequent start times of the queue-schedule on e_{ij} is for every firing larger than or

equal to the difference in subsequent start times in the queue-schedules on e_{ik} and e_{hi}. According to Lemmas 6.3 and 6.4 this requires to show that ${}^{\Pi(e_{ij})}/_{\hat{r}(e_{ij})} \geq {}^{\Pi(e_{ik})}/_{\hat{r}(e_{ik})}$ and ${}^{\Pi(e_{ij})}/_{\hat{r}(e_{ij})} \geq {}^{\Gamma(e_{hi})}/_{\hat{r}(e_{hi})}$, or, equivalently, we need to show that ${}^{\hat{r}(e_{ik})}/_{\hat{r}(e_{ij})} \geq {}^{\Pi(e_{ik})}/_{\Pi(e_{ij})}$ and ${}^{\hat{r}(e_{hi})}/_{\hat{r}(e_{ij})} \geq {}^{\Gamma(e_{hi})}/_{\Pi(e_{ij})}$. By Lemmas 6.8 and 6.9, respectively, these conditions are satisfied. □

Lemma 6.11 shows that if the actor-schedule is valid for the maximum value of a shared parameter, the validity of this actor-schedule is independent of the value of this shared parameter.

Lemma 6.11 *Given a parameter $p \in S$. If for each $v_i \in V$ with $p \in P(v_i)$ holds that $\mathsf{valid}(\sigma(v_i))$ for $\hat{p}$, then $\mathsf{valid}(\sigma(v_i))$ for all values of p.*

Proof. The fact $p \in S$ implies that for every v_i with $p \in P(v_i)$ there is an $v_j \neq v_i$ with $p \in P(v_j)$. Furthermore, every path from v_i to v_j that starts with an output queue e_1 of v_i that has $\Pi(e_1)$ parameterised in p includes an input queue e_2 of v_j that has $\Gamma(e_2)$ that is parameterised in p. By consistency of the graph, we have that $\exists \lambda \in \mathbb{Q} \bullet \Pi(e_1) = \lambda \cdot \Gamma(e_2)$. Lemmas 6.3 and 6.4 tell us that $\sigma(v_i, e_1) = \sigma(v_j, e_2)$.

Since every actor has a queue that is towards v_τ, Lemma 6.10 tells us that $\sigma(v_i)$ has later start times than queue-schedule $\sigma(v_i, e_1)$ in case of $\hat{p}$ by $\hat{\Delta}_i \geq 0$ and that $\sigma(v_j)$ has later start times than the queue-schedule $\sigma(v_j, e_2)$ in case of $\hat{p}$ by $\hat{\Delta}_j \geq 0$. For smaller parameter values than $\hat{p}$ these queue-schedules have a smaller difference in subsequent start times by Lemmas 6.3 and 6.4 and therefore have a larger difference in the start time of a particular firing compared to the start time in the actor-schedule. Since these queue-schedules are the same, this additional difference with the actor-schedule $\Delta \geq 0$ is the same for both schedules. Given that $\mathsf{valid}(\sigma(v_i))$ and $\mathsf{valid}(\sigma(v_j))$ for $\hat{p}$, linearity of VPDF graphs tells us that a Δ later token production on e_1 cannot lead to token arrival times that are delayed by more than Δ. Therefore, tokens arrive on time on queue e_2, because the consumption time according to the actor-schedule of v_j on e_2 was also delayed by Δ compared to the consumption time according to the queue-schedule of v_j on e_2. □

For any shared parameter p. The following theorem shows that the schedules of the actors in the sub-graph G'_p, see for example Figure 6.24, do not influence the schedule of v_τ. This implies that these subgraphs can be removed from the graph, resulting in a graph that does not have shared parameters. This implies that the assumption under which Theorem 6.3 holds is valid.

Theorem 6.4 *The number of initial tokens that is computed by Equation* (6.16) *is sufficient to let v_τ execute wait-free.*

Proof Theorem 6.3 shows that this theorem is true given Assumption 6.1. However, Lemma 6.10 states that for each actor the actor-schedule equals the queue-schedule on a queue towards v_τ that has a cumulative token production quantum that is not parameterised in a shared parameter. By construction of VPDF graphs, each actor has such a queue. Further, Lemma 6.11 states that the value of shared parameters does not affect the validity of the actor-schedules of the actors that have cumulative token transfer quanta parameterised in these shared parameters. Together this implies that for every actor with cumulative transfer quanta parameterised in shared parameters that the queue-schedules on these queues do not influence the actor-schedule. For every shared parameter, we can remove all queues with cumulative transfer quanta parameterised in this parameters. This disconnects a subgraph from the VPDF graph. By construction of VPDF graphs, this subgraph does not include v_τ, and we have just established that it does not affect the actor-schedules of the actors in the remaining graph. After disconnecting all these subgraphs, we have a remaining VPDF graph that does not have shared parameters. This means that Assumption 6.1 is irrelevant for the remaining VPDF graph that still includes v_τ. Since removal of these subgraphs did not change the actor-schedules of the remaining VPDF graph, this theorem holds because Theorem 6.3 holds. □

6.3.8.3 Computed Number of Tokens and Required Buffer Capacity

Theorem 6.4 established that the computed number of tokens is sufficient to let actor v_τ execute wait-free. The following theorem shows that this number of initial tokens corresponds with the required buffer capacity to satisfy the throughput constraint.

Theorem 6.5 *The number of initial tokens that is computed by Equation* (6.16) *is a sufficient buffer capacity to guarantee that the throughput constraint is satisfied.*

Proof. The VPDF graph is required to be temporally conservative to the task graph by Theorem 5.2. Therefore, actor firings are only enabled later and produce tokens later than the corresponding sequence of task executions are enabled and produce their containers.

Aggregate firings are only enabled later and produce tokens later then normal firings. By monotonicity of VPDF graphs, we have the following. A VPDF graph with normal firings does not have later token arrival times than a VPDF graph with aggregate firings. The task graph does not have later container arrival times than the token arrival times in the VPDF graph with normal firings. This implies that token arrival times in the VPDF graph with aggregate firings are conservative container arrival times. Since task executions are enabled by container arrivals, if the number of

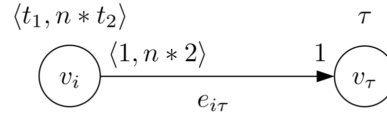


Figure 6.25: VPDF graph with $n \in [0, \infty)$, i.e. the second phase has an unbounded number of firings.

tokens is sufficient to let v_τ execute wait-free, then the corresponding buffer capacities are sufficient to let the sink, or source, of the task graph execute wait-free. Therefore, by Theorem 6.4 this theorem holds. □

6.4 Unbounded Iteration

From the definition of an aggregate firing in Section 6.3.2 it is clear that the production time of tokens depends on the number of iterations, i.e. the more iterations the later an aggregate firing produces its tokens. An unbounded number of iterations leads with this approach to an unbounded production time and in the end to unbounded buffer capacities. Apart from a small change in the determination of the maximum required rate and feasibility, this section proposes a different definition of aggregate firings and discusses its consequences. The proposed definition of aggregate firings allows for unbounded firing parameters. An example graph with unbounded firing parameters is shown in Figure 6.25. In this figure and subsequent figures in this section, we omitted the queue from actor v_τ to actor v_i for reasons of clarity.

6.4.1 Outline of the Approach

In Section 6.3.2, we defined a rectangular aggregate firing. This rectangular aggregate firing abstracts a sequence of normal firings and consumes all tokens consumed by this sequence of normal firings at its start and produces all tokens produced by this sequence of normal firings at its finish. As remarked upon in Section 6.3.2 this abstraction leads to a loss in accuracy. However, when we want to analyse VPDF graphs with an unbounded number of firings per phase, then this abstraction is no longer applicable, because it leads to the requirement of unbounded buffer capacities. Figure 6.25 shows an example VPDF graph with an actor that has a phase with an unbounded number of firings.

In Figure 6.26(a), we show a sequence of four normal firings of actor v_i from Figure 6.25, where v_i has a firing duration of $t_1 = \tau$ for its first phase and a firing duration of $t_2 = 2\tau$ for firings of its second phase. The rectangular aggregate firing is not applicable for this actor, because the

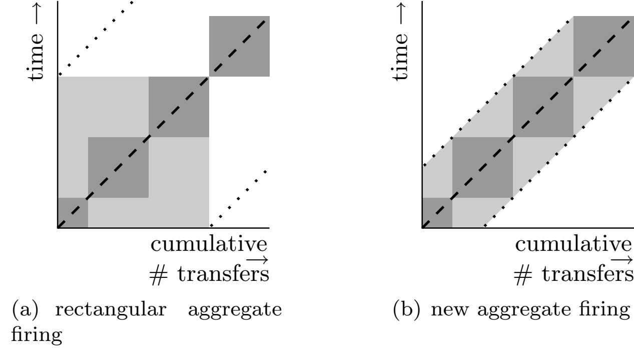


Figure 6.26: Schedule of firings of actor v_i from Figure 6.25, with $t_1 = \tau$ and $t_2 = 2\tau$. The aggregate firing is depicted with light grey.

production time of the first token is increased with an increasing number of firings of the second phase of actor v_i. Figure 6.26(a) shows, as an example, the aggregate firing corresponding with two firings of the second phase and the upper bound on token production times and lower bound on token consumption times that correspond with this rectangular aggregate firing. In this example, we have that with rectangular aggregate firings, the distance between these two bounds increases with an increasing number of firings of the second phase. In this section, we redefine aggregate firings such that they no longer have a rectangular token transfer shape but instead a shape as shown in Figure 6.26(b). This new shape grows along the required token transfer rate, resulting in a difference between the upper bound on token production times and a lower bound on token consumption times that no longer grows with the number of firings, in case there is no upper bound specified on this number of firings.

In Figure 6.26, we constructed a schedule of normal firings in which every firing starts immediately after the previous firing finished. This is the schedule of firings that was used to construct the rectangular aggregrate firings in Section 6.3.2. In Section 6.3.4, we delayed aggregate firings to let the token transfer rate of the constructed schedule match the required token transfer rate. However, if we construct this schedule of normal firings, then an unbounded number of firings of a phase can lead to an unbounded difference between the token transfer rate of this constructed schedule and the required token transfer rate. The VPDF graph as shown in Figure 6.25 with $t_1 = \tau$ and $t_2 = 3\tau$ is already rejected by our check on the firing durations of actors, because with these firing durations actor v_i from Figure 6.25 cannot transfer tokens at a sufficient rate, as illustrated by the schedule in Figure 6.27.

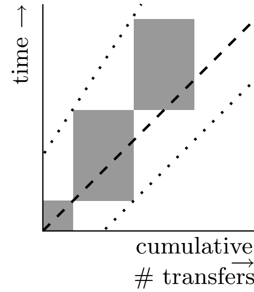


Figure 6.27: Schedule of firings of actor v_i from Figure 6.25, with $t_1 = \tau$ and $t_2 = 3\tau$.

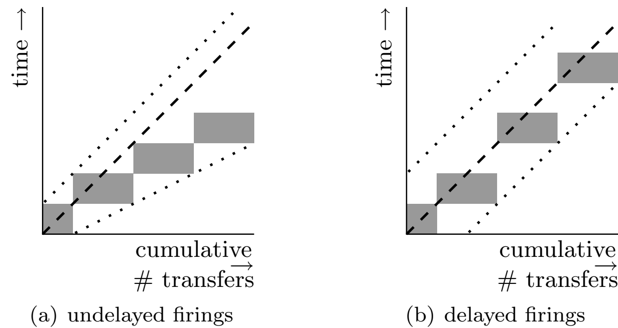


Figure 6.28: Schedule of firings of actor v_i from Figure 6.25, with $t_1 = \tau$ and $t_2 = \tau$.

However, the VPDF graph as shown in Figure 6.25 with $t_1 = \tau$ and $t_2 = \tau$ is allowed. The schedule in Figure 6.28(a) starts a firing immediately after the previous has finished, which in this case results in a token transfer rate that is higher than the required rate. This schedule will result in an unbounded required buffer capacity.

The schedule of normal firings that we will construct in this section will delay firings of phases with no upper bound on the number of firings to adapt the schedule of these firings to the required token transfer rate. Our adaptation of the schedule from Figure 6.28(a) results in the schedule as shown in Figure 6.28(b), in which the second and third firing of the second phase of v_i from Figure 6.25 are delayed.

However, the introduction of delay in the schedule of normal firings is non-trivial. This is because the firing durations of different phases can compensate each other, as for example actor v_i in Figure 6.29. This actor

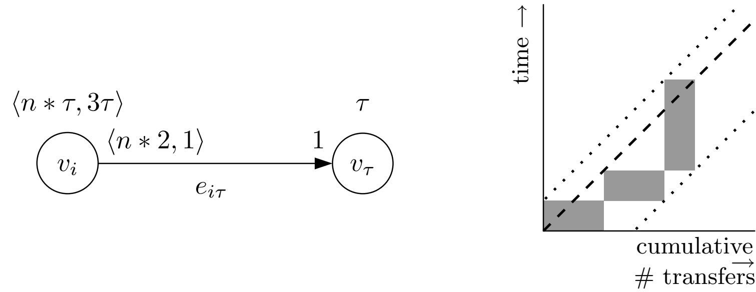
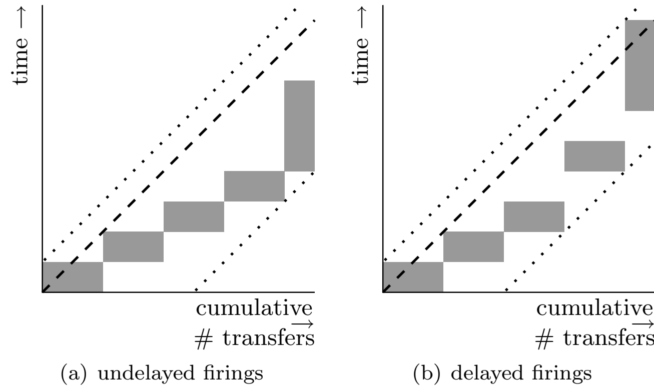


Figure 6.29: VPDF graph with $n \in [2, \infty)$ and a schedule of firings of actor v_i for $n = 2$.



(a) undelayed firings (b) delayed firings

Figure 6.30: Schedules for firings of actor v_i from Figure 6.29, with $n = 4$.

v_i requires n to have a lower bound of 2 in order to transfer tokens at a sufficient rate. The first two firings of the first phase of this actor cannot be delayed, otherwise the rate would be insufficient. However, as shown in Figure 6.30(a), if subsequent firings of the first phase are not delayed than the rate will diverge from the required token transfer rate. In Figure 6.30(b), it is shown where we introduce the required delay to match the token transfer rate of the constructed schedule to the required token transfer rate.

The basic idea is that the schedule of firings within an aggregate firing is constructed by starting with a reference schedule in which the phases with no upper bound on their number of firings have a minimal number of firings. In this reference schedule, we insert the omitted firings of phases with no upper bound on their number of firings. The insertion of a firing has as

its consequence that the subsequent firings in this schedule are translated over the maximum required token transfer rate. This process is illustrated by Figures 6.29(b) and 6.30(b), where the schedule of firings shown in Figure 6.29(b) forms the reference schedule. In the schedule of Figure 6.30(b), two additional firings of the first phase have been inserted and the firing of the second phase is translated over the maximum required transfer rate compared to its position in the token transfer curve of Figure 6.29(b).

In Section 6.3, schedules of aggregate firings were determined per queue. The schedule of firings within aggregate firings was the same on every adjacent queue, i.e. a firing within an aggregate firing always started immediately after the previous firing within the same aggregate firing finished. The proof of Theorem 6.5 depends on lemmas that show that combining these schedules of aggregate firings per queue to a schedule of aggregate firings per actor leads to the situation that the schedules on queues towards v_τ determine the schedule of the actor. This means that the start times of aggregate firings in the constructed schedules on the queues towards v_τ are later than the start times in the constructed schedules on other adjacent queues. The two main differences between the approach in this section and our approach of Section 6.3 are that we no longer abstract firings to rectangular aggregate firings and that we now introduce delay in the schedule of firings within an aggregate firing. The first main difference implies that we need to derive new bounds on token transfer times given the new aggregate firings. The second main difference implies that we need to extend our previous reasoning and show that the schedule of firings in queue-schedules of queues towards v_τ has later start times than in queue-schedules of other adjacent queues. This is because we delay firings such that the schedule of firings has a transfer rate that matches the maximum required transfer rate. Therefore, on different adjacent queues firings are delayed by different amounts. We will show that firings are delayed most in the constructed queue-schedules of queues towards v_τ.

In this section, we first adapt the determination of the maximum required rate in Section 6.4.2. Subsequently, we present the new definition of aggregate firings in Section 6.4.3. Given this definition of aggregate firings, we derive new expressions for the linear bounds on token transfer times in Section 6.4.4. Section 6.4.5 shows that the number of initial tokens computed in Section 6.4.4 corresponds with buffer capacities that are sufficient to let task w_τ execute wait-free, i.e. Section 6.4.5 shows that Theorem 6.5 still holds.

6.4.2 Maximum Required Token Transfer Rate

The determination of the maximum required rate as described by Theorem 6.1 in Section 6.3.3 is found by considering all combinations of extreme values that parameters can attain. In case a parameter does not have an upper bound, then there is no maximum value. We adapt the procedure

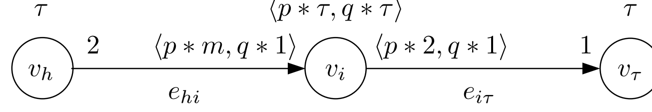


Figure 6.31: Example VPDF graph, with $p, q \in [1, \infty)$ and $m \in [1, 3]$.

from Theorem 6.1 as follows. For a queue e_{ij}, we consider all combinations of extreme values of the parameters in $\Pi(e_{ij}) \cdot z_i / z_\tau \cdot \tau$, except that for parameters with no upper bound we take the limit to positive infinity. This limit can be determined relatively straightforwardly. This is because parameters with no upper bound are local to a single actor, while further in Lemma 6.2 it is shown that $\Pi(e_{ij}) \cdot z_i / z_\tau \cdot \tau$ equals a product of factors, where each factor is a ratio containing only cumulative transfer quanta and cumulative firing duration of a single actor. This implies that the limit to positive infinity of multiple parameters that are associated with different actors is the product of the limit to positive infinity of the ratio in this product that corresponds with this actor. Because of the sequential nature of the phases of an actor, we do not need to consider the limit to positive infinity of multiple parameters of the same actor.

For example, for the VPDF graph from Figure 6.31, the maximum required transfer rates are determined as follows. We have symbolic repetition rates $z_h = p \cdot m + q$, $z_i = 2$, and $z_\tau = 4p + 2q$. On queue e_{hi}, we have $\Pi(e_{hi}) \cdot z_h / z_\tau \cdot \tau = 2 \cdot (p \cdot m + q) / (4p + 2q) \cdot \tau = p \cdot m + q / (2p + q) \cdot \tau$. Since p and q have no upper bound and belong to the same actor, we consider the following combinations of extreme values: (1) m, p, q minimal, (2) m minimal and limit to positive infinity in p, (3) m minimal and limit to positive infinity in q, (4) m maximal and p, q minimal, (5) m maximal and limit to positive infinity in p, (6) m maximal and limit to positive infinity in q. Taking the minimal value for p and q, we obtain $1 \cdot m + 1 / (2 \cdot 1 + 1) \cdot \tau = m + 1 / 3\tau$, which means case (1) equals a rate of $2/3\tau$ and case (4) equals a rate of $4/3\tau$. Evaluating $\lim_{p \to \infty} p \cdot m + q / (2p + q) \cdot \tau$ and $\lim_{p \to \infty} p \cdot m + q / (2p + q) \cdot \tau$, we obtain $m/2\tau$ and $1/\tau$, respectively. This implies that case (2) has a rate of $1/2\tau$, case (5) has a rate of $3/2\tau$, and cases (3) and (6) have a rate of $1/\tau$. The maximum required rate on queue e_{hi} is, therefore, $\hat{r}(e_{hi}) = \max(\{2/3\tau, 4/3\tau, 1/2\tau, 3/2\tau, 1/\tau\}) = 3/2\tau$. The maximum required transfer rate on $e_{i\tau}$ is $\hat{r}(e_{i\tau}) = 1/\tau$.

The procedure from Theorem 6.2 to verify that the firing durations enable the satisfaction of the throughput constraint is extended similarly. For actor v_h, we have that $\max_{\phi(P)}(\Upsilon(v_h) \cdot z_h / z_\tau \cdot \tau) \leq 1$ holds. This is because we have that $\max_{\phi(P)}(\Upsilon(v_h) \cdot z_h / z_\tau \cdot \tau) = \max_{\phi(P)}(\tau \cdot (p \cdot m + q) / (4p + 2q) \cdot \tau) = \max_{\phi(P)}(p \cdot m + q / 4p + 2q) = \max(\{1/3, 1/4, 1/2, 2/3, 3/4\}) = 3/4 \leq 1$.

6.4.3 Aggregate Firing

As outlined in Section 6.4.1, our basic idea is to have aggregate firings that grow along the maximum required token transfer rate line instead of having rectangular aggregate firings. The schedule of firings within these new aggregate firings is constructed by first considering a reference schedule and then inserting additional firings. The reference schedule only schedules a minimum number of firings of the phases that have no upper bound on their number of firings. Say that the parameters of actor v_i in aggregate firing f have parameter values $P(v_i, f)$. Given the values $P(v_i, f)$, we derive the parameter values $P_0(v_i, f)$ in which the parameters that specify the number of firings of phases that have no upper bound on their number of firings obtain their minimal value. Our check on the firing durations of an actor ensures that for all possible parameter values the firing duration allows the throughput constraint to be satisfied. For the phases that can have an unbounded number of firings, this implies that these phases have a token transfer rate that is higher than or equal to the maximum required token transfer rate.

Consider, for example, aggregate firing f of actor v_i from Figure 6.29 with $n = 4$. To obtain $P_0(v_i, f)$, we minimise the number of firings of the first phase since this phase does not have an upper bound on its number of firings. Therefore, in our reference parameter valuation $P_0(v_i, f)$, we have that $n = 2$. Given this reference parameter valuation, we construct a reference schedule in which each firing in the aggregate firing starts immediately after the previous firing in this aggregate firing has finished. This is the schedule shown in Figure 6.29. To obtain the schedule of firings within aggregate firing f, we insert firings in the reference schedule. We insert firings in this reference schedule that are present given parameter values $P(v_i, f)$, but that are not present given parameter values $P_0(v_i, f)$. In the reference schedule as shown in Figure 6.29, we insert the third firing of the first phase, which is present given $P(v_i, f)$ in which $n = 3$, but not present given $P_0(v_i, f)$ in which $n = 2$. Because we insert a firing that has a token transfer rate that is higher than the maximum required token transfer rate, we can insert this firing and translate subsequent firings over the maximum required transfer rate line, because this will only delay firings. This is shown in Figure 6.30(b), where the third firing of the first phase is inserted and the subsequent firing is translated over the maximum required transfer rate line compared to its position in the reference schedule from Figure 6.29. In the resulting schedule, as shown in Figure 6.30(b), no longer every firing starts immediately after the previous has finished. Instead delay is introduced in the schedule of firings within an aggregate firing.

To illustrate the construction of the schedules more precisely, our running example is actor v_i from Figure 6.31 with parameter values $P(v_i, f)$ equal to $p = 3$, $q = 2$, and $m = 2$. Since p and q could have no upper

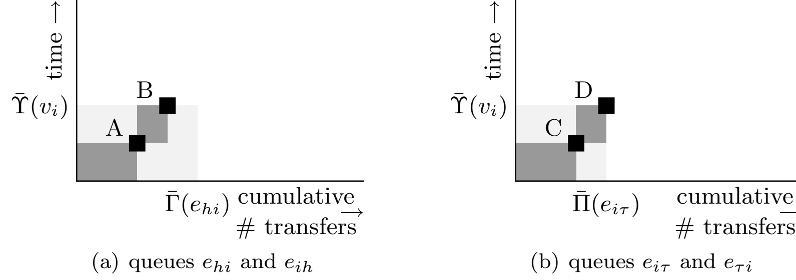


Figure 6.32: Schedule of firings for parameter valuation $P_0(v_i, f)$.

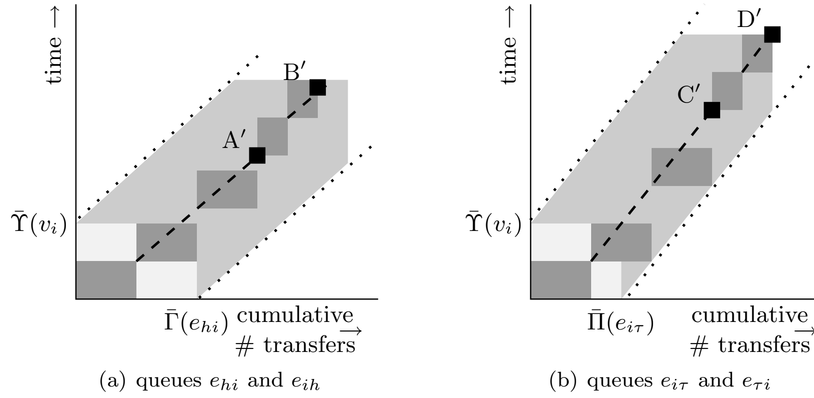


Figure 6.33: Increased number of transferred tokens leads to start times that are shifted along required rate.

bound, we take the minimal value of p and q, which results in $P_0(v_i, f)$ equal to $p = 1$, $q = 1$, and $m = 2$. The reference schedule on queues e_{hi} and $e_{i\tau}$ as constructed using $P_0(v_i, f)$ is shown in Figure 6.32, while the schedule for $P(v_i, f)$ on these queues is shown in Figure 6.33.

In order to define this reference schedule more precisely, we define $\Phi(n, f)$ as the sum of firing durations of firing 1 up to and including firing n of aggregate firing f. We define $\Phi_0(n, f)$ as $\Phi(n, f)$, however, in $\Phi_0(n, f)$, we exclude firings that are not present given parameter valuation $P_0(v_i, f)$. This means that we exclude firings of phases with an unbounded number of firings, if these firings are additional to the minimum number of firings of that phase. More specifically, let a phase h have a firing parameter p, i.e. $\chi(v_i, h) = p$. Let p have value p_0 given parameter valuation $P_0(v_i, f)$, which we denote by $[\chi(v_i, h)]_{P_0(v_i,f)} = p_0$. In $\Phi_0(n, f)$ we exclude the fir-

ing duration of any firing $q > p_0$ of this phase h. We define $\Phi_0(n, f)$ by Equation (6.31), given that firing n in aggregate firing f is the q-th firing of phase h. In Equation (6.31), we have a term that corresponds with the cumulative firing duration of the current phase h and a term that corresponds with the previous phases. For the current phase we exclude the firing duration of any firing after p_0. For the previous phases we multiply the number of firings given parameter valuation $P_0(v_i, f)$ with the firing durations of these firings.

$$\Phi_0(n, f) = \min(q, [\chi(v_i, h)]_{P_0(v_i,f)}) \cdot \rho(v_i, h) + \sum_{k=1}^{k=h-1} [\chi(v_i, k)]_{P_0(v_i,f)} \cdot \rho(v_i, k) \quad (6.31)$$

With $s(f, \sigma(v_i, e))$ equal to the start of aggregate firing f on queue e in the constructed schedule of aggregate firings $\sigma(v_i, e)$, we define the reference start time of firing n of f on e, $s_0(n, f, e)$, by Equation (6.32).

$$s_0(n, f, e) = s(f, \sigma(v_i, e)) + \Phi_0(n - 1, f) \quad (6.32)$$

In our example, we have $P(v_i, f)$ for actor v_i from Figure 6.31 equal to $p = 3$, $q = 2$, and $m = 2$. For the five firings that follow from the parameter values $P(v_i, f)$, we have reference start times $s_0(1, f, e_{hi}) = 0$, $s_0(2, f, e_{hi}) = \tau$, $s_0(3, f, e_{hi}) = \tau$, $s_0(4, f, e_{hi}) = \tau$, $s_0(5, f, e_{hi}) = 2\tau$ on queue e_{hi} and the same reference start times on queue $e_{i\tau}$ all relative to the start time of aggregate firing f. This reference schedule is shown in Figure 6.32. In such a reference schedule, multiple firings can be assigned the same start time. For v_i from our example, firings 2, 3, and 4 all have the same start time denoted A in Figure 6.32(a) and denoted C in Figure 6.32(b). Times A and C will be the reference points, $s_0(n, f, e)$, to define the start times of firings 2, 3, and 4 that will be inserted in the reference schedule to obtain the schedules shown in Figure 6.33.

Given this reference schedule of firings, the schedule of firings of aggregate firing f on queue e is constructed as follows. The basic idea is as follows. Let firings 1 up to and including firing $n - 1$ transfer y tokens on queue e, given parameter valuation $P_0(v_i, f)$. Let these firings transfer $y + z$ tokens on e, given parameter valuation $P(v_i, f)$. By construction of $P_0(v_i, f)$, we have that $z \geq 0$. We define the start time of firing n in aggregate firing f on queue e to be $s(n, f, e) = s_0(n, f, e) + z/\hat{r}(e)$.

To define the schedule precisely, we first differentiate between a schedule on an output queue e_1 of v_i and a schedule on an input queue e_2 of actor v_i. To define the schedule on output queue e_1, we define $\Xi(n, f, e_1)$ as the sum of tokens produced in firing 1 up to and including firing n of aggregate firing f on queue e_1. As $\Phi_0(n, f)$ excluded firings not present

given parameter valuation $P_0(v_i, f)$, $\Xi_0(n, f, e_1)$ excludes tokens not produced given parameter valuation $P_0(v_i, f)$. Given that firing n of aggregate firing f is a firing of phase h, we define $\Xi_0(n, f, e_1)$ by Equation (6.33).

$$\Xi_0(n, f, e_1) = \min(n, [\chi(v_i, h)]_{P_0(v_i,f)}) \cdot [\pi(e_1, h)]_{P_0(v_i,f)} + \sum_{k=1}^{k=h-1} [\chi(v_i, k)]_{P_0(v_i,f)} \cdot [\pi(e_1, k)]_{P_0(v_i,f)} \quad (6.33)$$

The start time of firing n of aggregate firing f on output queue e_1 of v_i is defined by Equation (6.34). This start time is determined by delaying its reference start time linearly in the number of additional tokens produced by previous firings. The number of additional tokens is determined by the difference in the number of tokens produced given valuation $P(v_i, f)$ and the number of tokens produced in case of valuation $P_0(v_i, f)$. This difference in number of produced tokens can be caused by an increased number of firings or by an increase in token production quanta.

$$s(n, f, e_1) = s_0(n, f, e_1) + \frac{\Xi(n-1, f, e_1) - \Xi_0(n-1, f, e_1)}{\hat{r}(e_1)} \quad (6.34)$$

The schedule of aggregate firing f of v_i on input queue e_2 is defined analogously to the schedule on output queue e_1. To define the schedule on input queue e_2, we define $\Lambda(n, f, e_2)$ as the sum of tokens produced by firings 1 up to and including firing n of aggregate firing f. As $\Xi_0(n, f, e_1)$ excluded tokens not produced given parameter valuation $P_0(v_i, f)$, $\Lambda_0(n, f, e_2)$ excludes tokens not consumed given parameter valuation $P_0(v_i, f)$. Given that firing n is a firing of phase h, $\Lambda_0(n, f, e_2)$ is defined by Equation (6.35).

$$\Lambda_0(n, f, e_2) = \min(n, [\chi(v_i, h)]_{P_0(v_i,f)}) \cdot [\gamma(e_2, h)]_{P_0(v_i,f)} + \sum_{k=1}^{k=h-1} [\chi(v_i, k)]_{P_0(v_i,f)} \cdot [\gamma(e_2, k)]_{P_0(v_i,f)} \quad (6.35)$$

The start time of firing n of aggregate firing f on input queue e_2 of v_i is defined by Equation (6.36).

$$s(n, f, e_2) = s_0(n, f, e_2) + \frac{\Lambda(n-1, f, e_2) - \Lambda_0(n-1, f, e_2)}{\hat{r}(e_2)} \quad (6.36)$$

As shown in Figure 6.33, for the five firings that follow from valuation $P(v_i, f)$, we have the following start times. The first firing was already present in the reference schedule, $s(1, f, e_{i\tau}) = s_0(1, f, e_{i\tau}) = 0$.

The second firing is inserted in the reference schedule, but because it is the first inserted firing there are no additional tokens transferred by previous firings, $s(2, f, e_{i\tau}) = s_0(2, f, e_{i\tau}) + {}^{0}/_{\hat{r}(e_{i\tau})} = \tau + {}^{0}/_{\hat{r}(e_{i\tau})} = \tau$. For the third firing, we have that the previous firings transferred two additional tokens, $s(3, f, e_{i\tau}) = s_0(3, f, e_{i\tau}) + {}^{2}/_{\hat{r}(e_{i\tau})} = \tau + {}^{2}/_{\hat{r}(e_{i\tau})} = 3\tau$. The fourth firing was already present in the reference schedule, but its start time is translated because previous firings transferred four additional tokens, $s(4, f, e_{i\tau}) = s_0(4, f, e_{i\tau}) + {}^{4}/_{\hat{r}(e_{i\tau})} = \tau + {}^{4}/_{\hat{r}(e_{i\tau})} = 5\tau$. For the fifth firing, we have that the previous firings transferred four additional tokens, $s(5, f, e_{i\tau}) = s_0(5, f, e_{i\tau}) + {}^{4}/_{\hat{r}(e_{i\tau})} = 2\tau + {}^{4}/_{\hat{r}(e_{i\tau})} = 6\tau$. This schedule is shown in Figure 6.33(b). The schedule on queues e_{hi} and e_{ih} is shown in Figure 6.33(a). As illustrated in these figures, the start times of firings 2, 3, and 4 on queue e_{hi} are derived from point A of Figure 6.32(a) by a translation over a line with a slope $\hat{r}(e_{hi})$. On queue $e_{i\tau}$, these start times are derived from C by a translation over a line with a slope $\hat{r}(e_{i\tau})$.

This example, as shown in Figure 6.33, illustrates the main difference in this definition of aggregate firing and the definition from Section 6.3.2, which is that this definition already delays firings within an aggregate firing. We no longer have an aggregate phase that consumes $\Gamma(f, e_1)$ tokens at its start and produces $\Pi(f, e_2)$ tokens $\Upsilon(v_i, f)$ later. Instead, we bound the token transfer times of the just defined schedule by observing that the reference schedule is contained in a rectangular shape and that all start times are a linear translation of a reference start time that depends on the number of additional tokens transferred.

6.4.4 Buffer Capacity Computation

Given the definition of aggregate firing from the previous section, this section bounds token production and consumption times on a queue e_{ij}. Given these bounds the same reasoning is followed as in Sections 6.3.4, 6.3.5, and 6.3.6 to derive a sufficient number of initial tokens.

For an actor v_x, we define parameter valuation $\bar{P}(v_x)$ as follows. For every parameter of v_x that has an upper bound we take the maximum value and for every parameter that does not have an upper bound we take the minimum value. It follows that for every aggregate firing f parameter valuation $P_0(v_x, f)$ always has parameter values that are smaller than or equal to the values in $\bar{P}(v_x)$.

Given parameter valuation $\bar{P}(v_i)$, we have a cumulative firing duration $\bar{\Upsilon}(v_i)$ as defined by Equation (6.37).

$$\bar{\Upsilon}(v_i) = \sum_{h=1}^{h=\theta(v_i)} [\chi(v_i, h)]_{\bar{P}(v_i)} \cdot \rho(v_i, h) \tag{6.37}$$

We define $W(v_i)$ as the set of phases of v_i that do not have an upper bound on their number of firings, and we define $L(v_i, e_{ij})$ as the set of

phases of v_i that can produce tokens on queue e_{ij}, i.e. these phases have a parameterised token production quantum on queue e_{ij} that can attain a value larger than zero. We define $\hat{\rho}(v_i, e_{ij})$ as the largest firing duration of the phases with an unbounded number of firings that can produce on e_{ij}, i.e. $\hat{\rho}(v_i, e_{ij}) = \max(\{\rho(v_i, h) | h \in W(v_i) \cap L(v_i, e_{ij})\})$.

The following lemma gives a linear upper bound on the token production times of an aggregate firing as defined in Section 6.4.3. The derivation of this bound can be intuitively understood as follows. The reference start time of any firing n of aggregate firing f is smaller than or equal to $\bar{\Upsilon}(v_i)$. Given these reference start times, all tokens produced on e_{ij} given valuation $\bar{P}(v_i)$ are produced at latest at $\bar{\Upsilon}(v_i)$. Further, any additional firing, relative to valuation $\bar{P}(v_i)$, has a firing duration smaller than or equal to $\hat{\rho}(v_i, e_{ij})$. Therefore, given the reference start times all tokens of aggregate firing f are produced before $\bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij})$. Additional token production, relative to valuation $\bar{P}(v_i)$, results in start times that are delayed. However, these start times are delayed linearly in the number of additional tokens along the maximum required rate line, thereby not violating the upper bound on token production times.

Lemma 6.12 *An upper bound on the token production time of token number $y \geq 1$ produced in aggregate firing f of v_i on queue e_{ij} is*

$$s(f, \sigma(v_i, e_{ij})) + \bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij}) + \frac{y - 1}{\hat{r}(e_{ij})} \tag{6.38}$$

Proof. Let token y be produced by firing n of aggregate firing f. Let n be a firing of phase h of v_i. Token y is produced at the finish time of firing n, which equals

$$s(n, f, e_{ij}) + \rho(v_i, h) = \\ s_0(n, f, e_{ij}) + \frac{\Xi(n-1, f, e_{ij}) - \Xi_0(n-1, f, e_{ij})}{\hat{r}(e_{ij})} + \rho(v_i, h) \tag{6.39}$$

Therefore we have that

$$s(n, f, e_{ij}) + \rho(v_i, h) \leq s_0(n, f, e_{ij}) + \frac{\Xi(n-1, f, e_{ij})}{\hat{r}(e_{ij})} + \rho(v_i, h) \tag{6.40}$$

We distinguish two cases based on the reference start time of n, and then show that Equation (6.38) is an upper bound on the finish time of firing n in both cases.

If $s_0(n, f, e_{ij}) = s_0(n + 1, f, e_{ij})$, then according to Equation (6.32), $\Phi_0(n - 1, f) = \Phi_0(n, f)$. This implies that firing n is not included in $\Phi_0(n, f)$. Therefore, n is a firing of a phase $h \in W(v_i)$. Because token y is produced on e_{ij} by firing n, we have that $h \in L(v_i, e_{ij})$, and, therefore,

$\rho(v_i, h) \leq \hat{\rho}(v_i, e_{ij})$. This implies that in this case an upper bound on the token production time of y is

$$s(n, f, e_{ij}) + \rho(v_i, h) \leq \\ s_0(n, f, e_{ij}) + \frac{\Xi(n-1, f, e_{ij}) - \Xi_0(n-1, f, e_{ij})}{\hat{r}(e_{ij})} + \hat{\rho}(v_i, e_{ij}) \quad (6.41)$$

If $s_0(n, f, e_{ij}) \neq s_0(n+1, f, e_{ij})$, then, according to Equation (6.32), we have that $s_0(n+1, f, e_{ij}) = s_0(n, f, e_{ij}) + \rho(v_i, h)$. Substitution of $s_0(n+1, f, e_{ij}) = s_0(n, f, e_{ij}) + \rho(v_i, h)$ in Equation (6.40) implies that in this case an upper bound on the token production time of y is

$$s(n, f, e_{ij}) + \rho(v_i, h) \leq s_0(n+1, f, e_{ij}) + \frac{\Xi(n-1, f, e_{ij})}{\hat{r}(e_{ij})} \quad (6.42)$$

Because $s_0(n, f, e_{ij}) \leq s(f, \sigma(v_i, e_{ij})) + \bar{\Upsilon}(v_i)$ and $s_0(n+1, f, e_{ij}) \leq s(f, \sigma(v_i, e_{ij})) + \bar{\Upsilon}(v_i)$, an upper bound on the production time of y in both cases is given by

$$s(n, f, e_{ij}) + \rho(v_i, h) \leq s(f, \sigma(v_i, e_{ij})) + \bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij}) + \frac{\Xi(n-1, f, e_{ij})}{\hat{r}(e_{ij})} \quad (6.43)$$

Because token y is produced by firing n of f, we have that $y \geq \Xi(n-1, f, e_{ij}) + 1$. Substitution of $y - 1$ for $\Xi(n-1, f, e_{ij})$, therefore, implies that Equation (6.38) is an upper bound on Equation (6.39). □

The schedule $\sigma(v_i, e_{ij})$ of aggregate firings of actor v_i on output queue e_{ij} is defined by Equation (6.44), which equals the schedule as defined by Equation (6.7).

$$s(f, \sigma(v_i, e_{ij})) = s(v_i) + \frac{\Xi(f-1, e_{ij}) - \delta(e_{ij})}{\hat{r}(e_{ij})} \quad (6.44)$$

Let a token $x \in \mathbb{N}^*$ be produced on e_{ij} by aggregate firing f of v_i, we have that $x = \Xi(f-1, e_{ij}) + y$. With the start time of aggregate firing f given by Equation (6.44), and an upper bound on the production time of token y given by Equation (6.38) of Lemma 6.12, a linear upper bound on the production time of token x is given by Equation (6.45).

$$\hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij})) = s(v_i) + \bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij}) + \frac{x - \delta(e_{ij}) - 1}{\hat{r}(e_{ij})} \quad (6.45)$$

We define $M(v_j, e_{ij})$ as the set of phases that can consume tokens from e_{ij}, i.e. these are the phases that have a parameterised token consumption quantum on e_{ij} that can attain a value larger than zero. We define $\hat{\gamma}(e_{ij})$ as the largest token consumption quantum of the phases

with an unbounded number of firings that can consume from e_{ij}, i.e. $\hat{\gamma}(e_{ij}) = \max(\{\gamma(e_{ij}, o) | o \in W(v_i) \cap M(v_i, e_{ij})\})$.

Given parameter valuation $\bar{P}(v_i)$, we have a cumulative token consumption quantum $\bar{\Gamma}(e_{ij})$ as defined by Equation (6.46).

$$\bar{\Gamma}(e_{ij}) = \sum_{h=1}^{h=\theta(v_i)} [\chi(v_i, h)]_{\bar{P}(v_i)} \cdot \gamma(e_{ij}, h) \tag{6.46}$$

The following lemma provides a linear lower bound on token consumption times in an aggregate firing f of actor v_j on input queue e_{ij}. The derivation can be understood as follows. Given the reference parameter valuation $P_0(v_j, f)$, the firings of f together consume maximally $\bar{\Gamma}(e_{ij})$ tokens. Any firing that consumes an additional number of tokens maximally has a token consumption quantum of $\hat{\gamma}(e_{ij})$ tokens. Therefore, maximally $\bar{\Gamma}(e_{ij}) + \hat{\gamma}(e_{ij})$ tokens are consumed at once. Additional consumption relative to $\bar{\Gamma}(e_{ij})$ results in a delay of subsequent firings that corresponds with the number of additional consumed tokens and the slope of the bound on token consumption times. Therefore, these subsequent firings do not consume tokens earlier than the presented lower bound. Figure 6.33 illustrates that additional token consumption relative to the reference parameter valuation, which is shown in Figure 6.32, leads to delay of subsequent firings.

Lemma 6.13 *A lower bound on the token consumption time of token number $y \geq 1$ consumed in aggregate firing f of v_j from queue e_{ij} is*

$$s(f, \sigma(v_i, e_{ij})) + \frac{y - \bar{\Gamma}(e_{ij}) - \hat{\gamma}(e_{ij})}{\hat{r}(e_{ij})} \tag{6.47}$$

Proof. Let token y be consumed by firing n of aggregate firing f. Then y is consumed at $s(n, f, e_{ij})$, which is given by

$$s(n, f, e_{ij}) = s_0(n, f, e_{ij}) + \frac{\Lambda(n-1, f, e_{ij}) - \Lambda_0(n-1, f, e_{ij})}{\hat{r}(e_{ij})} \tag{6.48}$$

By construction of $\hat{\gamma}(e_{ij})$, we have that

$$\hat{\gamma}(e_{ij}) \geq (\Lambda(n, f, e_{ij}) - \Lambda_0(n, f, e_{ij})) - (\Lambda(n-1, f, e_{ij}) - \Lambda_0(n-1, f, e_{ij})) \tag{6.49}$$

Which can be rewritten to

$$\Lambda(n-1, f, e_{ij}) - \Lambda_0(n-1, f, e_{ij}) \geq \Lambda(n, f, e_{ij}) - \Lambda_0(n, f, e_{ij}) - \hat{\gamma}(e_{ij}) \tag{6.50}$$

Since $\Lambda_0(n, f, e_{ij}) \leq \bar{\Gamma}(e_{ij})$, and $\Lambda(n, f, e_{ij}) \geq y$, this can be rewritten to

$$\Lambda(n-1, f, e_{ij}) - \Lambda_0(n-1, f, e_{ij}) \geq y - \bar{\Gamma}(e_{ij}) - \hat{\gamma}(e_{ij}) \tag{6.51}$$

Together with Equation (6.48), this implies that

$$s(n, f, e_{ij}) \geq s_0(n, f, e_{ij}) + \frac{y - \bar{\Gamma}(e_{ij}) - \hat{\gamma}(e_{ij})}{\hat{r}(e_{ij})} \tag{6.52}$$

Since by Equation (6.32), $s_0(n, f, e_{ij}) \geq s(f, \sigma(v_i, e_{ij}))$, Equation (6.47) is a lower bound on the start time of firing n that consumes token y. Further, because token y is consumed at $s(n, f, e_{ij})$, Equation (6.47) is a lower bound on the consumption time of token y. □

The schedule $\sigma(v_j, e_{ij})$ of aggregate firings of actor v_j on input queue e_{ij} is defined by Equation (6.53), which equals the schedule as defined by Equation (6.11).

$$s(f, \sigma(v_j, e_{ij})) = s(v_j) + \frac{\Lambda(f-1, e_{ij})}{\hat{r}(e_{ij})} \tag{6.53}$$

Let a token $x \in \mathbb{N}^*$ be consumed from e_{ij} by aggregate firing f of v_j, we have that $x = \Lambda(f-1, e_{ij}) + y$. With the start time of aggregate firing f given by Equation (6.53), and a lower bound on the consumption time of token y given by Equation (6.47) of Lemma 6.13, a linear lower bound on the consumption time of token x is given by Equation (6.54).

$$\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij})) = s(v_j) + \frac{x - \bar{\Gamma}(e_{ij}) - \hat{\gamma}(e_{ij})}{\hat{r}(e_{ij})} \tag{6.54}$$

Since, on any queue, tokens can only be consumed after they have been produced, we have that for every token x, $\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij})) \geq \hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij}))$ should hold. After substitution of Equations (6.45) and (6.54) in $\check{\alpha}_c(x, e_{ij}, \sigma(v_j, e_{ij})) \geq \hat{\alpha}_p(x, e_{ij}, \sigma(v_i, e_{ij}))$, we obtain

$$s(v_j) - s(v_i) \geq \frac{\bar{\Gamma}(e_{ij}) + \hat{\gamma}(e_{ij}) - \delta(e_{ij}) - 1}{\hat{r}(e_{ij})} + \bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij}) \tag{6.55}$$

Given the constraint on the minimum difference between the start times of the first aggregate firings of two adjacent actors as specified by Equation (6.55), the constraints in the network flow problem of Section 6.3.5 are now given by Equation (6.56).

$$\beta(e_{ij}) \geq \frac{\bar{\Gamma}(e_{ij}) + \hat{\gamma}(e_{ij}) - \delta(e_{ij}) - 1}{\hat{r}(e_{ij})} + \bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij}) \tag{6.56}$$

After the start times of the first aggregate firings of each actor are computed by solving the network flow problem of Algorithm 6.1 using Equation (6.56), the required number of initial tokens on queue e_{ij} is the smallest integer value that satisfies the constraint in Equation (6.57).

$$\delta(e_{ij}) \geq \hat{r}(e_{ij}) \cdot (\bar{\Upsilon}(v_i) + \hat{\rho}(v_i, e_{ij}) + s(v_i) - s(v_j)) + \bar{\Gamma}(e_{ij}) + \hat{\gamma}(e_{ij}) - 1 \tag{6.57}$$

6.4.5 Sufficiency of Buffer Capacities

In this section, we show that the number of initial tokens as computed in the previous section is a sufficient buffer capacity to let w_τ execute wait-free. This is done by showing that Theorem 6.5 still holds, which, in turn, is done by showing that Lemmas 6.10 and 6.11 from Section 6.3 still hold given the definition of aggregate firings from Section 6.4.

In Section 6.3, schedules of aggregate firings were constructed per queue for every parameter valuation. Lemma 6.10 showed that the constructed queue-schedule of aggregate firings on any queue e_{ij} that is on a simple directed path from v_i to v_τ has start times that are later than or equal to the start times in the schedules on other adjacent queues of v_i. Therefore, the schedule on e_{ij} determines the actor-schedule of v_i and delays the start times on other adjacent queues. However, while in Section 6.3, the schedule of firings within an aggregate firing is the same on every queue, this no longer holds for the aggregate firing as defined in Section 6.4.3. This means that, with the definition of an aggregate firing from Section 6.4.3, we have a schedule of firings that does not have to be the same on every queue. It is, therefore, no longer sufficient to show the relation between schedules of aggregate firings.

Lemma 6.16 shows that the start times in the constructed schedules of firings on any queue e_{ij} that is on a simple directed path from v_i to v_τ are later than or equal to the start times of firings of v_i in the schedules on other queues adjacent to v_i. Therefore, the schedule of actor v_i is determined by constructed queue-schedules of firings on the same queues as the queues that in Lemma 6.10 determined the schedule of aggregate firings.

In Section 6.3, Lemma 6.11 showed that if actors v_i and v_j share a parameter p, then the token arrival times on queues with cumulative transfer quanta that are parameterised in p do not influence the schedule of actors v_i and v_j. This result implied that the sub-graph G'_p as for instance shown in Figure 6.24 does not influence the schedule of v_τ. In this section, Lemma 6.17 shows that the result of Lemma 6.11 is not affected by the change in definition of aggregate firings. Since only Lemmas 6.10 and 6.11 involve schedules of aggregate firings, Theorem 6.5 still holds.

The organisation of this section is as follows. Lemma 6.14 shows that the schedule of aggregate firings on queues towards v_τ will delay the schedules of aggregate firings on queues away from v_τ. This will help in establishing Lemma 6.16 that states that this same property also holds for the schedule of firings. This section is concluded by Lemma 6.17.

Lemma 6.14 *Lemma 6.10 holds for aggregate firings as defined in Section 6.4.3.*

Proof. We need to show that in the schedules determined for queues individually, the start times of firings of v_i on queue e_{ij} are later than or equal to the start times of firings of v_i on queues e_{ik} and e_{hi}. Because the

schedule of aggregate firings is the same as considered in Lemma 6.10, we have $s(f, \sigma(v_i, e_{ij})) \geq s(f, \sigma(v_i, e_{ik}))$ and $s(f, \sigma(v_i, e_{ij})) \geq s(f, \sigma(v_i, e_{hi}))$ for the same reasons as in the proof of Lemma 6.10. □

Lemma 6.16 shows that for an actor v_i the constructed schedule on a queue e_{ij} that is towards v_τ has later start times of firings than the constructed schedules on other queues adjacent to v_i.

For the VPDF graph shown in Figure 6.31, we have that, for actor v_i, queue $e_{i\tau}$ is a queue towards v_τ. In Figure 6.33, we, for instance, see that the start time of the fourth firing is later in the schedule on $e_{i\tau}$, which is point C' in Figure 6.33(b), than its start time in the schedule on e_{hi}, which is point A' in Figure 6.33(a).

In order to show Lemma 6.16, we require the following lemma. The property $D(e)$ of an edge e, as used in the following lemma, is defined in Definition 6.11 and is true for an edge that is from an actor v on a path towards v_τ and has a cumulative production quantum that is not parameterised in a shared parameter.

Lemma 6.15 *Lemmas 6.8 and 6.9 hold in case the VPDF graph has parameters with no upper bound. This means that Equation 6.58 and Equation 6.59 hold for output queues e_{ij} and e_{ik} and input queue e_{hi} of actor v_i.*

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e_{ij}, e_{ik} \in E(v_i) \bullet D(e_{ij}) \implies \frac{\hat{r}(e_{ik})}{\hat{r}(e_{ij})} = \max_{\phi(P)} \left(\frac{\Pi(e_{ik})}{\Pi(e_{ij})} \right) \tag{6.58}$$

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e_{ij}, e_{hi} \in E(v_i) \bullet D(e_{ij}) \implies \frac{\hat{r}(e_{hi})}{\hat{r}(e_{ij})} = \max_{\phi(P)} \left(\frac{\Gamma(e_{hi})}{\Pi(e_{ij})} \right) \tag{6.59}$$

Proof. The proof of Lemmas 6.8 and 6.9 rests on showing independence of a number of terms in the ratio that determines the maximum required rate on a queue. This independence is not affected by allowing parameters to not have an upper bound, neither is the fact that we now need to determine limits to find the maximum required rate of influence on this independence. □

Lemma 6.16 *The following holds:*

$$\forall v_i \in V \setminus \{v_\tau\}\ \forall e, e_{ij} \in E(v_i) \bullet D(e_{ij}) \implies s(n, f, e_{ij}) \geq s(n, f, e) \tag{6.60}$$

Proof. We will show that, for every output queue e_{ik} of actor v_i, $s(n, f, e_{ij}) \geq s(n, f, e_{ik})$. The proof that, for every input queue e_{hi} of v_i,

$s(n, f, e_{ij}) \geq s(n, f, e_{hi})$ is completely analogous. We have that $s(n, f, e_{ij})$ is given by Equation (6.61).

$$s(n, f, e_{ij}) = s_0(n, f, e_{ij}) + \frac{\Xi(n-1, f, e_{ij}) - \Xi_0(n-1, f, e_{ij})}{\hat{r}(e_{ij})} \tag{6.61}$$

Further, we have that $s(n, f, e_{ik})$ is given by Equation (6.62).

$$s(n, f, e_{ik}) = s_0(n, f, e_{ik}) + \frac{\Xi(n-1, f, e_{ik}) - \Xi_0(n-1, f, e_{ik})}{\hat{r}(e_{ik})} \tag{6.62}$$

By construction, we have that $s_0(n, f, e_{ij}) = s_0(1, f, e_{ij}) + \Phi_0(n-1, f)$. Because Lemma 6.14 tells that $s_0(1, f, e_{ij})$ on queue e_{ij} is larger than or equal to $s_0(1, f, e_{ik})$ on queue e_{ik}, we need to show that Equation (6.63) is true.

$$\frac{\Xi(n-1, f, e_{ij}) - \Xi_0(n-1, f, e_{ij})}{\hat{r}(e_{ij})} \geq \frac{\Xi(n-1, f, e_{ik}) - \Xi_0(n-1, f, e_{ik})}{\hat{r}(e_{ik})} \tag{6.63}$$

The terms $\Xi(n-1, f, e_{ij}) - \Xi_0(n-1, f, e_{ij})$ and $\Xi(n-1, f, e_{ik}) - \Xi_0(n-1, f, e_{ik})$ denote increases in the number of tokens produced on queues e_{ij} and e_{ik} in the schedule of firings, which are relative to the number of tokens produced in the reference schedule of firings. This increase in the number of tokens produced on queues e_{ij} and e_{ik} is caused by an increase in the number of firings of phases of v_i that have an unbounded number of firings. The increase in the number of tokens produced on queues e_{ij} and e_{ik} equals the sum of the token production quanta on these queues of these additional firings. Let one of these additional firings be a firing of phase m of actor v_i, with a token production quantum $\pi(e_{ij}, m)$ on queue e_{ij} and a token production quantum $\pi(e_{ik}, m)$ on queue e_{ik}. We will show that Equation (6.64) holds for any such additional firing from a phase m.

$$\frac{\pi(e_{ij}, m)}{\hat{r}(e_{ij})} \geq \frac{\pi(e_{ik}, m)}{\hat{r}(e_{ik})} \tag{6.64}$$

The numerators of the left and right-hand side of Equation (6.63) are sums of token production quanta, where these sums have an equal number of terms, because this number of terms is determined by the number of additional firings of actor v_i. Because for each of these terms, Equation (6.64) holds, Equation (6.63) holds. We have that Equation (6.64) can be rewritten into Equation (6.65).

$$\frac{\hat{r}(e_{ik})}{\hat{r}(e_{ij})} \geq \frac{\pi(e_{ik}, m)}{\pi(e_{ij}, m)} \tag{6.65}$$

Lemma 6.15 states that Equation (6.66) holds.

$$\frac{\hat{r}(e_{ik})}{\hat{r}(e_{ij})} = \max_{\phi(P)} \left(\frac{\Pi(e_{ik})}{\Pi(e_{ij})} \right) \tag{6.66}$$

We have that m is a phase with an unbounded number of firings. Let p be the parameterised number of firings of m, i.e. $p = \chi(v_i, m)$, then we have that Equation (6.67) holds, because taking the limit to infinity in p is only one of the cases considered when determining the maximum value of the ratio over all possible parameter values.

$$\max_{\phi(P)} \left(\frac{\Pi(e_{ik})}{\Pi(e_{ij})} \right) \geq \lim_{p \to \infty} \left(\frac{\Pi(e_{ik})}{\Pi(e_{ij})} \right) \tag{6.67}$$

A cumulative production quanta is a sum of products where these products have the parameterised number of firings and the parameterised token production quanta as factors. Taking the limit in the parameterised number of firings p evaluates in this case to the ratio of its coefficients in the numerator and denominator, which implies that this limit evaluates to the ratio of production quanta of phase m. This means that Equation (6.68) holds.

$$\lim_{p \to \infty} \left(\frac{\Pi(e_{ik})}{\Pi(e_{ij})} \right) = \frac{\pi(e_{ik}, m)}{\pi(e_{ij}, m)} \tag{6.68}$$

Substitution of terms in Equations (6.66), (6.67), and (6.68) results in Equation (6.65), which therefore holds for any such phase m. Since Equation (6.65) can be rewritten to Equation (6.64), which therefore holds for any of the terms of Equation (6.63). □

Given that v_i and v_j, with $v_i \neq v_j$, share parameter p, where on output queue e_{ik} of v_i it holds that $\Pi(e_{ik})$ is parameterised in p, and on input queue e_{hj} of v_j it holds that $\Gamma(e_{hj})$ is parameterised in p. Lemma 6.17 will show that the schedule of firings of v_i on queue e_{ik} and of v_j on e_{hj} as defined in Sections 6.4.3 and 6.4.4 are the same as defined in Section 6.3. Therefore, Lemma 6.11 still holds. Lemma 6.11 states that the constructed actor-schedules of v_i and v_j are not influenced by the value of a shared parameter such as parameter p.

Lemma 6.17 *Lemma 6.11 holds for aggregate firings as defined in Section 6.4.3.*

Proof. Let actors v_i and v_j, with $v_i \neq v_j$, share a parameter p. Further let v_i have an adjacent queue e_i with a cumulative token transfer quantum that is parameterised in p and let v_j have an adjacent queue e_j with a cumulative token transfer quantum that is parameterised in p.

By consistency of the graph all parameters of the cumulative token transfer quanta of v_i on e_i and all parameters of the cumulative token transfer quanta of v_j on e_j are shared between v_i and v_j, or are constant, because these cumulative transfer quanta share parameter p. Because of the restrictions on parameter sharing, all shared parameters have an upper

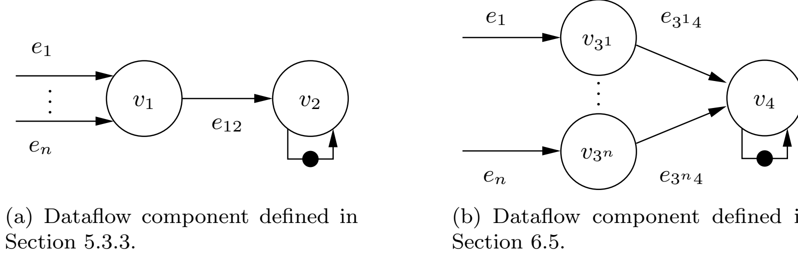
bound, i.e. have a maximum value. This implies that there is no parameter in the mentioned cumulative transfer quanta that has no upper bound. All differences between the schedule of firings on a queue e and the reference schedule of firings on this queue e are caused by insertion of firings from phases with an unbounded number of firings that transfer tokens on queue e. Since the mentioned cumulative transfer quanta are not parameterised in a parameter that has no upper bound, their queue-schedules are equal to their reference queue-schedules. The reference queue-schedule of firings as defined in Section 6.4 equals the schedule of firings as defined for the rectangular aggregate firings in Section 6.3. Therefore, the definitions of the queue-schedules in Section 6.4 and Section 6.3 result in the same schedule of firings in the queue-schedule of v_i on e_i and the same schedule of firings in the queue-schedule of v_j on e_j. □

6.5 Modelling Run-Time Scheduling

In Section 6.2.3, we required that the relation between the task graph and the VPDF graph is a one-to-one relation between tasks and actors. This implies that the effects of run-time scheduling can only be captured in a VPDF graph using response times and not using the more accurate model with a latency and a rate parameter. This is because the model with a latency and a rate parameter results in a VPDF graph where a task is modelled by two actors of which one actor only models latency and is allowed to fire concurrently to itself. In this chapter, we defined a VPDF actor as an actor that cannot fire concurrently to itself. We see the extension of the semantics of VPDF to allow for auto-concurrency as future work.

However, VPDF graphs that capture the effects of run-time scheduling with a latency-rate dataflow component are a special case for which the semantics and the presented buffer capacity algorithm can be relatively straightforwardly extended. In the following, we first introduce a new dataflow component that we prove equivalent to the dataflow component defined in Section 5.3.3. This new dataflow component only has actors with auto-concurrency that are single-rate dataflow actors, for which their semantics is known (Reiter 1968). Subsequently, we discuss the required adaptation to our buffer capacity computation algorithm. We discuss this extension to VPDF in this section, because it is a very specific case that would have unnecessarily complicated our formalism and proofs.

As defined in Section 5.3.3, a dataflow component has two actors, actor v_1 and v_2, of which actor v_1 has no self-edge, and can fire auto-concurrently. In Section 5.3.3, there is a one-to-one relation between task executions and component firings, where a component firing is a firing of v_1 and a firing of v_2. Furthermore, there is a one-to-one relation between the number of containers consumed and produced by a task execution and the number of



(a) Dataflow component defined in Section 5.3.3.

(b) Dataflow component defined in Section 6.5.

Figure 6.34: Two dataflow components that are used to capture the effects of run-time scheduling of a task.

tokens consumed and produced by the corresponding dataflow component firing, i.e. with the number of tokens consumed by the firing of v_1 and the number of tokens produced by the firing of v_2. Theorem 5.7 states that the effects of run-time scheduling are conservatively captured with a latency-rate dataflow component, if each firing of actor v_1 of this component has a firing duration equal to the difference between the period and the allocated budget, i.e. $Q - R$.

Given the dataflow component as defined in Section 5.3.3, we define a new equivalent dataflow component as follows. Let the dataflow component from Section 5.3.3 have actors v_1 and v_2 and let v_1 have n input queues e_i, with $i \in [1, n]$, as exemplified in Figure 6.34(a). We define a new dataflow component with n actors v_{3^i}, and one actor v_4, as exemplified in Figure 6.34(b). Actor v_4 is the same as v_2 except that it now has the firing rules of actor v_1, i.e. actor v_4 has $n + 1$ input queues, while v_2 has 2 input queues. For every input queue e_i of v_1, we now have an actor v_{3^i} that has queue e_i as its input queue, and has an output queue e_{3^i4} to v_4. Every actor v_{3^i} consumes one token and produces one token in every firing. Every firing of an actor v_{3^i} has a firing duration equal to $Q - R$, while the firing durations of v_4 equal the firing durations of v_2.

This new dataflow component is equivalent to the dataflow component defined in Section 5.3.3, because the relation between token arrival times on the input and output queues of the dataflow component is the same. This is shown by the following theorem.

Theorem 6.6 *The token arrival times on the output queues of the new dataflow component defined in the previous paragraph equal the token arrival times of the dataflow component that is defined in Section 5.3.3.*

Proof. Consider the dataflow component from Section 5.3.3, let a firing f of actor v_1 consume p tokens, then the external enabling time of this firing of v_1 equals $\hat{e}(v_1, f) = \max(\{\hat{a}(1), \ldots, \hat{a}(p)\})$. This results in an external enabling time of firing f of v_2 that equals $\hat{e}(v_2, f) = \hat{e}(v_1, f) + Q - R =$

$\max(\{\hat{a}(1), \ldots, \hat{a}(p)\}) + Q - R$. In our new dataflow model, we have that any token l with an arrival time of $\hat{a}(l)$ on the input queue of the dataflow component has an arrival time of $\hat{a}(l) + Q - R$ on the input queue of actor v_4. The external enabling time of actor v_4 is, therefore, $\hat{e}(v_4, f) = \max(\{\hat{a}(1) + Q - R, \ldots, \hat{a}(p) + Q - R\}) = \max(\{\hat{a}(1), \ldots, \hat{a}(p)\}) + Q - R$. Because the external enabling time of v_4 equals the external enabling time of v_2, and because the firing durations and produced tokens of v_4 are the same as those of v_2, the token arrival times on the output queues of v_4 are the same as the token arrival times on the output queues of v_2. □

Given a VPDF graph that satisfies all restrictions that are specified in Section 6.2, in which an actor v_4 has queues e_1 through e_n as input queues. The VPDF graph in which every such input queue e_i is now to an actor v_{3^i} which (1) can fire auto-concurrently, (2) has an output queue to v_4, and (3) consumes and produces one token in every firing, is also a valid VPDF graph. This is because this actor v_{3^i} only adds time to token arrival times and does not change the semantics of the VPDF graph.

Given this actor v_4 and constructed queue-schedule $\sigma(v_4, e_i)$ on this input queue e_i. The effect of the extension of the graph with actor v_{3^i} is that tokens need to arrive $Q - R$ earlier on e_i to sustain this queue-schedule. Therefore, if we consider the original VPDF graph and decrease the lower bound on consumption times with $Q - R$ on every queue e_i that is extended with an actor v_{3^i}, then the effects of v_{3^i} are included in the buffer capacity computation.

6.6 Experiment

In this section, we show two example applications that can be modelled and analysed by VPDF graphs. The first example is a fragment from an IEEE 802.11 receiver taken from (Moreira and Bekooij 2007), which exemplifies loops with no known upper bound on the number of iterations. The second example is an H.263 video decoder, which exemplifies communication of parameter values.

6.6.1 802.11 Receiver

Figure 6.35 shows a simplified task graph of an 802.11 receiver, with a sink that periodically produces samples, a channel decoding task, CD, and a source decoding task, SD. For every packet decoded by CD, first a signal from the base-station needs to be detected and synchronised to. This is done in the first non-blocking code-segment, which executes $d \geq 2$ times. Subsequently, the packet is received which implies reading n samples. Then m bytes are written into the buffer to the source decoder.

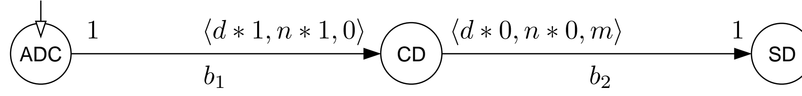


Figure 6.35: Simplified task graph of an 802.11 receiver.

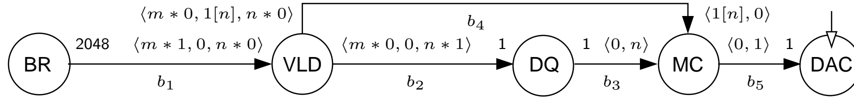


Figure 6.36: Simplified task graph of an H.263 video decoder.

The number of samples that need to be processed before a signal is detected and synchronised to depends on various channel conditions. This implies that there is no known upper bound on d. In (Moreira and Bekooij 2007) this aspect of the radio receiver could not be included in their model. Analysis of the VPDF model that corresponds with the task graph from Figure 6.35 results in buffer capacities that are sufficient to not lose samples at the sink in the different modes and transitions between these modes of this receiver.

In case the ADC has a response time of 10, the vector of response times of the CD is $\langle d * 1, n * 1, 10\rangle$, and the response time of the SD is 2, with furthermore $n \geq 0$ and $0 \leq m \leq 10$, then a buffer capacity of 5 on buffer b_1 and a buffer capacity of 16 on buffer b_2 is computed by our algorithm. Simulations in our dataflow simulator confirm that these are sufficient capacities.

6.6.2 H.263 Video Decoder

Figure 6.36 shows a simplified task graph of an H.263 video decoder. This decoder includes a BR task that reads bytes from a storage device, a VLD task that parses the byte stream and produces macro blocks, a DQ task that dequantises, a MC task that assembles a picture, and the sink that periodically consumes a picture. The DQ task processes macro blocks. Therefore, the MC task needs to be informed by the VLD task how many macro blocks comprise a complete picture. In contrast to VRDF (Wiggers et al. 2008b), with VPDF, the value of m does not need to be known before the first execution of this non-blocking code-segment is executed. This allows to model a VLD task that has a loop in which it consumes bytes and exits this loop based on the processed byte stream.

Suppose that the BR has a response time of 10000, the vector of response times of the VLD is $\langle m * 3, 3, n * 3\rangle$, the DQ has a response time of

14, the MC has response times $\langle 1000, 32000 \rangle$ and the DAC has a response time of 33000. Furthermore suppose that $0 \leq m \leq 6536$ and $0 \leq n \leq 2376$. Then this results in computed capacities of 15859 on b_1, 4302 on b_2, 4752 on b_3, 3 on b_4, and 2 on b_5.

The presented algorithm is implemented in a tool using the GiNaC (Bauer et al. 2002) library for the algebraic manipulations. Buffer capacities are computed for all presented graphs and confirmed to be conservative with our dataflow simulator.

6.7 Conclusion

This chapter presented an algorithm that uses a new dataflow model, variable-rate phased dataflow, to compute buffer capacities that satisfy timing and resource constraints for task graphs with inter-task synchronisation behaviour that is dependent on the processed data stream.

A variable-rate phased dataflow actor can model a task that consists of a sequence of loops, where these loops can have a number of iterations that depends on the processed stream. This data-dependent number of iterations leads to data-dependent task execution rates. For example, task execution in a digital radio receiver can depend on whether the receiver is in the synchronisation or synchronised mode.

We have shown that the validity of instances of variable-rate phased dataflow can be efficiently verified. Furthermore, for every valid instance, buffer capacities can be efficiently computed that are sufficient to satisfy a throughput constraint and any constraints on maximum buffer capacities.

By exposing individual loop iterations of a task, buffer capacities can still be computed for tasks that include loops with an unbounded number of iterations. This is done by bounding, on each buffer, the variation in transfer rate relative to the required transfer rate. This in contrast to bounding the cumulative response time and transfer quanta, which are unbounded in case of unbounded iteration.

Chapter 7

Case Study

Abstract – *The applicability and correctness of our approach is verified in this chapter through a case-study with an MP3 playback application, which executes on a multiprocessor system with run-time schedulers that guarantee resource budgets. Simulations in a cycle-accurate simulator confirm that our approach leads to buffer capacities that are indeed sufficient to satisfy the throughput constraint.*

In this chapter, we will look at the applicability and correctness of our approach by considering an MP3 playback application that executes on a multiprocessor system with a shared memory and two processors. This MP3 playback application consists of three tasks: a block reader, an MP3 decoder, and a sample rate converter. The output of the sample rate converter is fed to a digital-to-analog converter that forms the time-triggered interface with the environment. For analysis purposes, we consider this interface as a task and check whether data arrival times are such that this task can execute strictly periodically with the required frequency. We used the MAD (Underbit Technologies 2009) MP3 decoder and the sample rate converter from the libsamplerate library (Castro Lopo 2009), which are both publicly available.

In this application, the MP3 decoder consumes a number of bytes from its input buffer that depends on the processed data stream, which is a variable bitrate stream in our experiments. We can model this data-dependent inter-task synchronisation behaviour with our VPDF model. We are not aware of alternative approaches that can compute buffer capacities that

are guaranteed to be sufficient to satisfy the throughput constraint for this MP3 playback application.

We show that this MP3 playback application can be modelled by a VPDF graph, both in case the tasks execute in isolation and in case the tasks are scheduled by run-time schedulers that guarantee resource budgets. Furthermore, cycle-true simulation confirms that the buffer capacities computed with the VPDF model are indeed sufficient to satisfy the throughput requirement of this MP3 playback application. This shows that our approach is applicable in a setting with real-life software.

Even though we are not aware of alternative approaches for the complete MP3 playback application, existing approaches are applicable for the part of the task graph consisting of the MP3 decoder, sample rate converter, and digital-to-analog converter. These alternative approaches derive buffer capacities with multi-rate or cyclo-static dataflow graphs. The first section of this chapter presents multi-rate and cyclo-static dataflow models of this part of the application when all tasks execute in isolation, i.e. with no resource sharing. Since multi-rate and cyclo-static dataflow graphs are special cases of VPDF graphs, we can apply our buffer capacity computation algorithm from Chapter 6 on these graphs. We compare the accuracy of our results and the run-time of our algorithm with the alternative approaches.

Subsequently, in the second section of this chapter, we consider the situation when the tasks are scheduled by a run-time scheduler that guarantees resource budgets to the tasks. With multi-rate and cyclo-static dataflow models, we capture the effects of run-time scheduling by a model based on response times and by a model with a latency and a rate parameter. A realistic example is presented in which analysis with a model based on response times does not lead to useful results. Analysis using a latency-rate model does lead to results with an accuracy that is satisfactory. Again, we apply our buffer capacity computation algorithm that uses VPDF graphs on these multi-rate and cyclo-static graphs and evaluate the accuracy of our results.

The third section of this chapter considers a task graph formed by the block reader, MP3 decoder, and digital-to-analog converter. We are not aware of alternative approaches that provide guarantees on the satisfaction of timing constraints and that can model and analyse the inter-task synchronisation behaviour of the MP3 decoder on its input buffer, since it depends on the processed data stream. We compute buffer capacities for this task graph that are sufficient to satisfy the throughput constraint with our algorithm based on VPDF graphs from Chapter 6. These buffer capacities as well as all other analytical results presented in this chapter are confirmed to be correct by simulations in our cycle-true simulator. In our cycle-accurate simulator, we have two instances of the swarm (SWARM 2003), which is a model of an ARM7, and a digital-to-analog converter. In this simulator, all three entities have single cycle accesses to a single shared memory in which all private data, shared data, and instructions are placed.

The outline of this chapter is as follows. Section 7.1 presents experiments that consider the task graph with the MP3 decoder, sample rate converter and digital-to-analog converter, where these tasks execute in isolation. The experiments in Section 7.2 consider the same task graph, but now the tasks are executed on resources with run-time schedulers that guarantee resource budgets. The task graph with the block reader, MP3 decoder and digital-to-analog converter is considered in the experiment presented in Section 7.3. We conclude with Section 7.4.

7.1 Execution in Isolation

In this section, the task graph consisting of the MP3 decoder, the sample rate converter and the digital-to-analog converter is considered. The MP3 decoder processes a 48 kHz variable bitrate MP3 file, while the sample rate converter converts this to a 44.1 kHz stream to match the frequency of the digital-to-analog converter. We let the MP3 decoder and sample rate converter execute on different processors. In our cycle-true simulator, we can only have the digital-to-analog converter sample on an integer number of processor clock cycles. With a sampling frequency of 44.1 kHz of the digital-to-analog converter, this results in an assumption that the processors are clocked at 220.5 MHz and that the digital-to-analog converter samples every 5000 cycles after it is started.

The sample rate converter is implemented as follows. In every execution a number of input samples is consumed and processed that is fixed over all executions. Depending on the conversion ratio, a specific number of output samples is produced, which can vary from execution to execution. This is because, in case the number of input samples and conversion ratio is such that in fact a non-integer number of output samples would be required, then the number of produced samples over multiple executions will approximate the required conversion ratio. In our first experiment, we let the sample rate converter process 480 input samples per execution upon which it produces 441 samples in every execution, i.e. in this case the number of produced samples is constant over all executions. This inter-task synchronisation behaviour can be modelled with multi-rate dataflow. Subsequently, in our second experiment, we let the sample rate converter process 48 input samples per execution. In this case a sequence of ten executions is required to obtain the required sample rate conversion. This results in a cyclo-static sequence of produced output samples, which can no longer be modelled with multi-rate dataflow, but can be modelled with cyclo-static dataflow. Intuitively, a smaller synchronisation granularity allows for a smaller buffer. We show that deriving buffer capacities with cyclo-static dataflow can indeed leverage this smaller synchronisation granularity to obtain smaller buffers.

Deriving an accurate estimated worst-case execution time is outside

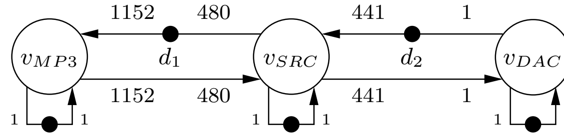


Figure 7.1: Multi-rate dataflow graph of task graph with sample rate converter that processes 480 samples per execution.

the scope of this thesis. In these experiments, we considered only a single input stream of 30 MP3 frames and took the maximum of the observed execution times as our estimated worst-case execution time. With this procedure the MP3 decoder has an estimated worst-case execution time of 1603621 clock cycles. In every execution the MP3 decoder produces 1152 samples. In all experiments, execution times are given in clock cycles. The buffer capacities are given in bytes for the buffer between the block reader and the MP3 decoder and given in 16 bit samples for the other buffers.

Experiment 1 In our first experiment, we let the sample rate converter process 480 input samples upon which it produces 441 output samples in every execution. The task graph with this behaviour of the sample rate converter can be modelled as a multi-rate dataflow graph, as shown in Figure 7.1.

Given this behaviour of the sample rate converter, the maximum execution time that we observed for our input stream is 1320974 clock cycles, which we take as the estimated worst-case execution time of this task.

The multi-rate dataflow graph as shown in Figure 7.1 is a special case of a VPDF graph. With the algorithm from Chapter 6, we derive that a buffer capacity of 2267 samples is sufficient for the buffer between the decoder and the sample rate converter, and that a buffer capacity of 706 samples is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. These capacities guarantee that data is available every time the digital-to-analog converter samples its input buffer, given that its first sampling time is 5125000 clock cycles later than the MP3 decoder starts its first execution. Simulation in our cycle-accurate simulator with these buffer capacities and this start time for the digital-to-analog converter confirm that these results are conservative and guarantee that data arrives in time at the digital-to-analog converter. Latency as defined in Section 3.4 corresponds in this experiment with the required difference in start times of the first executions of the digital-to-analog converter and MP3 decoder, such that data arrives in time at every execution of the digital-to-analog converter. This required difference in start times of the first executions of tasks is computed as part of our algorithm to compute buffer capacities as presented in Chapter 6.

Our approach directly computes a conservative schedule that satisfies the throughput constraint and derives buffer capacities given this schedule. The approach described in (Stuijk et al. 2006a) determines the exact throughput of the multi-rate dataflow graph for given buffer capacities and iterates over the possible buffer capacities to find the minimal buffer capacities that satisfy a throughput constraint. This approach is implemented in a tool called SDF3 (Stuijk et al. 2006b). Using this exact approach, we derive that a buffer capacity of 1536 samples for the buffer between MP3 decoder and sample rate converter and a buffer capacity of 517 samples for the buffer between the sample rate converter and the digital-to-analog converter are sufficient. This approach does not directly compute a sufficient start time for the digital-to-analog converter. However, with the start time that our approach computed these buffer capacities are sufficient to let data arrive in time in our cycle-accurate simulation. This might be counter-intuitive, because in our buffer capacity computation algorithm of Chapter 6 a later start time of the first firing of an actor implies a larger buffer capacity. However, our task graph only has a single time-triggered interface, which acts as a sink of the task graph, i.e. it does not have any output buffers. In this case, we can delay the start of this interface without requiring larger buffers. This is because the delay is relative to a start time from which the interface can sample data strictly periodically and all tasks first wait on empty buffer locations in their output buffers before they write data in these buffers. Therefore, there is sufficient data present at the interface from its delayed start time and no data is lost.

For the given input stream, we explored which buffer capacities are sufficient to let data arrive in time in our cycle-accurate simulation. In general, there is a trade-off possible between the capacities of different buffers, i.e. a smaller capacity for one buffer can sometimes be found after increasing the capacity of another buffer. We have strived to explore this trade-off space when deriving sufficient buffer capacities in our cycle-true simulator. In the experiments of this chapter the trade-off space is relatively limited, because in all experiments we have a result in which there is one buffer that cannot be further reduced without introducing deadlock. In this experiment, we found that a buffer capacity of 1536 for the buffer between the MP3 decoder and the sample rate converter and a buffer capacity of 442 samples for the buffer between the sample rate converter and the digital-to-analog converter is sufficient to let data arrive in time in our cycle-accurate simulation, given that the digital-to-analog converter starts 5125000 cycles after the first start of the MP3 decoder.

Experiment 2 In our second experiment, we let the sample rate converter process 48 input samples per execution. The number of produced samples follows a cyclo-static sequence, in which the sample rate converter first produces 45 output samples after which it has nine executions in which

	MP3 – SRC	SRC – DAC
Buffer capacity computation	2267	706
Exact buffer capacity derivation	1536	517
Simulation	1536	442

Table 7.1: Buffer capacities computed in Experiment 1.

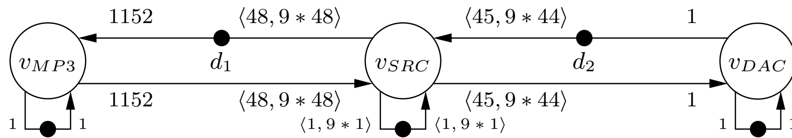


Figure 7.2: Cyclo-static dataflow graph of task graph with sample rate converter that processes 48 samples per execution.

it produces 44 output samples. This is because after these ten executions the required sample rate conversion of $^{480}/_{441}$ is obtained. The task graph with this inter-task synchronisation behaviour of the sample rate converter can be modelled as a cyclo-static dataflow graph, as shown in Figure 7.2. Cyclo-static dataflow actors have sequences of phases in which each phase is executed once per iteration through all phases. In this experiment, we consider actor v_{SRC} in Figure 7.2 to have ten phases that are each executed once, which is compactly represented in Figure 7.2.

We take the maximum observed execution time of each phase as the estimated worst-case execution time of that phase. For the sequence of ten phases of the sample rate converter, this results in the following sequence of estimated worst-case execution times: 136577, 133824, 133760, 133750, 133748, 133863, 133844, 133955, 133882, 133862. The smaller synchronisation granularity results in more calls to the read and write functions that perform the inter-task communication, which results in a sum of estimated worst-case execution times that is higher than the estimated worst-case execution time of the sample rate converter configured to process 480 samples every execution.

The cyclo-static dataflow graph as shown in Figure 7.2 is a VPDF graph. With the algorithm from the previous chapter, we derive that a buffer capacity of 2272 samples is sufficient for the buffer between the MP3 decoder and the sample rate converter and a buffer capacity of 710 samples is sufficient for the buffer between the sample rate converter and the digital to analog converter. These buffer capacities are sufficient to let data arrive in time at the digital-to-analog converter, given that the digital-to-analog converter starts 5145090 cycles later than the first start of the MP3 de-

coder. Simulation in our cycle-accurate simulator confirmed that these buffer capacities and this start time for the digital-to-analog converter are sufficient to let data arrive in time at the digital-to-analog converter.

The exact approach to derive buffer capacities for multi-rate dataflow graphs as described in (Stuijk et al. 2006a) is extended to cyclo-static dataflow graphs in (Stuijk et al. 2008), and implemented in a new version of their tool SDF3. Using this approach, we derive that a buffer capacity of 1152 is sufficient for the buffer between the MP3 decoder and the sample rate converter and a buffer capacity of 65 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. Simulation in our cycle-accurate simulator confirmed that these capacities are indeed sufficient given that the digital-to-analog converter starts 5145090 cycles after the first start of the MP3 decoder.

Given our input stream, exploration of various buffer capacities in our cycle-accurate simulator resulted in the following observation for this task graph. We found that a buffer capacity of 1152 for the buffer between the MP3 decoder and the sample rate converter and a buffer capacity of 45 for the buffer between the sample rate converter and the digital-to-analog converter are sufficient to let data arrive in time at the digital-to-analog converter, given that the digital-to-analog converter starts 5145090 cycles after the first start of the MP3 decoder.

	MP3 – SRC	SRC – DAC
Buffer capacity computation	2272	710
Exact buffer capacity derivation	1152	65
Simulation	1152	45

Table 7.2: Buffer capacities computed in Experiment 2.

Discussion Both experiments confirm that the buffer capacities that we compute with the algorithm from Chapter 6 are sufficient to satisfy the specified throughput constraint. This means that our approach is applicable in a setting with real-life software. The run-time of our algorithm is a number of milliseconds, while the run-time of SDF3 is a number of minutes. As discussed in Chapter 1, we envision that an automated mapping flow in which buffer capacity computation will be performed very often to evaluate other settings such as scheduler settings and task to processor mapping. The low run-time of our algorithm enables this approach.

The accuracy of our algorithm in these experiments is not completely satisfactory. For the first experiment with the multi-rate dataflow graph this is because the accuracy of our algorithm decreases when the throughput constraint is weak and there is a lot of scheduling freedom. Experiments

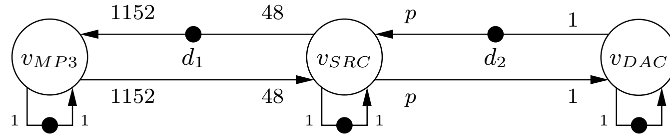


Figure 7.3: VPDF model of sample rate converter, with $p \in \{44, 45\}$.

with tighter constraints show a better accuracy (Wiggers et al. 2006). The second experiment exposes a short-coming of our current algorithm that computes buffer capacities with VPDF graphs. We have shown that buffer capacities can be efficiently computed with cyclo-static dataflow graphs, while leveraging the additional information to obtain smaller buffers (Wiggers et al. 2007a,c). For example, with the algorithm from (Wiggers et al. 2007c), we compute using the cyclo-static dataflow graph from Figure 7.2 that a buffer capacity of 1536 is sufficient for the buffer between the MP3 decoder and the sample rate converter and that a buffer capacity of 90 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. However, currently, the abstraction from firings to aggregate firings in the algorithm for VPDF graphs looses this additional information, and basically transforms a cyclo-static dataflow graph in a multi-rate dataflow graph. We do not see this as a fundamental limitation of the algorithm for VPDF graphs and expect that future work can improve the accuracy of the algorithm.

Interestingly, the VPDF model as shown in Figure 7.3 leads to more accurate results than the model as shown in Figure 7.2. In Figure 7.3, we let actor v_{SRC} have a single phase in which it consumes 48 tokens and produces p tokens, where p can attain either the value 45 or 44. The firing duration that we associate with this actor is the estimated worst-case execution time of an execution of the sample rate converter that consumes 48 samples per execution and equals 136577 clock cycles, which is the maximum execution time that we observed for our input stream. With this VPDF graph, the algorithm from Chapter 6 computes that a buffer capacity of 1578 is sufficient for the buffer between the MP3 decoder and the sample rate converter and a buffer capacity of 72 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. These buffer capacities are confirmed to be sufficient by our cycle-accurate simulation.

In (Moonen et al. 2007), two sources of inaccuracy of multi-rate and cyclo-static dataflow modelling were identified in case tasks are modelled that execute in isolation. One source of inaccuracy is the use of worst-case execution times. The other source of inaccuracy is the fact that dataflow actors consume tokens at their start and produce tokens at their finish, while tasks consume and produce containers during their execution. The variation in execution times of our tasks is relatively limited, for the MP3

	MP3 – SRC	SRC – DAC
Buffer capacity computation	2272	710
Exact buffer capacity derivation	1152	65
Simulation	1152	45
CSDF specific algorithm	1536	90
Alternative VPDF model	1578	72

Table 7.3: Buffer capacities as already computed in Experiment 2, now together with discussed results obtained with a CSDF specific buffer capacity algorithm (Wiggers et al. 2007c) and the alternative VPDF model from Figure 7.3.

decoder we observed a 4% difference between the minimal and maximal observed execution time, while the sample rate converter has a difference of less than 1%, both when it processes 480 input samples as well as when it processes 48 input samples.

We expect that the second source of inaccuracy that is identified in (Moonen et al. 2007) has a larger effect on the accuracy of our model. We modelled an MP3 decoder where space is acquired at the start of an execution and data is released at the end of the execution. This is a correct model in the sense that it is conservative. However, the implementation of the MP3 decoder communicates its data using a write primitive, which means that the acquisition of space and the release of data follow each other very closely. A model in which this early release of data is captured would lead to more accurate results. We could envision a cyclo-static dataflow actor that has one phase in which tokens are consumed and produced and one phase in which no tokens are consumed nor produced, where the first phase models the write primitive and the second phase models the execution of the subsequent code. However, our definition of non-blocking code-segments and our required relation between non-blocking code-segments and actor firings does not allow two firings of the same actor to model a single execution of a non-blocking code-segment. We expect that relaxing this requirement between task graph and dataflow graph is possible and can lead to more accurate models and analysis results.

7.2 Execution on Shared Resources

In this section, the MP3 decoder and the sample rate converter still execute on their different processors, but now they are both allocated a budget on their processor. We use the same TDM scheduler as used in Section 5.4 on both processors. We first consider the sample rate converter that processes 480 samples per execution. We show that this task graph can be conser-

vatively and accurately modelled with a multi-rate dataflow graph, where the firing durations of the actors correspond with the worst-case response times of the tasks. Subsequently, we consider the sample rate converter that processes 48 input samples per execution and show that a model with response times does not provide useful results. This is because the model does not capture that multiple executions of the sample rate converter can occur in the same TDM time-slice. We show that modelling the effects of run-time scheduling with a latency and a rate parameter does capture the possibility of multiple executions in the same TDM time-slice and leads to more accurate analysis results.

Experiment 3 In this experiment, we consider the configuration of the sample rate converter in which it processes 480 input samples per execution. This task graph is modelled by the multi-rate dataflow graph as shown in Figure 7.1.

Both the MP3 decoder and the sample rate converter now share their processor with another task. The MP3 decoder shares its processor with task w_1 and the sample rate converter shares its processor with task w_2. We allocate the MP3 decoder a time slice of 500000 cycles and task w_1 a time slice of 500000 cycles. Task switching takes time. As described in Section 5.4, in our implementation of a TDM scheduler we have a maximum difference of 98 cycles between the allocated time slice and the resulting budget. The task switches also result in a period that is larger than the sum of the allocated time slices, the period increases with 249 cycles with every task switch. Therefore, with our TDM scheduler, the allocated time slices on the processor with the MP3 decoder result in a budget of 500000 - 98 = 499902 cycles in a period of 1000000 + 2 * 249 = 1000498 cycles for the MP3 decoder. Given an estimated worst-case execution time of 1603621 cycles for the MP3 decoder, we derive a corresponding upper bound on the response time of the MP3 decoder equal to 3606005 cycles, with Lemma 5.1. Theorem 5.1 tells us that a firing duration of 3606005 cycles for actor v_{MP3} is a conservative model given that the estimated worst-case execution time is a conservative upper bound on the execution time. On the other processor, we allocate the sample rate converter a time slice of 675000 cycles and task w_2 a time slice of 325000 cycles. With an estimated worst-case execution time of 1320974 cycles, this results in an upper bound on the response time of the sample rate converter of 1972166 cycles, which we take as the firing duration of actor v_{SRC}.

Our algorithm from Chapter 6 computes that a buffer capacity of 2845 is sufficient for the buffer between the MP3 decoder and the sample rate converter and that a buffer capacity of 836 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. These buffer capacities are guaranteed to satisfy the throughput constraint, given that the digital-to-analog converter starts 7778580 cycles later than the

first start of the tasks. In our cycle-accurate simulator, we verified these buffer capacities and this start time of the digital-to-analog converter. We configured our system such that on one processor first task w_1 obtained its time slice before the MP3 decoder obtained its time slice, while on the other processor first the sample rate converter obtained its time slice. With this configuration, we verified that indeed data arrives in time at the digital-to-analog converter.

The exact approach from (Stuijk et al. 2006a) derives that a buffer capacity of 1824 is sufficient for the buffer between the MP3 decoder and the sample rate converter and that a buffer capacity of 782 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. Simulation in our cycle-accurate simulator with these buffer capacities and for the rest the same configuration as in the previous paragraph confirmed that these buffer capacities are sufficient.

Exploration of sufficient buffer capacities resulted in the following observation given our input stream and the described configuration of the schedulers and a digital-to-analog converter that starts 7778580 cycles later than the first start of the tasks. We found that a buffer capacity of 1536 is sufficient for the buffer between the MP3 decoder and the sample rate converter and that a buffer capacity of 508 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter to let data arrive in time at the digital-to-analog converter for this input stream and for this initial state of the schedulers.

	MP3 – SRC	SRC – DAC
Buffer capacity computation	2845	836
Exact buffer capacity derivation	1824	782
Simulation	1536	508

Table 7.4: Buffer capacities computed in Experiment 3.

Experiment 4 In this experiment, we have exactly the same set-up as in Experiment 3 except that the sample rate converter is configured to process 48 input samples per execution. The sample rate converter is allocated a time slice of 675000 cycles and shares its processor with task w_2 which is allocated a time slice of 325000 cycles. This means that the sample rate converter has a budget of 6749902 cycles in a period of 1000498 cycles. Furthermore, the sample rate converter has the sequence of estimated worst-case execution times: 136577, 133824, 133760, 133750, 133748, 133863, 133844, 133955, 133882, 133862. With Lemma 5.1 this results in a sequence of worst-case response times: 462173, 459420, 459356, 459346, 459344, 459459, 459440, 459551, 459478, 459458. Theorem 5.1 tells that

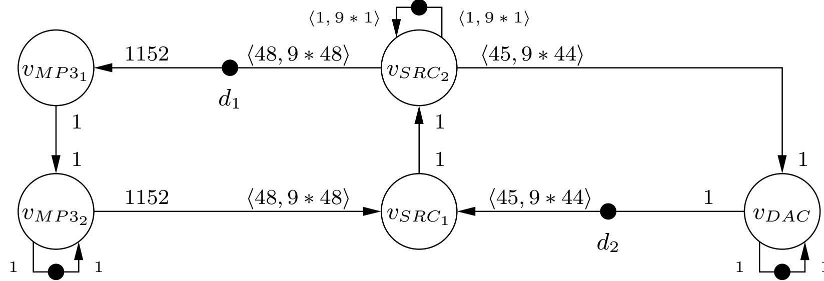


Figure 7.4: CSDF graph with run-time scheduling captured by latency-rate models for both tasks.

associating these worst-case response times with the firing durations of the ten phases of actor v_{SRC} results in a conservative dataflow model. However, actor v_{SRC} now produces 441 tokens in ten phases that cumulatively have a firing duration of 4597025, and therefore produces 1 token every 10425. Because actor v_{DAC} consumes 1 token every 5000, we have that the firing durations of actor v_{SRC} cannot satisfy the throughput constraint.

This analysis result can be understood by realising that the calculation of the worst-case response time assumes that every task execution is enabled just before the task depleted its budget. In this case this means that this calculation assumes that every execution of the sample rate converter first waits for 325596 cycles before it can start. The calculation of worst-case response times, therefore, ignores that multiple executions can subsequently start in the same time slice.

Instead of modelling the effects of run-time scheduling with response times, we will now model the effects of run-time scheduling with a latency and a rate parameter. This results in the cyclo-static dataflow graph as shown in Figure 7.4. Given that the MP3 decoder has an estimated worst-case execution time of 1603621, and a budget R_{MP3} in every period Q_{MP3}, Theorem 5.5 tells us that we obtain a conservative model, if we associate a firing duration of $Q_{MP3} - R_{MP3} = 500596$ cycles with actor v_{MP3_1} and a firing duration of ${}^{Q_{MP3} \cdot 1603621}/_{R_{MP3}} = 3209469$ cycles. Similarly, Theorem 5.5 tells us that we obtain a conservative model, if we associate a firing duration of 325596 with each phase of actor v_{SRC_1} and the following sequence of firing durations with the phases of v_{SRC_2}: 202467, 198386, 198291, 198276, 198273, 198444, 198415, 198580, 198472, 198442.

With the extension to our algorithm as described in Section 6.5, which allows for actors without a self-edge, we compute that a buffer capacity of 2942 is sufficient for the buffer between the MP3 decoder and the sample rate converter and that a buffer capacity of 904 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter. Note

that Figure 7.4 has latency-rate dataflow components as defined in Section 5.3.3, while the extended buffer capacity computation algorithm of Section 6.5 has as input a dataflow graph with a dataflow component with two actors with no self-edge to model the sample rate converter.

While the CSDF graph as shown in Figure 7.4 is a valid input for the analysis from (Stuijk et al. 2008), the current implementation of the accompanying tool SDF3 cannot compute the required number of tokens. This is because, currently, SDF3 assumes a one-to-one relation between tasks and actors, i.e. assumes that the relation between task graph and dataflow graph satisfies Property 4.3. Using our own tool, we found that a buffer capacity of 2016 is sufficient for the buffer between the MP3 decoder and the sample rate converter and a buffer capacity of 175 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter.

Exploration of various buffer capacities in our cycle-accurate simulator resulted in the following observation for this input stream and the described initial state of the schedulers. We found that a buffer capacity of 1152 is sufficient for the buffer between the MP3 decoder and the sample rate converter and that a buffer capacity of 108 is sufficient for the buffer between the sample rate converter and the digital-to-analog converter.

	MP3 – SRC	SRC – DAC
Buffer capacity computation	2942	904
Exact buffer capacity derivation	2016	175
Simulation	1152	108

Table 7.5: Buffer capacities computed in Experiment 4.

Discussion The experiments in this section show that the effects of run-time scheduling can be conservatively modelled in a dataflow graph. Apart from the sources of inaccuracy described at the end of Section 7.1, the initial state of the scheduler is a new source of inaccuracy for the analysis presented in this section (Moonen et al. 2007). While Experiment 3 shows that analysis with response times can be applicable, Experiment 4 describes a realistic case in which analysis of run-time scheduled tasks with response times does not lead to useful results. The subsequent analysis in Experiment 4 of the same run-time scheduled tasks with a model that uses a latency and a rate parameter does lead to useful results. However, the accuracy of the results of the dataflow model that captures the effects of run-time scheduling with a latency and rate parameter is not completely satisfactory. This is because the buffer capacity computation algorithm currently does not leverage the additional information that a cyclo-static

dataflow graph provide, as already discussed at the end of Section 7.1. In Section 5.4, we did not suffer from this limitation in the analysis and the results of that section show that modelling the effects of run-time scheduling with a latency and a rate parameter leads to an accurate model.

7.3 Data-Dependent Inter-Task Synchronisation

In this section, the task graph consisting of the block reader, the MP3 decoder and the digital-to-analog converter are considered. The MP3 decoder processes files encoded at 48 kHz. In this section, the digital-to-analog converter samples the output of the MP3 decoder at 48 kHz. The sample rate converter is omitted from the task graph in this section in order to focus on the main concepts. This task graph includes the block reader. On the FIFO buffer between the block reader and the MP3 decoder, we have that the MP3 decoder has a synchronisation behaviour that is dependent on the processed stream. This means that this task graph cannot be modelled as a cyclo-static dataflow graph. We model this task graph by a variable-rate dataflow graph (Wiggers et al. 2008b) and subsequently by a more accurate variable-rate phased dataflow graph, which is the model that is introduced in Chapter 6 of this thesis.

Again in order to focus on the main concepts, we let the block reader and the MP3 decoder execute on different processors. This is because the inclusion of the effects of run-time scheduling in the dataflow models of this section is analogous to the inclusion in the dataflow models of the previous section. In this section, we let the digital-to-analog converter sample every 5000 cycles. This implies that we now assume that the processors are clocked at 240 Mhz, because the digital-to-analog converter samples at 48 kHz.

Experiment 5 In this experiment, the block reader reads data from a file. The maximum execution time that we observed to produce 2048 bytes equals 3953 cycles. The MP3 decoder has a local input buffer from which it takes its data to produce 1152 samples per MP3 frame. We set the input buffer size to 3000 bytes. The MP3 file that we decode is a variable-bit rate encoded file. Given that an MP3 frame contains 1152 samples and given a sampling frequency of 48 kHz, we have that with a maximum bit-rate of 320 kbit per second the maximum number of bytes per frame equals 960 bytes (MPEG 1992). Therefore, after a number of frames have been decoded there are insufficient bytes in this local input buffer to decode a next frame. At this moment a number of bytes is read from the input FIFO buffer to again fill the local input buffer to hold 3000 bytes. Both the number of frames that is decoded between subsequent refills of this local input buffer as well as the number of bytes that is read per refill depends on

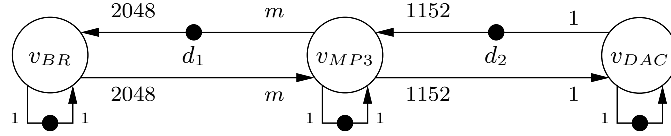


Figure 7.5: VRDF graph of MP3 playback application, with $m \in [0, 3000]$.

the processed data stream. As mentioned before, the estimated worst-case execution time of the MP3 decoder is 1603621 cycles.

In Figure 7.5, a VRDF model is provided of this task graph, where we take the estimated worst-case execution times of the tasks as the firing durations of the actors. In this VRDF graph, the MP3 actor produces 1152 tokens in every firing and consumes a quantum that is in the interval between 0 and 3000. With the algorithm from Chapter 6, we compute that a buffer capacity of 5884 is sufficient for the buffer between the block reader and the MP3 decoder and that a buffer capacity of 1473 is sufficient for the buffer between the MP3 decoder and the digital-to-analog converter, given that the digital-to-analog converter starts 7365650 cycles later than the first start of the block reader. Cycle-true simulation confirmed that data arrives in time at the digital-to-analog converter with these buffer capacities and this start time of the digital-to-analog converter.

The VPDF graph as shown in Figure 7.6 has a more intuitive and accurate model of the MP3 decoder. The VPDF actor that models the MP3 decoder has two phases, where the first phase is always fired once and the second phase is fired n times. Because the maximum number of bytes that is required to decode a frame equals 960 and the local input buffer has a size of 3000 bytes, we have that $n \geq 3$. The first phase models that m bytes are read from the input buffer and no samples are written in the output buffer, m corresponds with the number of bytes that is required to refill the local input buffer of the MP3 decoder. The second phase models that n MP3 frames are decoded given a completely filled local input buffer with a size of 3000 bytes, where each frame results in 1152 samples being written in the output buffer.

The estimated worst-case execution time of the non-blocking code-segment of the MP3 decoder that starts with the read primitive is 156185 cycles. The estimated worst-case execution time of the non-blocking code-segment of the MP3 decoder that starts with the write primitive is 1603621 cycles. We take these estimated worst-case execution times as the firing durations of the corresponding phases of VPDF actor v_{MP3}.

With the algorithm from Chapter 6, we computed that a buffer capacity of 5910 is sufficient for the buffer between the block reader and the MP3 decoder and that a buffer capacity of 5923 is sufficient for the buffer between the MP3 decoder and the digital-to-analog converter, given that the digital-

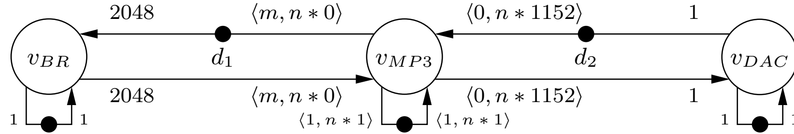


Figure 7.6: VPDF graph of MP3 playback application, with $m \in [0, 3000]$ and $n \geq 3$.

to-analog converter starts 23848900 cycles later than the first start of the block reader. Cycle-true simulation confirmed that data arrives in time at the digital-to-analog converter with these buffer capacities and this start time of the digital-to-analog converter.

For the given MP3 input file, we observed the following in our cycle-true simulator. We found that a buffer capacity of 3300 is sufficient for the buffer between the block reader and the MP3 decoder and that a buffer capacity of 1205 is sufficient for the MP3 decoder and the digital-to-analog converter, given that the digital-to-analog converter starts 7365650 cycles later than the block reader.

	BR – MP3	MP3 – DAC
Buffer capacity computation (VRDF)	5884	1473
Buffer capacity computation (VPDF)	5910	5923
Simulation	3300	1205

Table 7.6: Buffer capacities computed in Experiment 5.

Discussion The task graph of this MP3 playback application cannot be modelled by a cyclo-static dataflow graph, because the MP3 decoder reads a number of bytes from its input buffer that depends on the processed stream. The contribution of this thesis is that it enables to derive buffer capacities that satisfy a throughput constraint for task graphs such as this MP3 playback application. As pointed out by this experiment, the accuracy of our analysis can still be further improved. The VPDF model of Figure 7.6 provides more information than the VRDF model of Figure 7.5, but currently the analysis does not leverage this into more accurate results. On the contrary, the computed buffer capacities computed with the VPDF model are in this case larger than the buffer capacities computed with the VRDF model. This is because the analysis rests on a concept of aggregate firings that aggregates the minimal number of firings of actor v_{MP3} into a single firing that transfers 3000 tokens per firing on the queues to and from

actor v_{BR} and transfers $3 * 1152 = 3456$ tokens per firing on the queues to and from actor v_{DAC}.

7.4 Conclusion

In this chapter, we presented a sequence of experiments that highlighted various aspects of our approach. We started with a task graph consisting of an MP3 decoder, a sample rate converter, and a digital-to-analog converter. This task graph was modelled by a multi-rate dataflow graph, and, subsequently, by a cyclo-static dataflow graph. The tasks were initially executed in isolation on private resources, after which they were executed on resources on which the tasks have guaranteed resource budgets. We verified in a cycle-true simulator that our dataflow analysis approach computes sufficient buffer capacities for this task graph with tasks that are scheduled at run-time. We compared our dataflow analysis approach that computes conservative buffer capacities with an approach that computes exact results. We observed that the run-time of our approach is several orders of magnitude better. The accuracy of our approach is currently not that good, but we highlighted points that can be improved. We expect that the accuracy can be improved and that the low run-time of our approach is an enabling factor for an automated mapping flow.

The second part of this chapter considered a task graph consisting of a block reader, an MP3 decoder, and a digital-to-analog converter. The number of bytes that the MP3 decoder requires in order to produce a given amount of output samples depends on the actually processed data stream. This makes it impossible to model this task graph as a cyclo-static dataflow graph. Instead we model this task graph with variable-rate phased dataflow, which is the new dataflow model presented in this thesis. Comparing the results following from different variable-rate phased dataflow models leads us to expect that, also for task graphs with data-dependent inter-task synchronisation, the accuracy of our buffer capacity computation approach can be improved. We are not aware of alternative approaches that can compute buffer capacities that are sufficient to guarantee satisfaction of timing constraints for this task graph of an MP3 playback application. The extension in expressiveness of our new dataflow model to allow for data-dependent synchronisation behaviour is therefore of great practical interest, and an enabler for an automated mapping flow that is applicable for a broad class of stream processing applications.

Chapter 8

Conclusion

This thesis presents an algorithm that uses a new dataflow model to compute buffer capacities that are guaranteed to satisfy timing and resource constraints for task graphs with tasks that have data-dependent inter-task synchronisation behaviour and that are executed on resources with run-time schedulers that guarantee resource budgets.

Stream processing applications such as digital radio baseband processing and audio or video decoders are often firm real-time embedded systems. For a real-time embedded system, guarantees on the satisfaction of timing constraints are based on a model. This model forms a load hypothesis. In contrast to hard real-time embedded systems, firm real-time embedded systems have no safety requirements. However, firm real-time embedded systems have to be designed to tolerate the situation that the load hypothesis is inadequate. For stream processing applications, a deadline miss can lead to a drastic reduction in the perceived quality, for instance the loss of synchronisation with the radio stream in case of a digital radio can result in a loss of audio for seconds.

We identify four trends for stream processing applications: (1) increasing computational requirements, (2) increasing adaptivity to the environment, (3) increasing integration of stream processing functions, and (4) increasing context dependent resource provisions by hardware architectures, e.g. caches and speculative execution. We address these trends in the following way.

To address the increasing computation requirements of stream processing applications, typically a multiprocessor system is required. On this multiprocessor systems, these applications are implemented as task graphs, with tasks communicating data values over buffers.

We have the trends of an increasing adaptivity to the environment of stream applications and increasing context dependent resource provisions of the architectures on which these applications execute. These two trends make it increasingly difficult to determine a load hypothesis that holds for all cases. Therefore, we introduce our data-driven approach. In our data-driven approach, the interfaces with the environment are time-triggered, while the tasks that implement the functionality are data-driven. This results in a system where guarantees on the satisfaction of the timing constraints can be provided given that the load hypothesis is adequate. If the load hypothesis is inadequate, e.g. non-conservative worst-case execution times, then this cannot result in corrupted data values inside the application, i.e. in the buffers between the data-driven tasks. Furthermore, in our data-driven approach, non-conservative worst-case execution times also do not directly result in corrupted data values at the time-triggered interfaces, because task executions can compensate each other.

Further, stream processing applications increasingly process more independent streams. Typically, timing constraints are specified per stream. We apply on every shared resource a run-time scheduler that by construction guarantees every task a resource budget. These schedulers allow to provide timing guarantees per stream that are only dependent on the load hypothesis for the processing of this stream.

Before we present the contributions of this thesis and an outlook on future research, in Sections 8.2 and 8.3, we first present a summary of the results from the chapters of this thesis in Section 8.1.

8.1 Summary

We concluded in Chapter 1 that the established time-triggered and event-triggered design paradigms for real-time embedded systems are not applicable. This is because time-triggered systems are not tolerant to an inadequate load hypothesis, for example non-conservative worst-case execution times, and event-triggered systems have no temporal isolation from their environment. Our data-driven approach with time-triggered interfaces and data-driven tasks allows to provide guarantees on the satisfaction of timing constraints if the load hypothesis is adequate, and is robust to the case that the load hypothesis is temporarily inadequate.

We concluded in Chapter 2 that dataflow models are applicable to provide guarantees on the satisfaction of timing constraints of stream processing applications implemented in our data-driven approach. However, three limitations of current dataflow models and accompanying analysis techniques were identified. The first limitation is that there is currently not a dataflow model with an accompanying algorithm that computes sufficient buffer capacities that are guaranteed to satisfy timing constraints for task graphs with tasks that consume and produce a number of data

items that can change from one task execution to the next task execution dependent on the processed data stream. The second limitation is that, currently, there is only limited support for the inclusion of the effects of run-time scheduling in dataflow graphs. The third limitation is that, currently, buffer capacities are determined with dataflow analysis by iterating through the possible buffer capacities and for every selection of buffer capacities analyse whether the timing constraints are satisfied. Both the iteration as well as the analysis have an exponential complexity in terms of the graph size. In general, this is problematic, because we envision that the determination of sufficient buffer capacities is an essential kernel of automated programming flows for stream processing applications. These three limitations are addressed by the contributions of this thesis.

In order to guarantee satisfaction of timing constraints, restrictions are required on hardware and software. Chapter 3 presents our restrictions on the implementation of stream processing applications. We required that these applications are implemented as functionally deterministic task graphs that interface with their environment through strictly periodically sampling interfaces. Furthermore, we required that, on each shared resource, run-time schedulers are applied that guarantee resource budgets to tasks.

In Chapter 4, we presented state-of-the-art dataflow simulation and analysis techniques and modelled task graphs with tasks that execute on private resources. We concluded that dataflow simulation can provide guarantees on the satisfaction of timing constraints for any functionally deterministic task graph, for a given input stream and for given buffer capacities. Furthermore, we concluded that dataflow analysis can provide guarantees on the satisfaction of timing constraints for given buffer capacities and for any input stream of task graphs that have inter-task synchronisation behaviour that is independent of the processed data stream. The two essential properties, on which these conclusions rest, are monotonicity and linearity of the temporal behaviour of functionally deterministic dataflow graphs.

Chapter 5 addresses the currently limited support for inclusion of the effects of run-time scheduling in dataflow analysis approaches. This chapter extends the results of Chapter 4 and shows that the effects of run-time scheduling can be conservatively captured in dataflow models, in case of budget schedulers. The effects of budget scheduling can be included in dataflow analysis, because a budget scheduler guarantees a task a minimum resource budget that is independent of the execution times and execution rates of other tasks. Initially in this chapter, the effects of run-time scheduling are captured using response times, while subsequently a model is presented with a latency and a rate parameter. We concluded that capturing the effects of run-time scheduling using response times can result in inaccurate bounds on the temporal behaviour, because multiple subsequent task executions in a single budget are not captured in this model. The model with two parameters, latency and rate, results in accurate bounds

on the temporal behaviour.

Chapter 6 presents an efficient algorithm that computes sufficient buffer capacities to guarantee satisfaction of timing constraints for all input streams of task graphs that have inter-task synchronisation behaviour that depends on the processed stream. This algorithm uses a new dataflow model, called variable-rate phased dataflow. Further, the effects of run-time scheduling can be accurately captured in variable-rate dataflow with a latency and a rate parameter. The presented algorithm directly computes conservative buffer capacities, instead of iterating through possible buffer capacities. This results in a low run-time of the presented algorithm. Every cyclo-static dataflow graph with no auto-concurrency of its actors is a valid variable-rate phased dataflow graph. While the complexity of our algorithm is exponential in the number of parameters, it has a low-degree polynomial complexity for cyclo-static dataflow graphs. With this result the three identified limitations of dataflow analysis are addressed.

An MP3 playback application consisting of publicly available software forms the case study that is presented in Chapter 7. Through cycle-true simulation, we verified that our buffer capacity computation algorithm using variable-rate phased dataflow computes buffer capacities that are sufficient to satisfy the timing constraints of this application. We concluded that our approach is applicable for real-life software, while we are not aware of an alternative approach that can compute sufficient buffer capacities for this application, because the MP3 decoder consumes a number of bytes from its input buffer that depends on the processed data stream.

8.2 Contributions

This thesis presents an algorithm that uses a new dataflow model, variable-rate phased dataflow, to compute buffer capacities that guarantee satisfaction of timing and resource constraints for task graphs that have inter-task synchronisation behaviour that is dependent on the processed data stream and that have tasks that are scheduled by run-time schedulers that guarantee resource budgets. This is an important extension of dataflow analysis techniques, which allows to model a larger class of applications and allows to include the effects of a larger class of run-time schedulers. This is exemplified by the case study with an MP3 playback application, for which we are not aware of alternative approaches to compute buffer capacities that are sufficient to satisfy the timing constraints. Furthermore, we improved the accuracy with which the effects of run-time schedulers that guarantee tasks a minimum resource budget are modelled in dataflow graphs. Instead of capturing the effects of run-time scheduling using response times, we capture these effects with a model that has a latency and a rate parameter. Response times do not capture that multiple task executions can occur subsequently in the same allocated budget. This is captured with

the model with a latency and a rate parameter, which results in improved accuracy of the derived bounds on the temporal behaviour. Further, our algorithm has an attractive computational complexity. Every cyclo-static dataflow graph that is an intuitive model of a task graph is a variable-rate phased dataflow graph, i.e. every cyclo-static dataflow graph in which no actor has any auto-concurrency. The algorithm that computes buffer capacities has a polynomial complexity in the size of the cyclo-static dataflow graph. The validity of our analysis is confirmed by simulation in both a dataflow simulator as well as in a cycle-accurate simulator.

8.3 Outlook

The work presented in this thesis can be extended in the following ways.

- Increasing the expressiveness beyond variable-rate phased dataflow as presented in this thesis. As defined in Chapter 6, variable-rate phased dataflow does not allow its actors to fire concurrently to themself. A limited exception to this restriction has been made in Section 6.5, however, in general, defining the semantics of variable-rate phased dataflow where actors have auto-concurrency is future work. Two other notable restrictions are that variable-rate phased dataflow can only accurately model tasks that do not have a nesting of loops with a parameterised number of loop iterations and that no relations between parameters can be specified. We expect that allowing to model nested loops will be a more controlled extension of the expressiveness than allowing to model relations between parameters.

- Improving the accuracy of the buffer capacity computation algorithm as presented in Chapter 6 that uses variable-rate phased dataflow. In case the number of firings of one iteration through all phases of a variable-rate phased dataflow actor is finite, then all these firings are aggregated into a single aggregated firing. As suggested by the experiments in Chapter 7, finding another abstraction of these firings, then the current aggregate firing can lead to a significant improvement of the accuracy of the computed buffer capacities.

- Construction of a dataflow model given a task graph. Variable-rate phased dataflow has control flow and dataflow unified in single model, which allows for guarantees on the satisfaction of timing constraints over different modes of the application. However, this makes the control flow relatively implicit in the models. Further, our current requirements on the relation between task graph and dataflow graph already allow for multiple dataflow models for a single task graph. Constructing a correct variable-rate phased dataflow model of a task graph can, therefore, in some cases be non-intuitive or even challenging. Construction of a variable-rate phased dataflow model from

other models in which for instance the control flow is more explicit or even directly from the application code is an interesting line of future work.

- Designing an automated mapping flow for stream processing applications on multiprocessor systems. As discussed in Chapter 1, buffer capacity computation is envisioned as an essential kernel in such an automated mapping flow. Despite initial steps (Stuijk 2007), there remains a lot of research to be done on the derivation of scheduler settings, execution times, task to processor binding, and a partitioning such that timing and resource constraints are satisfied. Currently, the presented buffer capacity computation algorithm is already part of a design flow for resource allocation in a network-on-chip (Hansson et al. 2009b) and in an approach that adds the required inter-task synchronisation to a partitioned application (Bijlsma et al. 2008).

- Exploration of the class of budget schedulers. The only example of a budget scheduler that this thesis discusses in detail is time division multiplex scheduling. Recent work introduced a new budget scheduler called priority-based budget scheduling (Steine et al. 2009). We expect that many variants of this scheduler exist that each have different properties.

- Evaluation of applicability of the approach presented in this thesis to other programming models, i.e. other memory consistency models. This thesis shows the applicability of dataflow modelling for tasks that communicate over FIFO buffers with streaming consistency as the assumed memory consistency model. More research is required to evaluate the applicability of dataflow modelling for other programming models, for instance tasks that communicate using the POSIX thread library (Pthread 1995) and have release consistency as the assumed memory consistency model.

We hope that this thesis inspires and enables future research in the field of real-time embedded multiprocessor systems.

List of Symbols

T	Task graph	31
W	Set of tasks	31
B	Set of buffers	31
R	Budget	43
Q	Time interval in which budget is guaranteed	43
G	Dataflow graph	54
V	Set of actors	61
E	Set of queues	61
ρ	Firing duration	61
δ	Number of initial tokens on a queue	61
$a(c)$	Arrival time of container c	57
$\hat{a}(c)$	Arrival time of token c	57
$e(w,i)$	External enabling time of execution i of task w	58
$\hat{e}(v,i)$	External enabling time of firing i of actor v	58
$f(w,i)$	Finish time of execution i of task w	58
$\hat{f}(v,i)$	Finish time of firing i of actor v	58
dst	Destination of an edge	60
src	Source of an edge	60
$\mathbb{N}$	Set of non-negative natural numbers	60
$\mathbb{N}^*$	Set of positive natural numbers	61
$\mathbb{R}$	Set of real numbers	60
$\mathbb{R}^+$	Set of non-negative real numbers	60
σ	Schedule of actor firings	54
μ	Maximum cycle mean	61

$x(w,i)$	Execution time of task execution i	64
$\hat{x}(w)$	Upper bound on execution times of task w	64
$\hat{x}(w,i)$	Upper bound on execution times of non-blocking code-segment i of task w	68
$r(w,i)$	Upper bound on response time of task execution i of task w	75
K	Set of dataflow graph components	86
γ	Parameterised token consumption quantum	112
π	Parameterised token production quantum	112
θ	Number of phases	112
χ	Parameterised number of firings of a phase	112
ϕ	Set of values associated with a parameter	112
P	Set of parameters	112
$\Gamma(e)$	Parameterised cumulative token consumption quantum on queue e	113
$\Pi(e)$	Parameterised cumulative token production quantum on queue e	113
z_i	Parameterised repetition rate of actor v_i	113
φ	Parameter communicated on this queue	115
S	Set of shared parameters	134
$\hat{r}(e)$	Maximum required token transfer rate on queue e	122
$\Gamma(f,e)$	Token consumption quantum of aggregate firing f on queue e	121
$\hat{\Gamma}(e)$	Maximum cumulative consumption quantum on queue e	121
$\Pi(f,e)$	Token production quantum of aggregate firing f on queue e	121
$\Upsilon(v_i,f)$	Firing duration of aggregate firing f of actor v_i	121
$\hat{\Upsilon}(v_i)$	Maximum firing duration of an aggregate firing of actor i	121
$s(v_i)$	Start time of first aggregate firing of actor v_i	127
$s(f,\sigma)$	Start time of aggregate firing f of actor v_i given schedule σ	127

$\sigma(v_i, e)$	Constructed queue-schedule of actor v_i on queue e	132
$\sigma(v_i)$	Actor-schedule of actor v_i	132
$\check{\alpha}_c(x, e, \sigma)$	Lower bound on consumption time of token x on queue e given schedule σ	126
$\hat{\alpha}_p(x, e, \sigma)$	Upper bound on production time of token x on queue e given schedule σ	126
$\Lambda(f, e)$	Cumulative token consumption quantum on queue e in aggregate firings one up to and including f	128
$\Lambda_0(n, f, e)$	Cumulative token consumption quantum by firings 1 up to and including firing n of aggregate firing f on queue e, given parameter values $P_0(v_i, f)$	152
$\Lambda(n, f, e)$	Cumulative token consumption quantum by firings 1 up to and including firing n of aggregate firing f on queue e	152
$\Xi(f, e)$	Cumulative token production quantum on queue e in aggregate firings one up to and including f	126
$\Xi_0(n, f, e)$	Cumulative token production quantum by firings 1 up to and including firing n of aggregate firing f on queue e, given parameter values $P_0(v_i, f)$	152
$\Xi(n, f, e)$	Cumulative token production quantum by firings 1 up to and including firing n of aggregate firing f on queue e	151
$\Phi_0(n, f)$	Cumulative firing duration of firing 1 up to and including firing n of aggregate firing f given parameter values $P_0(v_i,)$	150
$\Phi(n, f)$	Cumulative response time of firing 1 up to and including firing n of aggregate firing f	150
$P(v_i, f)$	Parameter values of aggregate firing f of actor v_i	149
$P_0(v_i, f)$	Reference parameter values of aggregate firing f of actor v_i	149
$\bar{P}(v_i)$	Upper bound on reference parameter values of aggregate firings of actor v_i	153
$s(n, f, e)$	Start time of firing n of aggregate firing f on queue e	151
$s_0(n, f, e)$	Reference start time of firing n of aggregate firing f on queue e	151

$\bar{\Gamma}(e_{ij})$	Cumulative token consumption quantum on queue e_{ij} given parameter values $\bar{P}(v_i)$	156
$\bar{\Upsilon}(v_i)$	Cumulative firing duration of actor v_i given parameter values $\bar{P}(v_i)$	153
$\hat{\rho}(v_i, e)$	Maximum firing duration of phases that can have additional firings relative to reference parameter values	154
$\hat{\gamma}(e_{ij})$	Maximum token consumption quantum of phases that can have additional firings relative to reference parameter values	155

Bibliography

L. Abeni and G. Butazzo. Resource Reservation in Dynamic Real-Time Systems. *Real-Time Systems*, 27(2):123–167, 2004.

R. Agrawal. Performance Bounds for Flow Control Protocols. *IEEE/ACM Transactions on Networking*, 7(3):310–323, June 1999.

B. Åkesson, K.G.W. Goossens, and M. Ringhofer. Predator: a Predictable SDRAM Memory Controller. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, 2007.

B. Åkesson, L. Steffens, E. Strooisma, and K. G. W. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya. Intermediate Representations for Design Automation of Multiprocessor DSP Systems. *Design Automation for Embedded Systems*, 7, 2002.

C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *Journal of Symbolic Computation*, 33:1–12, 2002.

M. J. G. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M . Pastrnak, and J. van Meerbergen. Predictable Embedded Multi-Processor System Design. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2004.

M. J. G. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen.

Dataflow Analysis for Real-Time Embedded Multiprocessor System Design, chapter 15. Dynamic and Robust Streaming Between Connected CE Devices. Kluwer Academic Publishers, 2005.

M. J. G. Bekooij, A. J. M. Moonen, and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. In *Bits & Chips Symposium on Embedded Systems and Software*, 2007. available at http://www.arnomoonen.nl/publications/.

A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. le Guernic, and R. de Simone. The Synchronous Languages 12 Year Later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization.* Athena Scientific, 1997.

B. Bhattacharya and S. S. Bhattacharyya. Consistency Analysis of Reconfigurable Dataflow Specifications. In *Embedded Processor Design Challenges*, Lecture Notes in Computer Science, pages 1–17. Springer, 2002.

B. Bhattacharya and S. S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49(10): 2408–2421, 2001.

T. Bijlsma, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Communication between Nested Loop Programs via Circular Buffers in an Embedded Multiprocessor System. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2008.

G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

F. Boussinot and R. De Simone. The ESTEREL Language. *Proceedings of the IEEE*, 79(9), 1991.

J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model.* PhD thesis, University of California at Berkeley, 1993.

G. C. Butazzo. *Hard Real-Time Computing Systems.* Kluwer, 1997.

P. Caspi. From Synchrony to Asynchrony and Back. In *Proc. Int'l Conference on Embedded Software (EMSOFT)*, 2001.

E. de Castro Lopo. Secret Rabbit Code (libsamplerate). http://www.mega-nerd.com/SRC/, 2009.

T.J. Chaney and C.E. Molnar. Anomalous Behaviour of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, 22(4):421–422, April 1973.

F. Commoner, A. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2001.

R. L. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991a.

R. L. Cruz. A Calculus for Network Delay, Part II: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991b.

D. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann, 1999.

A. Dasdan. Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms. *ACM Transactions on Design Automation of Embedded Systems*, 9(4):385–418, October 2004.

E. A. de Kock. YAPI: Application Modeling for Signal Processing Systems. In *Proc. Design Automation Conference (DAC)*, 2000.

F. Bacelli, G. Cohen, G.J. Olsder, and J-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992. ISBN 0 471 93609 X.

J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications. In *Proc. Int'l Conference on Embedded Software (EMSOFT)*, 2008.

E.G. Friedman. Clock Distribution Networks in Synchronous Digital Integrated Circuits. *Proceedings of the IEEE*, 89(5):665–692, May 2001.

G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved Dataflow Programs for DSP Computation. In *Proc. Int'l Conference on Acoustics, Speech, and Signal Processing*, 1992.

A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. Application of Concurrency to System Design (ACSD)*, June 2006.

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. Int'l Symposium on Computer Architecture*, pages 15–26, 1990.

A. Girault, B. Lee, and E. A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, 1999.

R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2), 1969.

W. Haid and L. Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9), 1991.

A. Hansson, K. G. W Goossens, M. J. G. Bekooij, and J. Huisken. CoMPSoC: A Template for Composable and Predictable Multi-Processor System on Chips. *ACM Transactions on Design Automation of Embedded Systems*, 14(1), 2009a.

A. Hansson, M. H. Wiggers, A. J. M. Moonen, K. G. W. Goossens, and M. J. G. Bekooij. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *IET Computers & Digital Techniques*, 2009b. to appear.

T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 2007.

T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A Time-triggered Language for Embedded Programming. *Proceedings of the IEEE*, 91(1): 84–99, 2003.

M. Jersak, K. Richter, and R. Ernst. Performance Analysis of Complex Embedded Systems. *International Journal of Embedded Systems*, 1(1-2): 33–49, 2005.

B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic Dependencies in Modular Performance Analysis. In *Proc. Int'l Conference on Embedded Software (EMSOFT)*, 2008.

G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of Information Processing*, 1974.

G. Kahn and D. B. MacQueen. Coroutines and Networks of Parallel Processes. In *Proc. of Information Processing*, 1977.

B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2000.

H. Kopetz. Event-Triggered versus Time-Triggered Real-Time Systems. In *Proc. Int'l Workshop on Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101, 1991.

H. Kopetz. *Real-Time Systems: Design Principles for Distibuted Embedded Applications*. Kluwer Academic Publishers, 1997.

J. Y. Le Boudec. Appplication of Network Calculus to Guaranteed Service Networks. *IEEE Transactions on Information Theory*, 44(3), 1998.

J. Y. Le Boudec and P. Thiran. *Network Calculus: a Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9), 1991.

E. A. Lee. Absolutely Positively on Time: What Would It Take? *IEEE Computer*, July 2005.

E. A. Lee. Recurrences, Iteration, and Conditionals in Statically Scheduled Block Diagram Languages. In R. W. Brodersen and H. S. Moscovitz, editors, *VLSI Signal Processing III*, pages 330–340, 1988.

E. A. Lee. Consistency in Dataflow Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, 1991.

E. A. Lee and S. Ha. Scheduling Strategies for Multi-Processor Real-Time DSP. In *Proc. IEEE Global Telecommunications Conference and Exhibition (GLOBECOM)*, November 1989.

E. A. Lee and D. G. Messerschmitt. Synchronous Dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrms: Modularity vs. Code Size. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2009.

A. Maxiaguine, Y. Zhu, S. Chakraborty, and W. Wong. Tuning SoC Platforms for Multimedia Processing: Identifying Limits and Tradeoffs. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

A. J. M. Moonen, R. van den Berg, M. J. G. Bekooij, H. Bhullar, and J. van Meerbergen. A Multi-Core Architecture for In-Car Digital Entertainment. In *Proc. Int'l Conference on Global Signal Processing (GSPx)*, 2005.

A. J. M. Moonen, M. J. G. Bekooij, R. van den Berg, and J. van Meerbergen. Practical and Accurate Throughput Analysis with the Cyclo Static Dataflow Model. In *Proc. Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2007.

O. M. Moreira and M. J. G. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

O. M. Moreira, J. D. Mol, M. J. G. Bekooij, and J. van Meerbergen. Multiprocessor Resource Allocation for Hard real-time Streaming with a Dynamic job-mix. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 332–341, March 2005.

O.M. Moreira, F. Valente, and M.J.G. Bekooij. Scheduling Multiple Independent Hard-Real-Time Jobs on a Heterogeneous Multiprocessor. In *Proc. Int'l Conference on Embedded Software (EMSOFT)*, 2007.

MPEG. Coding of moving pictures and associated audio for digital storage media at up to 1.5 Mbit/s, part 3: Audio. International Standard IS 11172-3, ISO/IEC JTC1/SC29 WG11, 1992.

S. Neuendorffer and E. A. Lee. Hierarchical Reconfiguration of Dataflow Models. In *Proc. ACM/IEEE Int'l Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2004.

A. Nieuwland, J. Kang, R. Gangwal, O. P. Sethuraman, N. Busa, K. G. W. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. *Design Automation for Embedded Systems*, 7(3), 2002.

M. Pankert, O. Mauss, S. Ritz, and H. Meyr. Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis. In *Proc. Int'l Conference on Acoustics, Speech, and Signal Processing*, 1994.

G. Papadopoulos and F. Arbab. Coordination Models and Languages. In M. Zelkowitz, editor, *Advances in Computers—The Engineering of Large Systems*, pages 329–400. Academic Press, 1998.

T.M. Parks. *Bounded Scheduling of Dataflow Processes.* PhD thesis, University of California at Berkeley, December 1995.

J. L. Pino and E. A. Lee. Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors. In *Proc. Int'l Conference on Acoustics, Speech, and Signal Processing*, May 1995.

Pthread. *The POSIX Threads Standard, ISO/IEC 9945-1:1996, also known as ANSI/IEEE POSIX 1003.1-1995*, 1995.

R. Reiter. Scheduling Parallel Computations. *Journal of the ACM*, 15(4): 590–599, October 1968.

M. Sen, S. S. Bhattacharyya, L. Tiehan, and W. Wolf. Modeling Image Processing Systems with Homogeneous Parameterized Dataflow Graphs. In *Proc. Int'l Conference on Acoustics, Speech, and Signal Processing*, 2005.

L. Sha, T. Abdelzaher, K. Arzen, A. Cervin, T. Baker, A. Burns, G. Butazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Journal of Real-Time Systems*, 28: 101–155, 2004.

B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

M Spuri and G. C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Journal of Real-Time Systems*, 10(2), 1996.

S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker Inc., 2000.

M. Steine, M. J. G. Bekooij, and M. H. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Proc. Euromicro Conference on Digital System Design (DSD)*, 2009.

D. Stiliadis and A. Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, October 1998.

M.T.J. Strik, A.H. Timmer, J. van Meerbergen, and G.J. van Roostelaar. Heterogeneous Multi-Processor for the Management of Real-Time Video and Graphics Streams. *IEEE Journal of Solid State Circuits*, 35(11): 1722–1731, 2000.

S. Stuijk. *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Eindhoven University of Technology, 2007.

S. Stuijk, M. Geilen, and T. Basten. Exploring Trade-Offs in Buffer Requirements and Throughput Constraints for Synchronous Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, 2006a.

S. Stuijk, M. Geilen, and T. Basten. Sdf3: Sdf for free. In *Int'l Conference on Application of Concurrency to System Design (ACSD)*, 2006b.

S. Stuijk, M. Geilen, and T. Basten. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, 2007.

S. Stuijk, M. C. W. Geilen, and T. Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, October 2008.

SWARM. SoftWare ARM. www.cl.cam.ac.uk/∼mwd24/phd/swarm.html, 2003.

L. Thiele and R. Wilhelm. Design for Timing Predictability. *Real-Time Systems*, 28, 2004.

L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

Underbit Technologies. Mad: Mpeg Audio Decoder. http://www.underbit.com/products/mad/, 2009.

J.W. van den Brand and M.J.G. Bekooij. Streaming Consistency: a Model for Efficient MPSoC Design. In *Digital Systems Design, Euromicro Symposium on*, pages 27–34, 2007.

P. van der Wolf, E. A. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and Programming of Embedded Multiprocessors: an Interface-Centric Approach. In *Proc. ACM/IEEE Int'l Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS)*, September 2004.

W. F. J. Verhaegh, E. H. L. Aarts, P. C. N. van Gorp, and P. E. R. Lippens. A Two-Stage Solution Approach to Multidimensional Periodic Scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1185–1199, October 2001.

P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-Dynamic Dataflow. In *Proc. Workshop on Parallel and Distributed Processing*, 1996.

M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Multi-Rate Real-Time Systems with Back-Pressure. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006.

M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Real-Time Systems with Back-Pressure. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007a.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling Run-Time Arbitration by Latency-Rate Servers in Dataflow Graphs. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, April 2007b.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, June 2007c.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Computation of Buffer Capacities for Throughput Constrained and Data-Dependent Inter-Task Communication. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2008a.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2008b.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer Capacity Computation for Throughput-Constrained Modal Task Graphs. *ACM Transactions in Embedded Computing Systems*, 2009. to appear.

List of Publications

M. J. G. Bekooij, M. H. Wiggers, and J. van Meerbergen. Efficient Buffer Capacity and Scheduler Setting Computation for Soft Real-Time Stream Processing Applications. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, April 2007.

A. Hansson, M. H. Wiggers, A. J. M. Moonen, K. G. W. Goossens, and M. J. G. Bekooij. Applying Dataflow Analysis to Dimension Buffers for Guaranteed Performance in Networks on Chip. In *Proc. Int'l Symposium on Networks-on-Chip*, 2008.

A. Hansson, M. H. Wiggers, A. J. M. Moonen, K. G. W. Goossens, and M. J. G. Bekooij. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *IET Computers & Digital Techniques*, 2009. to appear.

M. Steine, M. J. G. Bekooij, and M. H. Wiggers. A Priority-Based Budget Scheduler with Conservative Dataflow Model. In *Proc. Euromicro Conference on Digital System Design (DSD)*, 2009.

M. H. Wiggers, N. K. Kavaldjiev, G. J. M. Smit, and P. G. Jansen. Architecture Design Space Exploration for Streaming Applications through Timing Analysis. In *Proc. Communicating Process Architectures (WoTUG-28)*, 2005.

M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Multi-Rate Real-Time Systems with Back-Pressure. In *Proc. ACM/IEEE Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006.

M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Real-Time Systems with Back-Pressure. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2007a.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling Run-Time Arbitration by Latency-Rate Servers in Dataflow Graphs. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, April 2007b.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Proc. Design Automation Conference (DAC)*, June 2007c.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Computation of Buffer Capacities for Throughput Constrained and Data-Dependent Inter-Task Communication. In *Proc. Design, Automation and Test in Europe (DATE)*, March 2008a.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2008b.

M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Buffer Capacity Computation for Throughput-Constrained Modal Task Graphs. *ACM Transactions in Embedded Computing Systems*, 2009. to appear.