Universidade de Aveiro
Departamento de
Electrónica, Telecomunicações e Informática,
2013

João Pedro
Nogueira Bastos

Análise de Prioridade Fixa em Multiprocessadores de Tempo-Real
Fixed Priority Analysis for Real-Time Multiprocessors

João Pedro Nogueira Bastos

Análise de Prioridade Fixa em Multiprocessadores de Tempo-Real
Fixed Priority Analysis for Real-Time Multiprocessors

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e de Telecomunicações, realizada sob a orientação científica do Dr. Paulo Bacelar Reis Pedreiras, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Dr. Orlando Moreira, Principal DSP Systems Engineer na Ericsson (Eindhoven).

Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

o júri / the jury

presidente / president

Professor Doutor Ernesto Fernando Ventura Martins
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Paulo José Lopes Machado Portugal
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

Doutor Orlando Miguel Pires dos Reis Moreira
Principal DSP Systems Engineer, Ericsson

agradecimentos / acknowledgements

I would like to thank everyone that has supported and help me throughout my academic endeavor. Unfortunately it is impossible to mention everyone, but I will try to address all the people that had a direct impact in my final project.

First, I want to thank Orlando Moreira for allowing me the opportunity to come to the Netherlands and making me feel very comfortable on my first work environment. For his patience and availability to help and guide me throughout my work. For his easy-going personality, sense of humor and never ending interesting topics of conversation. I trully appreciate his teachings and friendship.

I wish to thank Professor Paulo Pedreiras for his help in my preparation for this project. It was only so that I felt comfortable and at ease to go on this adventure. For his friendship, trust and support during all stages of this work.

I wish to thank Hrishikesh and Yu Yi for helping me settling in and for being my private printer and Heracles support team. To Alok for his sharpness, incredible patience and tutoring. To all of them for the cheerful and always interesting coffee breaks, lunches and procrastination pauses. To Cupertino for his curious personality and for understanding my slightly hypochondriac condition. And to all the folks at Ericsson for the good and relaxed company environment, it was really a pleasure.

To my parents, for all the unconditional support and love in all good and bad moments of my work and exile. For always knowing what to say and, although annoying sometimes, for constantly reminding me of my mistakes and how to learn from them.

To all the people I am fortunate to call my friends, for the moments, the growing up together and for being always there when I need. For those I met during my stay in the Netherlands, thank you for the amazing reception and for pulling me away from work, even when I shouldn't. And of course, a special thank you to my six Oceans mates, *obrigado malta*!

Thank you all.

Abstract

MultiProcessor Systems-on-chip (MPSoCs) are versatile and powerful platforms designed to fit the needs of modern embedded applications, such as radios and audio/video decoders. However, in a MPSoC running several applications simultaneously, resources must be shared while the timing constraints of each application must be met.

Embedded streaming applications mapped on a MPSoC, are often modeled using dataflow graphs. Dataflow graphs have the expressivity and analytical properties to naturally describe concurrent digital signal processing applications. Many scheduling strategies have been analyzed using dataflow models of applications, such as Time Division Multiplexing (TDM) and Non-preemptive Non-blocking Round Robin (NPNBRR). However, few approaches have focused on a preemptive Fixed Priority (FP) scheduling scheme.

Fixed Priority scheduling is a simple and often used scheduling scheme. It is easy to implement in any platform and it is quite predictable under overload conditions.

This dissertation studies the temporal analysis of a set of dataflow modeled applications mapped on the same resources and scheduled with a fixed priority scheme. Our objective is to improve the existing analysis for Single Rate Dataflow (SRDF) graphs and develop the necessary concepts and techniques to extend it for applications modeled with state-of-art dataflow flavor, Mode Controlled Dataflow (MCDF). MCDF is a more suitable model than SRDF, since it can capture the natural dynamic behavior of modern streaming applications, and therefore, reduce the gap between model and real application.

To reach our main objective we present two contributions: improvement of the existing fixed priority scheduling analysis for SRDF and a complete temporal analysis technique for MCDF graphs.

We propose a novel method, for a general case of a set of n graphs, to determine the worst-case response time of a low priority task by characterizing the worst-case load that higher priority tasks generated on the processor. We also demonstrate, that in the case of graphs that exhibit a single dominant periodic/sporadic source, it is possible optimize the analysis for tighter results. We validate and compare our analysis with the current state-of-art technique for periodic streaming applications, and conclude that, for all the experimented graphs, our analysis always provides tighter results.

Furthermore, we propose an implementation of a complete and optimal temporal analysis technique for MCDF graphs that is based on a finite state machine description of the graphs dynamic behavior. We propose solutions to include in the analysis specific properties of MCDF graph, such as pipelining execution and intermodal dependencies. Despite being limited to time bounded and strongly connected graphs, results obtained using this analysis are as good or better than any currently used temporal analysis technique.

Resumo

Os sistemas multiprocessadores (MultiProcessor Systems-on-Chip (MPSoCs)) são plataformas versáteis e potentes desenhadas especialmente para satisfazer as necessidades de aplicações de *streaming*, como rádios e descodificadores de áudio/vídeo. Contudo, quando se executam várias aplicações em simultâneos num MPSoC, os recursos têm de ser partilhados entre aplicações, ao mesmo tempo que os requisitos temporais de cada aplicação têm de ser cumpridos.

As aplicações de *streaming* mapeadas num MPSoC, são usualmente modeladas usando modelos de computação *dataflow*. Estes modelos possuem a expressividade e propriedades analíticas necessárias para representar idealmente aplicações de processamento digital de sinal.

Até hoje, foram analisados alguns esquemas de escalonamento em modelos de computação *dataflow*, como Time Division Multiplexing (TDM) e Non-preemptive Non-blocking Round Robin (NPNBRR). No entanto, poucas tentativas foram feitas no sentido de estudar escalonamentos de prioridade fixa.

O escalonamento de prioridade fixa é simples e popular, visto que é de fácil implementação e o seu comportamento em condições de sobrecarga é previsível.

Esta dissertação estuda a análise temporal de um conjunto de aplicações modeladas com grafos *dataflow*, quando mapeadas nos mesmos recursos e escalonadas com um esquema de prioridade fixa. O nosso objectivo é melhorar a análise existente para grafos do tipo *Single Rate Dataflow* (SRDF) e desenvolver os conceitos e as técnicas necessárias para realizar esta análise utilizando um tipo de *dataflow* de estado-de-arte, o *Mode Controlled Dataflow (MCDF)*. Os grafos MCDF são mais adequados como modelos de aplicações *streaming*, visto que conseguem capturar o seu comportamento dinâmico, e consequentemente, reduzir a diferença entre modelo e aplicação real.

Para atingir o objectivo apresentamos duas contribuições: melhoramos a análise de escalonamentos de prioridade fixa existente para grafos SRDF e apresentamos uma técnica de análise temporal completa para grafos MCDF.

Nesta dissertação apresentamos um método inovador, para o caso genérico de n grafos, para determinar o tempo de resposta máximo de uma tarefa de baixa prioridade, através da caracterização de carga máxima imposta num processador pelas tarefas de mais alta prioridade. Demonstramos ainda, que no caso particular de grafos com fontes dominantes, periódicas ou esporadicamente periódicas, é possível optimizar a análise para obter melhores resultados. A análise é ainda validada e comparada com o actual estado-de-arte, constatando-se que para todos os testes realizados a nossa análise apresenta melhores resultados.

Propomos ainda, uma implementação de uma análise temporal completa e óptima para grafos de dataflow do tipo MCDF, baseada na descrição do comportamento dinâmico dos grafos através de uma maquina de estados finitos. Apresentamos soluções para incluir na análise características especificas dos grafos MCDF, como execuções paralelas de tarefas e dependências intermodais. Apesar de limitada a grafos fortemente ligados, os resultados obtidos são iguais ou melhores que os de análises utilizadas actualmente.

Contents

List of Figures

List of Tables

List of Acronyms

BFS Breadth First Search

$CSDF$ Cyclo Static Data Flow

DDF Dynamic Data Flow

DES Discrete Event Systems

DFS Depth First Search

FPS Fixed Priority Scheduling

FSM Finite State Machine

$FSM - MCDF$ Finite State Machine Mode-Controlled Data Flow

$FSM - SADF$ Finite State Machine Scenario-Aware Data Flow

MC Mode Controller actor

$MCDF$ Mode-Controlled Data Flow

MCM Maximum Cycle Mean

MoC Model of Computation

$MPSoCs$ MultiProcessor Systems-on-Chip

$MRDF$ Multi Rate Data Flow

$NPNBRR$ Non-Preemptive Non-Blocking Round Robin

$ROSPS$ Rate-Optimal Static Periodic Schedule

$SADF$ Scenario-Aware Data Flow

SDF Synchronous Data Flow

SDT Static Dataflow Techniques

SPS Static Periodic Schedule

$SPS - AP$ Static Periodic Schedule AProximation

$SRDF$ Single Rate Data Flow

STS Self-Timed Schedule

TDM Time Division Multiplexing

$TDSCDMA$ Time Division Synchronous Code Division Multiple Acess

$WCET$ Worst Case Execution Time

WCT Worst-Case Throughput

$WCTS$ Worst-Case Self-Timed Schedule

$WLAN$ Wireless Local Area Network

Chapter 1

Introduction

Nowadays it is practically impossible not to be surrounded by one or more embedded system devices, even if we are not aware of it. Mobile phones, TV remotes, coffee machines or MP3 players are all examples of embedded systems, that can be seen as dedicated computer systems interacting with larger mechanical or electric systems. The worldwide market for embedded systems was valued around 160 billion euros, with growth of 9 percent per year, in 2009 [12]. Part of its growth is due to a high demand for smart embedded devices, such as smartphones [22, 35] or smart domestic utensils [16]. The evolution of the concept of the Internet of Things (IoT) allied with mass social networks, is poised to spur even more growth in the market for embedded systems. Sensors that inform the user via instant messaging that a plant needs watering or kitchen ovens that can be turn on and off via SMS are just a few examples of the perks of new products with direct connectivity to the Internet. Another sector that has been propelling the growth of market share for embedded systems is the automotive market. Almost every control system in a car is done by an individual embedded device: temperature control, electric steering, ignition and cruise control, to enumerate a few.

Multiprocessor technology has become a large part of the embedded system market. MultiProcessor Systems-on-Chip (MPSoCs) embody an important and distinct branch of multiprocessors. They have been designed to fulfill unique requirements of embedded applications, comprised of multiple, usually heterogenous, processing elements with specific functionalities (general-purpose processors, accelerators, vector processors, etc.) in the same chip. Advantages of MPSoCs include cost-savings, performance improvements and lower power consumption.

Multimedia and signal processing applications have been widely implemented using MPSoCs. The availability of multiple different processing units fits the need of such high computational and streaming applications. Most computing of digital signal processing involves two types of processing units: a general-purpose core and a vector processor. Most of the flow control decisions are made by the general-purpose processor, while high computational processes, like vector and matrix operations, are delegated to specific-purpose processors, vector processors.

Multi Radio systems benefit largely from the heterogeneity of MPSoCs. Such systems require several different radio transceivers running simultaneously on the same platform. For example, *smartphones* radios offer various different connectivity options to its users (Wifi, GSM, 3G, 4G, etc.). However, such flexible and complex systems also pose new challenges in

terms of analysis and design techniques. For instances, having multiple applications, running on limited shared resources, with high demands for performance and robustness, requires a highly reliable management of available resources and scheduling of tasks executions.

This dissertation intends to address some of the problems of scheduling real-time streaming applications on a MPSoC. We focus on the use of a fixed-priority scheduling scheme. Despite not being as fair as other scheduling techniques, such as Time Division Multiplexing, fixed-priority scheduling is predictable in overload conditions [10] and can be easily implemented in hardware platforms. Although much has been done in fixed priority scheduling of real-time periodic applications, few attempts have focused on real-time streaming applications, that may have a non-periodic behavior. Therefore, in this dissertation we will develop and improve the necessary methods and tools for the analysis of a fixed-priority assignment scheme for applications mapped on MPSoCs. For this purpose, we will use data flow as a basic model of computation as it fits the application domain.

Dataflow is a natural paradigm for describing digital signal processing applications with concurrency properties and parallel computation [8]. Dataflow models, such as Synchronous Dataflow (SDF), have been proved to offer the necessary properties to model and analyze real-time streaming applications. For instances, checking of deadlock-freedom and temporal analysis techniques. However, many of the applications modeled by this Model of Computation (MoC) have a natural dynamic behavior, which cannot be reproduced with such Dataflow models. Other models, such as Dynamic Dataflow (DDF), are expressive enough to mimic the dynamism of applications, but lack the formal analytical properties that are present in Static Dataflow models, such as SDF. This led to the appearance of new models that could both capture the dynamic behavior of an application and still provide the verifications and analysis which are important to the design and characterization of streaming applications.

Scenario-Aware Dataflow (SADF), [36], and Mode-Controlled Dataflow (MCDF), [28], are two of the state-of-art models in Dataflow MoC. Both offer strict constructs that guarantee the correctness of the model and can reproduce the dynamic behavior of applications, by allowing for different modes of operation within the same model.

For this dissertation our aim was to give the first steps in the analysis of a fixed-priority scheme using state-of-art dataflow model Mode-Controlled Dataflow. Unfortunately, due to lack of time and the meanwhile publishing of more related work, we were not able to meet our final goal. Instead, we focused on creating and improving the necessary tools and concepts to address this problem in the future. In order to do so, we make two major contributions to the state-of-art: develop a technique for temporal analysis of MCDF graphs that is complete and optimal, and improve the work done in [1] for the fixed-priority dataflow model using Single Rate Dataflow.

In the remainder of this chapter we will provide the reader with the fundamental concepts of the problem explored in this dissertation, as well as the state-of-art on fixed-priority analysis techniques on streaming applications.

1.1 Basic Concepts

1.1.1 Streaming Applications

As the denomination implies, a streaming application is an application whose input is a large (potentially infinite) sequence of input data items, a *data stream* [38]. Input data is

fed into the application by an external source and each data item is processed in a limited time before being discarded. The output is also a long (potentially infinite) sequence of data items. Typically applications in this domain have three characteristics:

- **High computational intensity**: High number of arithmetic operations per I/O.
- **Data parallelism**: Data items from the input stream can be processed simultaneously without needing results from previous items.
- **Data locality**: Data is local to the application, once produced and read it is never used again.

As an easy analogy, a dictation exercise can be seen as a streaming application. The teacher will read a text out loud at a certain speed while the student must follow and write down the text on a piece of paper. If the student takes to much time to write down certain words he might not be able to transcript the full text and, therefore, fail the exercise.

Examples of such applications include communication protocols, software-defined radio, audio and video decoding, cryptographic kernels and network processing [28].

1.1.2 Real-Time Applications

Real-time applications are applications that must produce results within certain imposed time constraints. In fact, if the output of such applications violates its temporal constraints it might become irrelevant, wrong or even lead to catastrophic results. For example, the Anti-lock Brake System (ABS) of a car is a real-time application. The ABS system allows the wheels to maintain tractive contact with the road surface, impeding the car from skidding, by implementing cadence braking. Therefore, locking and unlocking of the brakes must be done within strict time intervals. Otherwise, traction is lost and the driver loses control of the car. However, not all real-time application have such harsh consequences for failing to meet their timing constraints: MP3 players, TV receivers or Wifi transceivers are all examples of real-time systems that failing to meet their standard timing requirements will only result in degradation of service for the user. Thus, an important classification of real-time applications is related to how important the timing requirements are. One simple classification, widely used in industrial settings, divides real-time applications in two classes [9]:

- **Hard Real-Time**: Requirements cannot be violated under any circumstances, or the results of the computation will be completely useless, and failure may, in case of life critical systems, have catastrophic consequences.
  Examples: Pacemakers, Fly-by-wire systems, ABS and others.
- **Soft Real-Time**: Requirements can be occasionally disrespected, but the rate of failures must be kept below a certain maximum.
  Examples: Video Streaming, Gesture recognition in a tactile device and others.

In addition, if failure to meet the temporal of functional requirements of an application can jeopardize the safety of human lives or lead to important economical damage, then these

applications are referred as critical real-time applications.

The design of real-time applications is focused on providing different features than regular applications. The focus is not in user-friendliness, performance speed or usability, but instead, the desired features of real-time applications include the following, as in [9]:

- **Timeliness**: Results must be correct not only in their value but also be generated within a specific time interval.

- **Predictability**: The system must be analyzable in such way that all possible consequences of scheduling decisions are predictable.

- **Efficiency**: Resources must be efficiently managed to achieve the desired levels of performance.

- **Robustness**: Real-time systems must not collapse when they are subject to peak-load conditions, so they must be designed to manage all anticipated load scenarios.

In the next sub-sections we explore further the subjects of timing constraints and scheduling of real-time applications.

Timing Constraints

Our area of study focuses on streaming applications and we will define our timing requirements accordingly using throughput and latency constraints, as opposed to following the classical approach of deadline constraints. We define throughput as the rate at which an iterative application produces results and latency as the time between the arrival of input data and the production of the related output data. Applications may have one or both types of constraints. An example of an application with throughput requirements is a video streaming service, like Youtube, Hulu or Netflix. It is important to guarantee a certain rate of frames to the viewer but the time it takes for the signal to travel from the server to the user is quite irrelevant. Meanwhile, a game controller is an example of a latency constrained application. It is fundamental that the latency is respected otherwise the controller's response will be inconsistent. A game controller that fails to meet its latency requirements will certainly affect the performance of the player. In this project we focus on studying applications in terms of worst-case throughput and latency requirements.

We give special attention to Fixed Priority scheduling, since its the objective of this project is to study its analyzability in Dataflow Model of Computation.

1.1.3 Fixed Priority Scheduling

Priority-driven scheduling is a widely used approach to real-time scheduling. Fixed priority scheduling dates back to job-shop scheduling, or manufacturing problems regarding machinery

operation scheduling, and has, since, been improved and widely implemented in industry and computer systems. Tasks are studied at design time and given a priority value according to a chosen parameter (request rate, earliest deadline, etc). Priorities assigned to tasks do not change during executions, hence the term *fixed*. The scheduler, then, picks the next task to be executed in terms of its readiness and priority value. For example if we have two tasks A and B, sharing the same resources, with, respectively, priorities P_a and P_b, with $P_a > P_b$, then any scheduling decision will give preference to A. This known behavior is one of fixed priority's advantages, predictability. In a critical situation, such as a peak of load, it is known that only the lower priority tasks will be affected. This can be used to guarantee that, for example, a smartphone user might lose its Wi-Fi connectivity, but never its calling capabilities. However, fixed priority exchanges predictability for fairness. In a fixed priority scheme the behavior of low priority tasks is always dependent on the behavior of high priority tasks.

We now introduce the concept of preemption and provide the reader with some examples of fixed-priority scheduling algorithms: Rate-Monotonic and Deadline-Monotonic.

Preemption

A scheduler can have the ability to preempt, or interrupt, momentarily a task to give execution to another. In terms of fixed priority scheduling, a scheduler may interrupt a lower priority task to give way to a higher priority task, and resume the lower priority task as soon as the higher priority task finishes. Most fixed priority scheduling techniques, like Rate Monotonic (RM), are intrinsically preemptive and assume a preemptive system. However, some studies have focus on non-preemptive fixed priority scheduling [29]. In this situation, a low priority task might block a higher priority task that arrives during its execution. We focus our study only in preemptive fixed priority schedulers, therefore, throughout this dissertation we always assume availability of preemption.

Rate-Monotonic Scheduling

In this scheduling technique priorities are assigned according to task request rates. Each task is characterized by a period T_i, a phase ϕ_i, a worst-case execution time C_i and a relative deadline D_i. Task with higher request rates (that is, shorter periods) will have higher priorities. Assuming a periodic behavior, Rate Monotonic (RM) [25] is a fixed priority assignment. RM is also intrinsically preemptive, a currently executing task can be preempted by newly arrived tasks with shorter periods.

According to the study done in [25], RM is optimal, meaning that, amongst all fixed priority assignments, no other fixed priority algorithm can schedule a task set that cannot be scheduled by RM. Moreover, it was also found an upper bound on processor utilization for a generic set of n periodic tasks, providing useful tools for *a priori* knowledge of a given set of task schedulability according to its temporal requirements. The two reference criteria for processor utilization based tests are the *Minor Bound* of Liu and Layland [25] and the *Hyperbolic bound* of Bini, Buttazzo and Buttazzo [7].Further insight on the proposed tests and their proofs can be sought in [9].

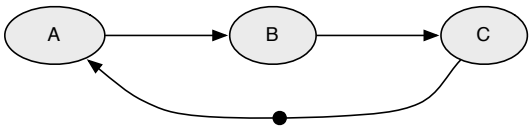

Figure 1.1: Dataflow graph example

Deadline-Monotonic Scheduling

In a Deadline-Monotonic (DM) Schedule tasks are assigned a priority based on their relative deadline. Again, each task is characterized by a period T_i, a phase ϕ_i, a worst-case execution time C_i and a relative deadline D_i. Since relative deadlines are constant, DM is a fixed priority assignment. In DM a task is assigned a fixed priority P_i, inversely proportional to its relative deadline. Thus, at any instant, the task with the shortest relative deadline is executed.

Deadline-Monotonic is optimal for independent tasks and when the period is equal to the deadline, such that any task set schedulable by other fixed priority algorithms is also schedulable by DM. Proof and more detailed explanations of the DM algorithm can be found in [3].

1.1.4 Dataflow Graphs

Dataflow modeling will be used extensively throughout this dissertation and therefore a detailed and formal approach will be address in its own chapter. However, we provide the reader with a brief and informal introduction to Dataflow graphs.

Dataflow is a natural paradigm for describing Digital Signal Processing (DSP) applications with concurrency properties and parallel computation [23]. A dataflow graph is a directed graph where nodes represent computations (or functions) and the arcs represent data paths. It is usual to refer to dataflow graph nodes as **actors** and arcs as **edges**. Whenever the input data of an actor is available in all of its incoming edges, the actor can fire, or in other words, perform its computation. An actor with no input arcs may fire at any time. Because all the behavior of the graph depends on data availability, dataflow programs are said to be *data-driven*. Moreover, actors of a dataflow graph cannot have *side-effects*, i.e. data dependencies amongst actors in a dataflow graphs have to be explicit by a connecting edge. Data is modeled by tokens, or delays, that are represented as a number, or dots, between connecting edges. Connections are conceptually FIFO queues that model communication between actors. Figure 1.1 portrays an example of a dataflow graph with three actors: A, B and C. The initial token in edge C-A allows token A to fire, which will consume the token in the input and output a token in the output edge A-B. B can then fire, and consecutively C will also fire. At this point the graph has returned to its initial state and an iteration has passed.

1.2 Related Work

So far we have introduced all the background needed to understand the work done in this dissertation. In this section we will make a summary of the published related work, until the writing of this dissertation, on fixed-priority scheduling of real-time streaming applications.

Early work on holistic dataflow analysis of embedded multiprocessors systems is presented in [5, 33]. Moreira et al propose a temporal analysis for Time Division Multiplexing (TDM) and its variant with static-ordering in [27] for heterogenous MPSoC. Wiggers et al [40] present a generalized technique to model starvation-free schedulers. Steine et al [34] propose a simple starvation-free (budgeted) variant of fixed priority scheduling for two levels of priority. However, response-modeling of non-starvation-free schedulers, like Fixed Priority Scheduling (FPS), has not received much attention from the dataflow community, even though many embedded processors employ FPS in real-life situations.

In terms of FPS temporal analysis for MPSoC we are aware of following work. Almeida [1] proposed an analysis for fixed-priority scheduling of two levels of priority, using dataflow based techniques. Hausmans et al, [18], proposed a different approach for the subset of periodic streaming applications. In this approach, interference of tasks was computed with a enabling jitter periodic event model, as based on [39]. Results were reached by convergence of upper bounds on worst-case or by unschedulability of the set of tasks.

Furthermore, attempts have been made to apply real-time scheduling to dataflow graphs. Park and Lee [30] studied the non-preemptive rate monotonic scheduling for dataflow application. In contrast this dissertation deals with preemptive fixed priority scheduling. Bamakhrama et al [4] shows that a *Strictly Periodic Schedule* for Cyclo Static Dataflow (CSDF) graphs such that traditional real time analysis techniques may be used for analyzing dataflow graphs. However, these works assume acyclic application graphs with no initial tokens on their edges. Application designers exploit the use of cycles and initial tokens to model buffer constraints between communicating actors, or complex inter-activation dependencies. In real-time calculus [37], event streams are used to capture the dependencies that describe the individual component timings. Another compositional approach, called SymTA/S, for analyzing system level performance is proposed in [20] that supports any standard event models for real-time system analysis. However, both real time calculus and SymTA/S acknowledge the complexity of considering cyclic dependencies. Dataflow can very effectively handle cyclic dependencies. Therefore, making it a popular mean for modeling streaming applications.

1.3 Problem Description

As we have already stated, it is expected from a multiprocessor embedded platform to handle several applications running simultaneously. It is also assumed that applications are real-time and their inputs are data streams. Furthermore, it has been concluded that a fixed-priority assignment can in fact be handled by dataflow analysis and a load analysis technique has been sketched to study task interference under worst-case assumptions. However, it is our belief that improvements can be made to current proposed techniques [1, 18] to better their results. The questions we wish to address are the following: *Given a MPSoC platform and a set of n hard real-time streaming applications scheduled in a fixed-priority scheme, what will be the interference amongst tasks and how will it affect the load in the available*

resources? How can we improve current fixed priority dataflow analysis techniques to optimal results? And how to adapt the current approach to state-of-art models of computation such as Mode-Controlled Dataflow?

The approach taken to reach the sought conclusion will be divided in the following steps:

- Extend current throughput analysis available for Mode-Controlled Dataflow to have an overall worst-case analysis of the model.
- Revisit current fixed-priority analysis for dataflow models, proposed by [1].
- Optimize current load analysis technique for fixed-priority dataflow for periodic bound cases and compare it with other proposed approaches.
- Extend current fixed-priority dataflow analysis to Mode-Controlled Dataflow models.

All the steps taken should have culminated in an attempt to join the optimized load analysis technique with the more realistic dynamic models provided by the use of Mode-Controlled Dataflow as a Model of Computation. However, due to pending issues with the previous fixed priority analysis for Single Rate Dataflow (SRDF) graphs and new work added to the state-of-art, we opted to focus on improving and optimizing the current analysis for fixed priority scheduling of SRDF graphs.

1.4 Contributions

In result of the work done during this dissertation, the contributions to the state-of-art can be summarized in the following points:

- **Dataflow Analysis** Implementation of an overall and optimal throughput analysis for Mode-Controlled Dataflow MoCs, using a finite state machine to model the possible mode transitions in the model and exploring a state-space of a symbolic execution of the model.
- **Fixed Priority Analysis** We revisit the analysis of dataflow models of fixed priority systems, proposed by [1], and optimize the calculation of worst-case response time upper bounds for periodic bound application graphs, strictly or sporadically periodic. We propose a new interference and response time analysis for n Single Rate Dataflow applications with a fixed priority assignment scheme.
- **Extension of the tools available** For the purpose of the research and analysis done in this dissertation we had at our disposition a set of dedicated software tools, namely a dataflow graph simulator and analysis tool. In result of improvements done to the simulation and analysis of fixed-priority dataflow graphs, we add new functionalities and modified already existing ones. We adapted the tools to correctly simulate Mode-Controlled Dataflow graphs and buffer states. Furthermore, we added new modules to convert Dataflow graphs into Max-Plus matrix representation and for the implementation of state-space exploration throughput analysis for MCDF.

Further minor contribution are summarized at the end of each chapter.

1.5 Document Organization

The remainder of this dissertation is organized as follows: in Chapter 2 we review dataflow computational models and their analytical properties. The software framework used throughout this project is introduced in Chapter 3, as well as detailed explanation of the modifications done to the tools. Chapter 4 explains in great detail how throughput analysis on Mode-Controlled Dataflow can be improved by extending the model with the use of a finite state machine and Chapter 5 introduces the proposed technique for fixed-priority dataflow analysis and optimization done for periodic bound applications. Finally, Chapter 6 states our conclusion and suggestions for future work.

Chapter 2

Data Flow Computational Models

As stated in the previous chapters, we use data flow as the base model of computation throughout this dissertation. Although there are many flavors of data flow we focus mainly in three variants: Single Rate Data Flow (SRDF), Mode-Controlled Data Flow (MCDF) and Scenario-Aware Data Flow (SADF). SRDF is the model used in the current fixed-priority dataflow analysis, whilst MCDF and SADF belong to a quite different category of data flow models, to which we wish to extend our fixed-priority analysis. In this chapter, we present the notation and relevant properties of the mentioned data flow models. This is reference material and most of it can be found in [8, 23, 28, 31, 33, 36].

2.1 Graphs

A **directed graph** G is an ordered pair G = (V, E) where V is the set of **vertexes** or **nodes** and E is the set of **edges** or **arcs**. Each edge is an ordered pair (i, j) where $i, j \in V$. If $e = (i, j) \in E$, we say that e is **directed** from i to j. i is said to be the **source node** of e and j the **sink node** of e. We also denote the source and sink nodes of e as $src(e)$ and $snk(e)$, respectively.

2.1.1 Paths and Cycles in a Graph

A **path** in a directed graph is a finite, nonempty sequence $(e_1, e_2, ..., e_n)$ of edges such that $snk(e_i) = src(e_{i+1})$, for i = 1,2,...,$n-1$. We say that path $(e_1, e_2, ..., e_n)$ is **directed from** $src(e_1)$ to $snk(e_n)$; we also say that this path **traverses** $src(e_1), src(e_2), ..., src(e_n)$ and $snk(e_n)$; the path is **simple** if each node is only traversed once, that is $src(e_1), src(e_2)$, ..., $src(e_n)$, $snk(e_n)$ are all distinct; the path is a **circuit** if it contains edges e_k and e_{k+m} such that $src(e_k) = snk(e_{k+m})$,$m \geq 0$; a **cycle** is a path such that the subsequence $(e_1, e_2, ..., e_n)$ is a simple path and $src(e_1) = snk(e_n)$.

2.1.2 Strongly-Connected Components

A directed graph is **strongly connected** if there is a path from each node in the graph to every other node. In a more practical definition this means that there must always exists paths in both directions; a path from a to b and also a path from b to a. The **strongly connected components** of a directed graph G are the set of all maximal strongly connected subgraphs

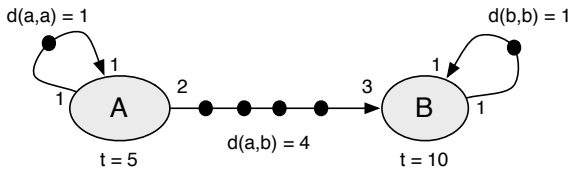

Figure 2.1: Synchronous Dataflow Graph (SDF) example

of G. If we contract each strongly connected subgraph to an equivalent node, the resulting graph is **directed acyclic graph** (DAG), the **condensation** of G.

2.2 Data Flow Graphs

Data Flow graphs are directed graphs where **nodes** are referred to as actors and represent time consuming functions, and edges are referred as **arcs** and represents a data path, commonly modeled as a FIFO queue that directs data from the output of an actor to the input of another. Data is transported in discrete chunks, referred to as **tokens**. In order for data to travel through a data flow graph actors need to fire. The firing of an actor represents the computation of the function associated to that actor and the conditions for an actor to fire are called **firing rules**. As a result of firing an actor, data is **consumed** from the input tokens and **produced** into the output tokens of the fired actor. Different flavors of data flow may have different firing rules or may have different construct rules associated.

All flavors of data flow used in this dissertation are subset or extensions on Synchronous Data Flow (SDF), or Multi Rate Data Flow (MRDF) [23], hence we will start by introducing the notation used with the behavior of SDF graphs. Synchronous Data Flow (SDF) is a data flow model where the firing rules are as follows: the number of tokens produced (consumed) by the actor on each output (input) edge per firing is fixed and known at compile time. We call execution time of an actor to the elapsed time from start to finish of the firing of an actor, and represent it as $t : V \to N_0$; $t(i)$ being the execution time of actor i. Arcs of a SDF graph can be represented by three fields: delay, tokens consumed and tokens produced. The delay $d(i, j) \in N_0$ is the number of initial tokens on arc (i, j). Whilst $prod(e)$ and $cons(e)$ represent, respectively, the constant value of tokens produced by $src(e)$ and consumed by $snk(e)$ in each firing. Therefore, a timed SDF is defined by a tuple $(V, E, t, d, prod, cons)$.

Figure 2.1 depicts a SDF graph with two actors, A and B, and three edges, (a,a), (a,b) and (b,b). Each actor has associated an execution time t, while each edge has an associated production/consumption rate and a delay d of initial tokens.

An **iteration** of an SDF graph occurs when all the initial tokens have travelled the graph and returned to their exact initial position. However, to reach this state some actors must fire different amount of times than others. In a SDF graph with $|V|$ actors numbered from 1 to $|V|$, the column vector of lenght $|V|$ in which each element represents the correct number of times a actor in the graph must fired to completer an iteration, is denominated the **repetition vector** and usually represented by r. Therefore in an iteration an actor fires as many times as indicated by the repetition vector.

Some applications may have tasks that are modeled with varying execution times, therefore, upper and lower bound can be used to model varying execution times of actors. We define $\check{t}$ as the best-case execution time (lower bound) and $\hat{t}$ as the worst-case execution time (upper bound). Consequently such graph is defined by a tuple $(V, E, \check{t}, \hat{t}, d, prod, cons)$.

2.2.1 Single Rate Dataflow

A very important subset of SDF graphs are Single Rate Dataflow (SRDF) graphs. An SDF graph in which, for every edge $e \in E$, it holds that prod(e) = cons(e), is an SRDF graph. Any SDF graph can be converted into an equivalent SRDF graph, each actor i is replaced by $r(i)$ copies of itself, representing a particular firing of the actor within each iteration of the graph. As an example of an SRDF graph, consider the SDF graph depicted in Figure 2.1 only with every port rate of the same value, such that each port consumes and produces the same amount of tokens. Thus, an SRDF graph can be represented as a subset of SDF graphs with characteristic tuple (V, E, d, t).

SRDF graphs have very useful analytical properties. A SRDF graph is **deadlock-free** if and only if there is at least one delay in every cycle [28]. A graph is deadlocked when there is a cyclic dependency where two actors cannot fire because each requires the other one to fire in order to obtain the data required for it to fire itself.

The **cycle mean** of a cycle c in an SRDF graph is a very important concept, and is defined as $\mu_c = \frac{\sum_{i \in N(c)} t_i}{\sum_{i \in N(e)} d_e}$, where $N(c)$ is the set of all nodes traversed by cycle c and $E(c)$ is the set of all edges traversed by cycle c. We can also define the **Maximum Cycle Mean (MCM)** $\mu(G)$ of a SRDF graph.

$$\mu_c = max_{c \in C(G)} \frac{\sum_{i \in N(c)} t_i}{\sum_{i \in N(e)} d_e} \tag{2.1}$$

The MCM of a SRDF graph can be directly related to its throughput, $\frac{1}{\mu(G)}$ is, in fact, equivalent to the maximum throughput achievable. Further explanation and proof of this statement can be found in [28].

2.3 Dataflow Scheduling

In this section we will introduce some of the modeled scheduling algorithm in dataflow that will be used as the basis for most of our analysis and proofs. We also introduce Time Division Multiplexing as a comparison to the scheduling algorithm we wish to study: Fixed Priority scheduling, which we will address in more detail in Chapter 5.

For simplicity sake, we will introduce these scheduling algorithms by using SRDF graphs. For SDF extended approach to these algorithms please refer to [28].

2.3.1 Self-Timed Scheduling

A **Self-Timed Schedule** (STS) of an SRDF graph is a schedule where each actor firing starts immediately when there are enough tokens on all its input edges. The **Worst-Case Self-Timed Schedule** (WCSTS) of an SRDF is the self-timed schedule of an SRDF where every iteration of every actor i takes $t(i)$ time to execute. The WCSTS of an SRDF graph is

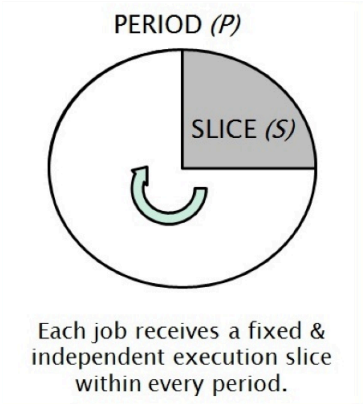

Figure 2.2: Example of a TDM wheel, as in [24]

unique. The WCSTS of a strongly-connected SRDF graph has an interesting property: after a transition phase, of some iterations, it will reach a periodic regime [26, 28]. The number of iterations is a constant for a given timed SRDF. The start time $s(i,k)$, of actor i in iteration k, on a STS schedule can be determine using Equation 2.2.

$$s(i,k) = max_{(x,i)\in E} \begin{cases} s(x, k-d(x,i)) + t(x, k-d(x,i)), & k \geq d(x,i) \\ 0 & k \text{ ¡ } d(x,i) \end{cases} \tag{2.2}$$

2.3.2 Static Periodic Scheduling

A **Static Periodic Scheduler** (SPS) of an SRDF graphs is where all actors fired periodically with constant period T. Therefore, it is true that for all nodes $i \in V$, and all $k > 0$

$$s(i,k) = s(i,0) + T \cdot k \tag{2.3}$$

Note that an SPS can be represented uniquely by a T and the values of $s(i,0), \forall i \in V$.

A SPS schedule only exists if, and only if, $T \geq \mu(G)$, thus $\frac{1}{\mu(G)}$ is the fastest possible rate (or throughput) of any actor in a strongly-connected SRDF [28]. If a SPS has a period T equal to the MCM of the SRDF graph $\mu(G)$, we say that the schedule is a **Rate-Optimal Static Periodic Schedule (ROSPS)**.

2.3.3 Time Division Multiplexing (TDM)

In a Time Division Multiplexing scheduling scheme each task is assigned a fixed amount of time, a **slice**, within a fixed frame, called **replenishment period**. This period is common to all tasks, but the slice sizes may be different for different tasks. Figure 2.2 depicts a simple example of a TDM scheduling scheme, called a **time wheel**.

In data flow, the effect of a TDM scheduling can be modeled by replacing the worst-case execution time of the actor by its worst-case response time under TDM scheduling. The response-time of an actor i is the total time necessary to complete the fire of actor i in an iteration, when resource arbitration is taken in consideration. Assuming that a TDM wheel replenishment period P is implemented on the processor and that a time slice with duration

S is allocated for the firing of i, such that $S \leq P$, a time interval equal or longer than $\theta(i)$ passes from the moment and actor is enabled by the availability of enough input tokens to the completion of its firings. The first of this is the arbitration time, i.e., the time it takes until the TDM scheduler grants execution resources to the actor. In the worst-case, i gets enabled when its time slice has just ended, which means that the arbitration time is the time it takes for the slice of i to start again. If we denote the worst-case arbitration time as $\hat{r}(i)$ then [24]:

$$\hat{r}(i) = P - S \tag{2.4}$$

2.4 Temporal Analysis

Temporal analysis is required in order to verify whether a given timed dataflow graph can meet a required throughput or latency requirement of the application. We assume, due to the application area we focus on, that applications graphs will behave in a self-timed fashion: a data flow actor fires immediately whenever its firing conditions are met. The behavior of a self-timed dataflow graph can be divided in two phases: transient and periodic. In [26] it is proved that the self-timed execution of a a dataflow graph, where actors have constant executions times, will eventually reach a regime with periodic behavior. Until that point, the graph's behavior is on its transient phase. In many of the applications we wish to study, timing requirements need to be guaranteed to have a strictly periodic behavior from the first output on. Therefore the techniques present in this subsection, as proposed in [28], will allow us to reason about the temporal behavior of self-timed execution of data flow graphs, even when considering the transient phase and/or varying execution times per actor firing.

2.4.1 Monotonicity

In a broad sense, a monotonic function can be defined as follows:

Definition 1. *(Monotonicity of a function): A function $f(n)$ is **monotonic increasing** if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is **monotonic decreasing** if $m \leq n$ implies $f(m) \geq f(n)$. A function $f(n)$ is **strictly increasing** if $m < n$ implies $f(m) < f(n)$ and **strictly decreasing** if $m < n$ implies $f(m) > f(n)$ [11].*

But in dataflow we are more interested in the concept of monotonicity of a self-timed execution, such that: if any given firing of an actor finishes execution faster than its worst-case execution time (WCET), then any subsequent firings in any self-timed schedule can never happen later than in the WCSTS, which can be seen as a function that bounds all start times for any self-timed execution of the graph. We can enunciate this as a theorem

Definition 2. *(Monotonicity of a self-timed execution): In a SRDF graph G = (V,E,t,d) with worst-case self-timed schedule s_{WCTS}, for any $i \in V$, and $k \geq 0$, it holds that, for any self-timed schedule s_{STS} of G:*

$$s_{STS}(i,k) \leq s_{WCSTS}(i,k) \tag{2.5}$$

2.4.2 Relation between STS and SPS

Because of monotonicity, there is a very important relation between STS and SPS that allows us to use any SPS start time of an actor as an upper bound to any start of the same firing of the same actor in the WCSTS. That relation is enumerated in the following theorem:

Theorem 1. *In any admissible SPS of an SRDF graph G = (V,E,t,d), all start times can only be later or at the same time than in the WCSTS of that graph, that is, for all $i \in V$, $k \geq 0$, and all admissible static periodic schedules s_{SPS} of G, it must hold that:*

$$s_{WCSTS}(i,k) \leq s_{SPS}(i,k) \tag{2.6}$$

Using the explanation from [28], *"this means that, given a SRDF graph for which we know the worst-case execution times of all actors, we can obtain a linear conservative upper bound on the output production times of any given actor i during self-timed execution. This bound is given by the expression $s(i,0)+t(i)+\mu_G \cdot k$, where $s(i,0)$ is the start time of the first firing of i on a Rate-Optimal Static Periodic Schedule (ROSPS)."*

2.4.3 Throughput Analysis

Definition 3. *(Throughput) The throughput of a graph is the rate at which a graph completes an iteration over an elapsed period of time.*

Previously, we stated that in our application scope and in the paradigm of dataflow modeling, throughput and latency will be the timing requirements we want to analyze and comply with. One way of analyzing the maximum achievable throughput of a certain graph is to, as we have already stated, determine the inverse of the Maximum Mean Cycle of the graph. Another way of finding the throughput of strongly connected graphs is by simulation. The graph is simulated until it reaches a periodic phase, at that point one can analyze the graphs throughput behavior. Its maximum throughput will be the overall longest time of completion of one iteration of the graph.

2.4.4 Latency Analysis

Although throughput is a very useful performance indicator for concurrent real-time applications, latency is also a relevant parameter of performance. Some applications might have strict latency constraints if the travel time of a signal is of great importance.

Latency is the time interval between two events. We measure latency as the difference between the start times of two specific firings of two actors, i.e.:

$$L(i,k,j,p) = s(j,p) - s(i,k) \tag{2.7}$$

where i and j are actors, p and k firings. We say that i is the source of the latency, and j is the sink. However, because dataflow graphs can execute for quite long lapses of time, we are more interested in the overall latency between two actors in all of their firings. Therefore we also introduce the concept of maximum latency in all iterations of the graph's execution:

$$\hat{L}(i,j,n) = max_{k \geq 0}(s(j,k+n) - s(i,k)) \tag{2.8}$$

where n is a fixed iteration distance. Next we will address the latency analysis of some specific cases of application, those with periodic behavior.

Maximum Latency from a periodic source

The start times of a periodic source are given by:

$$s(i,k) = s(i,0) + T \cdot k \tag{2.9}$$

In a graph with a periodic source of period T, the graph's behavior and rate of execution is imposed by the source, and we say that the graph is lower bounded by the period of the source. Notice that if this is not true, and the graph has a longer period than the source there would be an infinite token accumulation on some edge and periodic behavior is lost. For this reason we will perform latency analysis under the assumption that the MCM of the graph is lower or equal to the period T of the source. Assuming that the graph can at at most run at its ROSPS we can derive maximum latency has:

$$\hat{L}(i,j,n) = max_{k \geq 0}(s_{STS}(j,k+n) - s(i,k)) \leq \check{s}_{ROSPS}(j,0) - s(i,0) + \mu(G) \cdot n \tag{2.10}$$

where $\check{s}_{ROSPS}(j,0)$ represents the soonest start time of j in an admissible ROSPS. We can determine the maximum latency for a periodic source just by determining an ROSPS with the earliest start time j and a WCSTS for the earliest start time of i. A more extended approach to this analysis, including a detailed explanation of the proof of expression 2.8 can be found in [28].

Maximum latency from a sporadic source

Some applications, such as reactive systems, have sources that are not strictly periodic, but produce tokens sporadically with a minimum interval time (mit) between subsequent firings, which we will denote by η. In such applications, typically, a maximum latency must be guaranteed. In this situation an application to ensure its performance must keep up with the source, such is the case if the MCM, $\mu(G)$, of the graph G is equal or lower than the source's η. Therefore, we can bound the graphs behavior by its self-timed behavior with a static periodic schedule of period η, which as we have stated is possible if $\mu(G) \leq \eta$. We will assume that $\eta = \mu(G)$ and provide the formalization of maximum latency in a graph with a sporadic source:

$$\hat{L}(i,j,n) \leq \check{s}_{\eta} - s(i,0) + \eta \cdot n \tag{2.11}$$

The latency $\hat{L}(i,j,n)$ with a sporadic source has the same upper bound as the latency for the same source i, sink j, and iteration distance n in the same graph with a periodic source with period η. We opt to omit the proof as it is quite extensive and it can be found in [28].

2.5 Mode-Controlled Dataflow

Mode-Controlled Dataflow is a flavor of dataflow that captures both the expressive capabilities of dynamic dataflow and the analytical properties of static dataflow models. With

MCDF it is possible to build more realistic models of the application and express their dynamic behavior, as natural of streaming applications, while providing the necessary tools to do a temporal analysis on the built model.

Further on in this dissertation we will provide the reader with the context to which Mode-Controlled Dataflow (MCDF), [28], and Scenario-Aware Dataflow (SADF), [36], have been developed, however in this section we will focus on stating the properties and constructs of such models.

2.5.1 Overview

In a broad sense a MCDF graph is a compilation of several SDF graphs that model a certain behavior of an application. For example, in a WLAN application, there are several different operations depending on the input received: either synchronization, processing and CRC checking. These operations can be modeled by individual graphs, and then put together in a complete MCDF graph. Actors common to all modes are modeled as regular SRDF actors.

Intuitively, a MCDF graph is a dataflow graph where each firing of a designated actor, called the Mode Controller (MC), allows actors belonging to a specific subgraph are fired. In other words, a specific subgraph is chosen for execution, depending on an output value produced by the Mode Controller. Therefore there are data-dependent actor firings. After all the actors in the chosen subgraph have fired the graph returns to its initial token distribution.

Mode Controller

The **Mode Controller** (MC) is a special actor in MCDF graphs. The MC is responsible for the flow of the graph, its output token drives the control input tokens of all data-dependent actors in the MCDF graph. Each firing of the MC produces a single token with the value of the mode of operation to execute in that iteration. A mode of operation corresponds to an associated subgraph of the MCDF. For any given MCDF graph, there if a fixed number of modes, M. Therefore, the value of the output token of the MC has an integer value within the closed interval from 1 to M. Tokens produced by the MC are referred to as **control tokens**.

Data-Dependent Actors

Besides SRDF actors, a MCDF graph allows the usage of three types of data-dependent actors. These actors are the Mode Switch, Mode Select and Mode Tunnel actors. These data-dependent actors have in common the fact that they all have a control input port. A control token is read from this port for each firing of the data-dependent actor, and depending on its value it determines which of the port of the data-dependent actor are producing/consuming data during this firing.

We will provide the reader with the definition of each data-dependent actor exactly as they are defined in [28]:

Definition 4. *(Mode Switch) A **Mode Switch** actor has, besides the control input port, one data input port and M output ports. Each output port is associated with a mode. When a token is present on the control input port and on the data input port, the Mode Switch actor is fired. It consumes both input tokens and produces a token in the output port associated with*

the Mode indicated by the control token. The output token has the same size and value as the token consumed on the data input port.

Definition 5. *(Mode Select) A **Mode Select** actor has, besides the control input port, M data input ports and one output port. Each input port is associated with a mode. When a token is present on the control input port, its value is read and used to decide from which input port to consume a token. The actor is fired when a token is present in the control input port and in the data input port associated with the mode indicated by the control token. When fired, it consumes both of these tokens. At the end of the firing, it produces on the output port a token with the same size and value as read from the modal input port. if the input port was not connected, the output token will have some pre-defined default value, which an be mode dependent.*

Definition 6. *(Mode Tunnel) A **Mode Tunnel** actor has, besides the control input port, one data input port and one data output port. The data input port is associated with an arbitrary mode m of the graph, and the data output port is associated with a mode n of the graph, different from m. When the token reads at the control port the value of m, the Mode Tunnel, fires and consumes a token from the control input port and from the data input port. It stores the token read from the data input port in its internal state. When the control input port has value n, the Mode Tunnel fires, consuming that value and copying the token stores in its internal state to the data output port. The initial value of the internal state is graph-specific. In this way, the Mode Tunnel always delivers to its consumer the value produced by the last previous execution of the source orchestrated by the Mode Controller.*

2.5.2 MCDF Composition and Constructs

The rules we present for the construction of a well-constructed MCDF graph guarantee that it can always return to its initial state of token placement, independently of the sequence of mode control tokens. It will also ensure that per each fire of the MC only actor belong to the selected mode of operation, of actors with no assigned mode, will fire and that no other actors fire until another activation of the Mode Controller occurs.

An MCDF graph with M modes is composed of:

- A Mode Controller actor (MC);
- an arbitrary number of Data-Dependent actors;
- an arbitrary number of static, single-rate actors.

Before introducing the construction rules, we provide the reader with some important terminology and definition of MCDF graphs [28]:

Definition 7. *(Actor Modality) Actors in a MCDF graph are annotated with a valuation* mode*:* $V \rightarrow 1, ..., M$*. Since only some actors execute for specific modes, mode is partial function. If* mode*(i) is defined, then the actor is said to be **modal** and we can say that actor i **belongs** to* mode*(i). If* mode*(i) is undefined, denoted by* mode$(i) = \perp$*, the actor is said to be **amodal**.*

Definition 8. *(Actor Type) There are some actors in MCDF that have special attributes, these are the Mode Controller, the Mode Switch and the Mode Select. We use valuation atype:* $V \rightarrow mc, switch, select, normal$*, to indicate any special attributes of an actor.*

Definition 9. *(Modal Graph) The mode m subgraph of G,* modal graph *m, is a sub-graph* $G_m = (V_m, E_m, t, d)$ *of MCDF graph* $G = (V, E, t, d, M, mode, atype)$*, such that its vertex set* V_m *is composed of all amodal actors, and all actors that belong to mode m, and its edge set is composed of all edges which are in E and whose sources and sinks both belong to* V_m*, and where t and d are restricted to* V_m*.*

Construction Rules

The construction rules that must be respected by a MCDF graph for it to be considered well-constructed are as follows [28]:

- **Rule 1:** There is only one mode controller.
- **Rule 2:** Modal actors can either be connected to other modal actors, as sinks to the output ports associated with their mode on Mode Switches, or as sources to the input ports associated with their mode on Mode Selects. On the other hand, the ports of amodal actors other than the output ports of Mode Switches and the input ports of Mode Selects, can only be connected to fixed rate ports of other amodal actors.
- **Rule 3:** The mode control output port of the Mode Controller is connected to all control input ports in the graph through delay-less edges.
- **Rule 4:** There are no delay-less cycles in the graph.

2.5.3 Example

We will use WLAN as an application example to illustrate the behavior of a MCDF graph. Not only is it an easy to understand example of a dynamic application but also it will be a recurrent case study in this course of this dissertation.

Figure 2.3 depicts the MCDF modeled WLAN 802.11a receiver. Arcs that communicate control tokens are represented by dashed lines. The graph has 4 modes: Synchronization (Mode 1), Header Processing (Mode 2), Payload Processing (Mode 3) and CRC (Mode 4). The control flow of the modes of operations depends on the value outputted by the MC. Some modes may need to be repeated until the operation is completed, as is the case of Mode 1 where the mode is repeated until synchronization is successful. This information is sent to the MC by the Select actor.

After synchronization is complete the flow changes to Mode 2 to perform the processing of the header. As depicted in the figure, a Tunnel actor connects actor HDEC to PDEM, this represents the data dependency between modes 2 and 3, in this specific case HDEC will send to PDEM the actual parameters for the demodulation of the payload. Once the full payload is received the MC actor will change to Mode 4 to perform CRC checking of the packet. Notice that the only outputs of the MC that are dependent on the current mode of operation's results are the ones related to modes 1 and 2, therefore there is a connecting edge from the Select actor to the MC actor in these modes. The same does not occur for modes 3

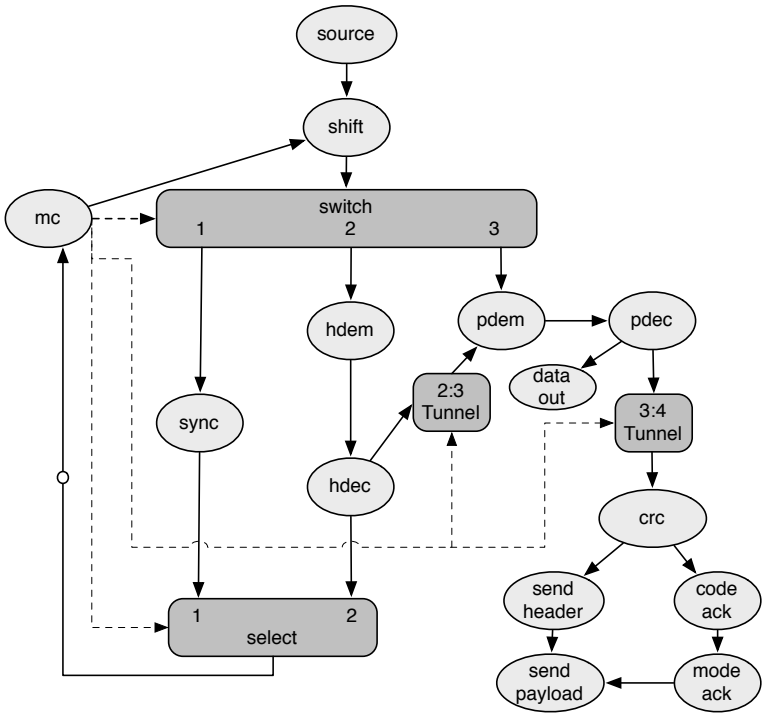

Figure 2.3: MCDF model of a WLAN Receiver

and 4. After completing mode 4 the next iteration of MC reverts to its initial state and the graph is ready to process a new incoming packet.

As a model, it is easy to understand that each mode of operation could be represented by a single SDF graph, and that a MCDF graph is a junction of several SDF graph with the graphs flow modeled by data dependent actors and dependencies between modal graphs represented by Tunnel actors.

2.5.4 Temporal Analysis

Temporal analysis of MCDF graphs is more complex than the previously introduced analysis for SRDF graphs. For this reason we opt to summarize the techniques proposed by [28] for worst-case temporal analysis, the current state-of-art we wish to improve in this dissertation. For a detailed explanation of these techniques and the concepts derived to reach the proofs associated, please refer to [28].

A first and simple approach is to bound on overall throughput and start times of all actors firings as given by MCM analysis of a rate-equivalent SRDF graph. This approach is a simple adaptation of the available technique for SRDF graphs, however it is pessimistic due to the fact of always taking into account the worst-case throughput across all modes as the worst-case throughput of the graph when executing in any mode. If we can reduce the set of mode sequences of interest for the application, as it is possible in some cases, then we may do an exhaustive simulation of that set of mode sequences, using a SPS schedule, to obtain more

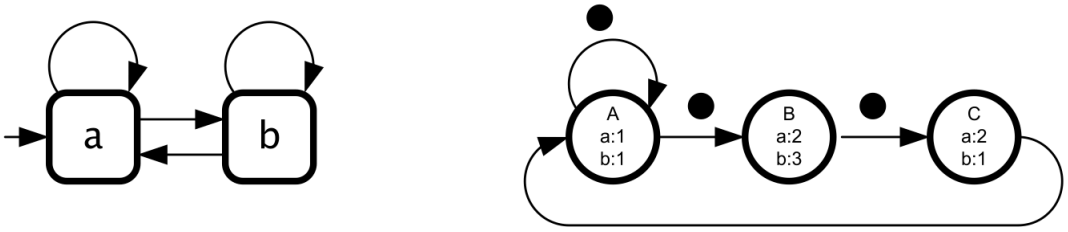

Figure 2.4: Example of an SADF graph

accurate results. Although it is still pessimistic since it still resorts to a SPS scheduling aproximation, not to mention the fact that this technique is not able to cope with infinite sequences.

2.6 Scenario-Aware Dataflow

Let us first present, in a simple and informal way, Scenario-Aware Dataflow.

SADF captures the dynamic behavior of an application in a model, which is still highly analyzable. The dynamic behavior of an application is captured in a set of scenarios, each modeled as an SDF graph, in which a task may have varying executions times and different port rates [36]. Each scenario models a specific mode of operation that may have unique tasks or shared tasks with other scenarios. Dividing the application in different scenarios allows for worst-case estimation to be more accurate. Scenario changes are controlled by scenario detectors and control channels, which control the flow of the application during its execution.

Figure 2.4 depicts an application modeled as a SADF graph. The available scenarios are a and b and in each scenario the actor might have different execution times. The application can either be represented by joining all possible scenarios and using different types of channels to distinguish different data-dependencies, data and control channels, or as a set of separate scenarios and a finite state machine to formalize all the possible transitions between application scenarios, such as the one in Figure 2.4.

2.6.1 Composition and Construct Rules

In SADF tasks are modeled as **kernel** and **detector** actors to be connected through **ports**. We distinguish three types of ports: data input ports, data output ports and control input ports. The finite set of input, output and control ports of a task p are denoted by I_p, O_p and C_p respectively. Edges that connect actors are referred as **channels**. If the channel connection is to either a data input port or output port the denomination is **data channel**; if it is to a control port then we refer to the channel as a **control channel**. A channel that connects ports of the same task is denominated a **self-loop channel**.

Kernel actors represent the data flow of the SADF graph, and each kernel actor might have different execution times in different scenarios. While, detector actors capture the control flow of the graph and define to which scenario a kernel actor belongs to in a certain iteration. To

define SADF, we first introduce the functions ϕ and ψ that capture the status of data and control channels.

Definition 10. *(Channel Status) Let B denote the set of all control channels of an SADF graph, where $B_c \subseteq B$ is the set of control channels. A data* channel status *is a function $\phi : B \backslash B_c \to \mathbb{N}$ that returns the number of tokens stored in the buffer of each data channel.*

Definition 11. *(Control Status) Let B_c denote the set of all control channels of an SADF graph. A* control channel status *is a function $\psi : B_c \to \cup_{c \in B_c} \sum^* c$ that returns the sequence of tokens stored in each control channel.*

Now we can define a SADF as:

Definition 12. *(SADF Graph) An SADF graph is described by a tuple $(K, D, B, \phi^*, \psi^*)$*

where, K and D are pairwise disjoint finite sets of kernels and detectors respectively, B is the set of channels and ϕ^* and ψ^* the respective channels status and control status. A kernel actor can be defined as:

Definition 13. *(Kernel Actor) A kernel actor $k \in K$ is a tuple $(I_k, O_k, C_k, S_k, \{(R_k^s, E_k^s) \mid s \in S_k\})$*

where I_k, O_k and C_k represent the actor's ports, S_k the set of scenarios to which k belongs,(R_k^s) the different port rates of the actor depending on the executing scenario and (S_k^s) the execution times of k in each scenario $s \in S_k$. A detector actor can be defined as:

Definition 14. *(Detector Actor) A detector actor $d \in D$ is a tuple (I_d, O_d, C_d, S_d)*

where I_d, O_d and C_d represent the actor's ports and S_d the set of scenarios of the SADF graph.

Some observations can be made from the previous definitions. A task can either be a kernel or a detector actor and a SADF graph consists of at least one such task. Each scenario determines the rates and execution time distribution for a kernel actor, and scenario changes are imposed by the control token provided by the connected detector actor. It is possible to have more than one detector actor in a SADF graph, and different detector actors may connect to different kernel actors, however, every detector actor must have a predefined action for all scenarios of the graph.

2.6.2 Example

For simplicity sake, we use the same case study as before, the WLAN baseband receiver. Figure 2.5 show a adaptation of the MCDF model in Figure 2.3 to SADF. It is important that the reader notices that a SADF graph can be easily reached by modifying a MCDF graph model, as this will be one of the arguments for the analysis technique implemented in Chapter 4.

We can see that, in Figure 2.5, only one actor is a detector, the MC actor, that will control the flow of the scenario choices. On the right side of the picture there is a finite state machine that describes the order in which scenarios can transit amongst each others. We will not repeat the behavior of the model as it is exactly the same as described previously. What

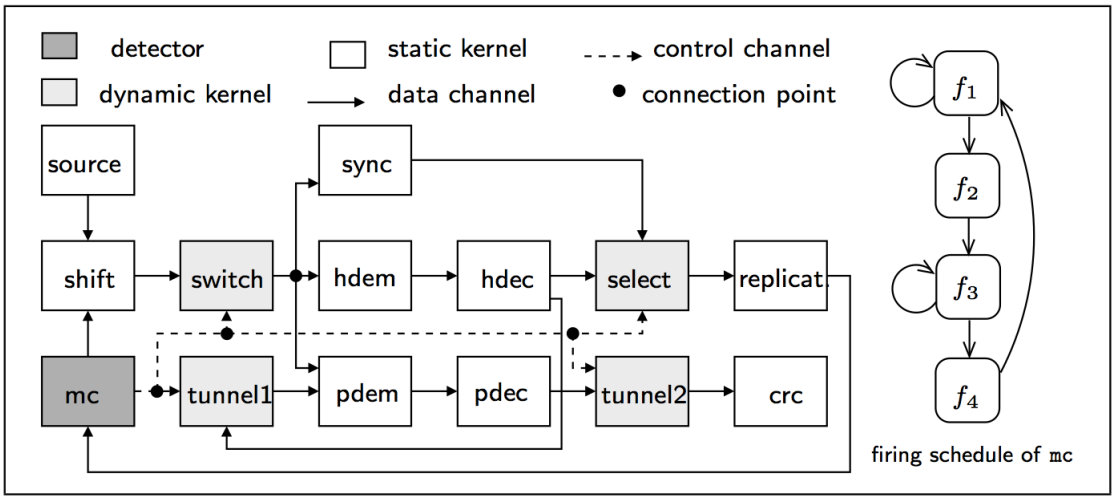

Figure 2.5: SADF model of a WLAN Receiver

is important to observe from this figure is how a SADF graph can be constructed and, that similarly to MCDF graphs, the detector actor defines the flow of the graph by sending control tokens to some kernel actors (Switch, Select and Tunnel).

For further information and examples on the construction and behavior of SADF graphs, please refer to [14, 36].

Chapter 3

Software Framework

This dissertation focuses on the study of temporal analysis techniques for dataflow models of computations. We make two major contributions: one in temporal analysis for Mode Controlled Dataflow (MCDF) graphs and one in fixed priority analysis for Single Rate Dataflow graphs (SRDF). For this purpose, the whole project relies heavily on software tools for simulation and analysis of dataflow graphs. We base our software framework on the Heracles simulator and analysis tool, envisioned and created by Orlando Moreira. Heracles is a complex and versatile tool for the design, schedule, simulation, programming and analysis of dataflow models. Despite the many functionalities we focus, during this dissertation, mostly on the simulation and temporal analysis aspects of the tool.

In this chapter we introduce the reader to our software framework. We start by giving an overview on the Heracles tool and providing a more detailed insight on the most important modules. Furthermore, we present the reader with the current state of the modules we set on improving: MCDF and Fixed Priority temporal analysis.

3.1 Heracles

3.1.1 Overview

Heracles provides the user with the necessary tools to throughly analyze and schedule one, or more, applications on a MPSoC platform. Figure 3.1 depicts the flow of the Heracles tool. For instances, if we want to test a model of a radio transceiver on a specific system, using Heracles, we input a dataflow model of the transceiver, a description of the system in terms of resources and their characteristics and the timing requirements of the application. From this point, Heracles combines its scheduler and temporal analysis modules to derive, an implementation-aware graph, from the functional graph of the transceiver, that models worst-case assumptions about the timing of actors firings and communications in the given platform. This allows for a mapping flow where every mapping decision can be translated onto a transformation of the application graph, and evaluated with respect to its temporal behavior. The scheduler will then try to find the solution that schedules the application and makes best use of the systems resources. Moreover, it is also possible to determine buffer sizes for the system.

Although, this is the main flow of Heracles, it is still possible to use each feature individually. For example, if the user just wants a temporal analysis of a dataflow graph, or simply to simulate the graphs execution. Therefore, we can distinguish three different work flows for

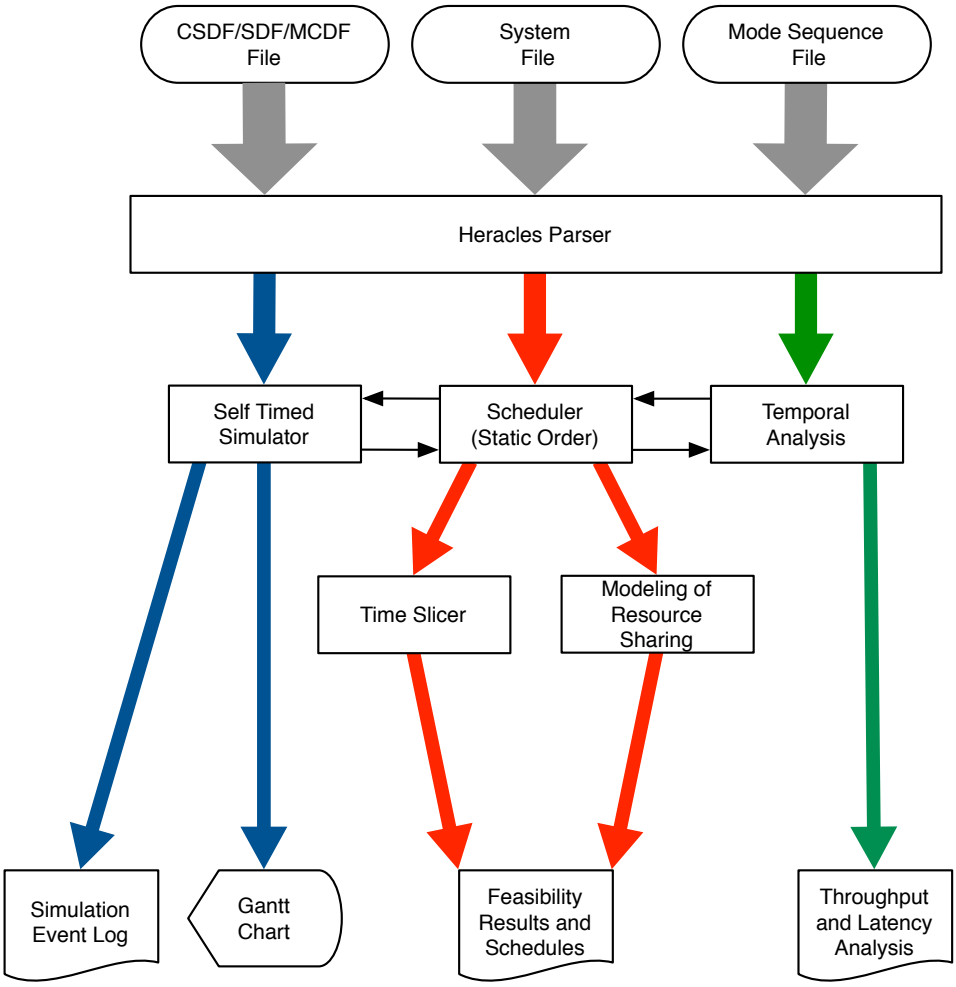

Figure 3.1: Description of the Heracles Tool General Flow

Heracles, as depicted in Figure 3.1. The three main flows are: simulation (blue), temporal analysis (green) and scheduling (red).

In the following sections we will describe each flow with more detail. Furthermore we will state the current implementation of Heracles modules regarding fixed priority and MCDF temporal analysis. But first, we will give a small introduction to the base language in which Heracles is written.

OCaml

Heracles is written in Objective Caml (OCaml). OCaml is a general-purpose language designed with robustness and reliability in mind, developed at INRIA (Institut National de Recherche en Informatique et en Automatique). Furthermore, OCaml has many interesting features. To enumerate a few:

- **Hybrid Paradigm**: OCaml is not just a functional language, but also an imperative and object oriented language. Therefore, it is possible to mix and match all those

paradigms at will.

- **Type Inference**: Type inference refers to the process of determining the appropriate types for expressions based on how they are used. For example, $fun\ a = a + 1$ is a function that receives a single input and adds the value 1. Therefore, the type of a is an *integer*.
- **Pattern Matching**: OCaml allows the use of *pattern matching* in function definitions. As a result, the structure of the function models the structure of the data it is processing, making it easy to see base cases and harder to miss a case.
- **Extensive Libraries** OCaml comes with an extensive standard library: **lists, tuples, hash-tables, POSIX, and others**. Moreover, there are many third-party libraries from community contributions.
- **Efficiency** OCaml is very efficient in terms of execution performance. Many benchmarks conclude that OCaml can be as fast as C.
- **Portability** The byte-code interpreter compiles and runs on any POSIX-compliant system with an ANSI C compiler. The generated byte-code files are also completely portable between most operating systems and processor architectures.
- **Debugging** and **profiling** tools are provided along with the compiler.

For more information regarding this programming language, please refer to [13, 17, 21].

3.1.2 Heracles Temporal Analysis

Temporal analysis is required in order to verify whether a given timed dataflow graph can meet a required throughput or latency requirement of the modeled application. In context with Heracles purposes, not only is temporal analysis important in terms of timing guarantees, but also in order to be able to make scheduling decisions.

Currently, Heracles has implemented analysis techniques for the analysis of self-timed behavior of data flow variants with static rates (SRDF, MRDF and MCDF), even when considering the transient phase and/or varying execution times per actor firing.

For SRDF and MRDF specifically, temporal analysis modules allow for the latency analysis of graphs with periodic, sporadic and bursty sources; and throughput analysis of static dataflow graphs by the use of Maximum Cycle Mean computation algorithms, or simulation.

MCDF Temporal Analysis

Current implemented temporal analysis for MCDF graphs are either pessimistic or non-complete. One method consists on analyzing the MCDF graph as an SRDF graph. Despite complete, this analysis is quite pessimistic as it does not take advantage of the dynamic behavior of the graph.

The second analysis, is the Static Periodic Schedule simulation and temporal analysis of a specific mode sequence. Although less pessimistic than the previous method, it requires that each individual mode sequence of the MCDF graph is tested, and only works for finite mode

sequences.

In this dissertation, we will present a temporal analysis for MCDF that is complete and which results are tighter than current implemented techniques.

3.1.3 Heracles Scheduler

Heracles scheduling strategy involves a combination of static-order scheduling per application per processor, and a Time Division Multiplexing (TDM) or Non-preemptive Non-blocking Round Robin (NPNBRR) scheduling to arbitrate between different jobs in each processor. Therefore, the scheduling flow is divided in two main steps: intra-job scheduling and inter-job scheduling per processor.

The scheduler requires three inputs: the task graph of the transceiver, a description of the target platform and a set of timing requirements.

The first step is to call the inter-job scheduler for each application, to build a static order schedule for each processor of the system. The second step is then to arbitrate, according to each processor scheduling type, the resource sharing between applications. A final optimization step can be done for TDM scheduling, where the slice times are readjusted to find a better values in terms of resources utilization.

After concluding its operations, the scheduler returns a best schedule found per processor of the system.

3.1.4 Heracles Simulator

Another of Heracles features is the dataflow graph simulator. Currently, it simulates any dataflow graph (SRDF, MRDF, CSDF and MCDF) in a self-timed schedule. However, it does not consider hardware mappings.

It is implemented as an event simulator. In other words, the execution of a dataflow graph is represented by a series of different events: *start events* and *finish events*. An event is defined by an id, an issue number, a start time and an actor. A *start event* is initially issued for every actor in the graph and added to an *ordered set* of events. Every time an event is added, the set is reordered. A *finish event* is issued when an actor finishes its firing and produces its output tokens.

When the simulator is initiated it picks the event at the top of the set and checks whether it is a *start* or *finish* event. If it is a *start event* then the simulator checks if the actor meets its firing conditions and if it does, then the necessary tokens are consumed and a *finish event* for that actor is issued. Otherwise, the *start event* is discarded. If the picked event is a *finish event* then tokens are produced on the output edges and start events for that actor and its dependencies are added to the event set. Furthermore, during its execution, the simulator keeps constant track of all buffer sizes (tokens available in all edges).

The fundamental base of the simulator is the *compare* function. This function is responsible for the ordering of the event set every time a new event is added. If two events have the same start time then *Finish events* are always given precedence. Why? Because *finish*

events release resources while *start events* reserve them, thus avoiding possible deadlock of the simulator. If the events have the same start time and are of the same type, then the decision factor is reduced to the issue value, in a first come first serve fashion. Notice that a poorly conceived *compare* function can quite easily create deadlock situations.

The simulator will run until the event queue is empty, either due to deadlock or expiration of the simulation time, or until the same exact state is reached, in other words, the graph assumed a periodic behavior and there is no added relevance to further simulation data. As a result, the simulator outputs an event list for actors and edges,and a gantt chart of the simulation. Figure 3.2 depicts the flow of the Heracles simulator.

3.1.5 Heracles Fixed Priority Analysis

Since the fixed priority analysis contribution of this dissertation is an extension of the work done in [1] it is important to differentiate from what was already implemented and what is new implementation work.

During the work done in [1], a fixed priority module was added to Heracles. This module was created to perform interference and response time analysis for applications with a fixed priority assignment. Simply put, it features two main external call functions, that we will refer to as: *merge* and *fill*. The *merge* function receives two different simulation timelines of fixed priority applications and merges them into a single timeline that considers preemption due to different priority of applications. This function is needed since Heracles simulator does not contemplate preemption or resource mappings.

The *fill* function receives a simulated timeline and slot fills a single task execution into the timeline. The purpose of the function is to determine the response time of a low priority actor when its execution is affected by the interference execution of higher priority actors.

In the course of this dissertation, we corrected and improve Heracles fixed priority module, and extended all algorithms to perform the analysis for n applications, instead of the previous limit to two applications. We will discuss further both these functions, and how we use them in our fixed priority analysis in Chapter 5.

On a later stage, the merged timeline would be used by the *slot fill* function to retrieve the response times of the the lowest priority application, considering the interference due to higher priority applications task.

3.2 Major Modifications to Heracles

In order to achieve the objectives proposed for this dissertation some modifications had to be made to the Heracles tool. Detailed explanations of specific modifications will be addressed in the ending of each corresponding chapter.

- **MCDF simulator:**

  The Heracles simulator, at the time of the beginning of this dissertation, was not able to simulate MCDF graphs. In order to adapt the simulator to correctly run MCDF graphs we needed to change the firing and the consumption/production rules. Firing rules for

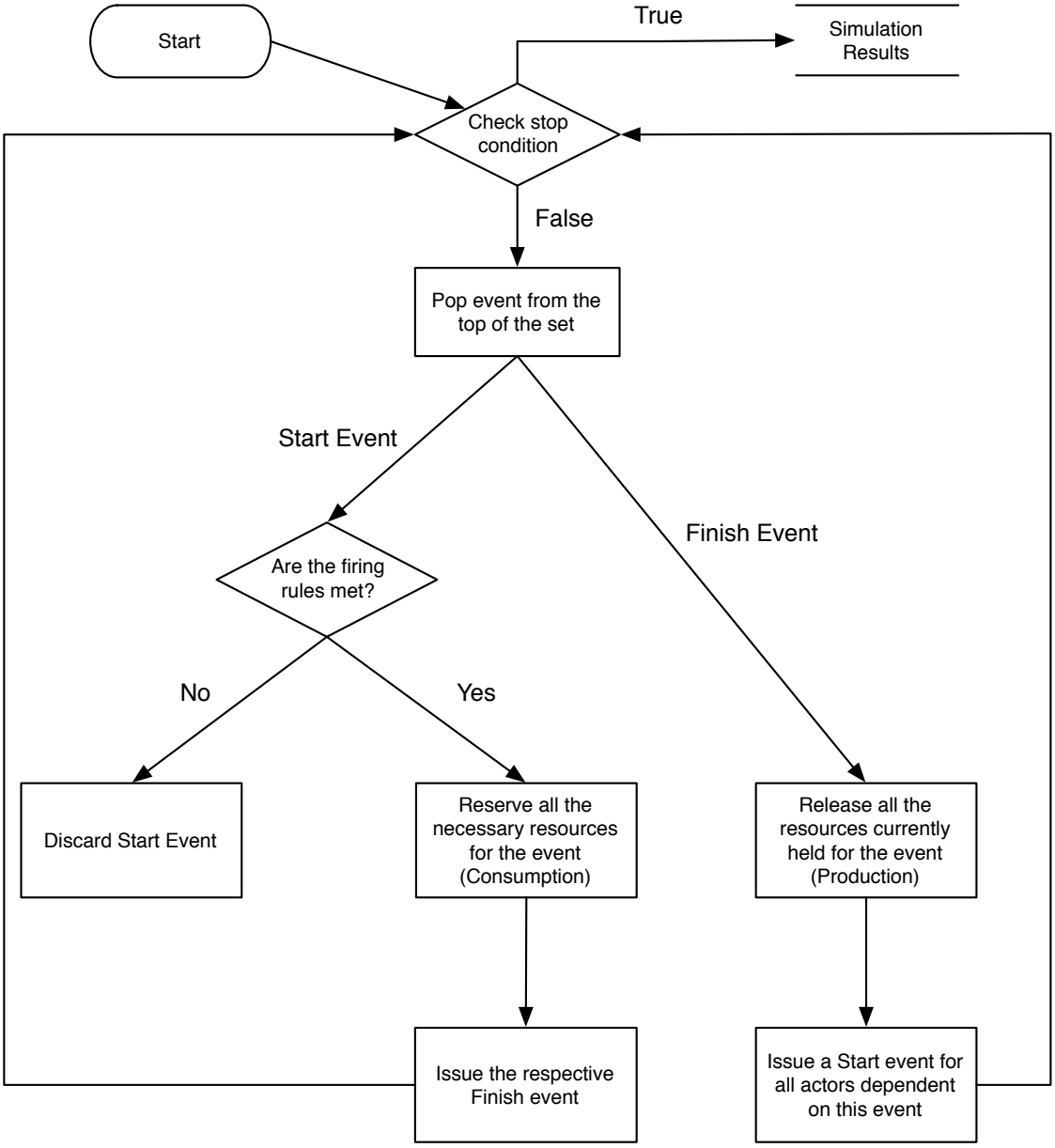

Figure 3.2: Description of the Heracles Simulator Flow

MCDF graphs, besides the natural SDF rules, need to verify that the *start event* either belongs to an actor with the same mode as the currently selected mode of operation, or to an amodal actor. The same principle had to be applied to the consumption and production rules. Consumption and production should only occur in edges that link to actors of the same mode as the currently selected mode of operation, or to an amodal actor.

Implementation wise, we added a function to retrieve the current mode of operation from the number of firings of an amodal actor, which would coincide with the correct mode from the *mode sequence*. The retrieved *current mode* parameter is then used for verification in various steps of the event simulator to assure that all events issued and completed are in conformity with the correct dynamic flow of the application. Notice that in order to make the necessary changes the core flow and algorithm of the simulator was not changed.

- **MCDF throughput analysis:**

  As was part of the motivation of this work, we implemented an optimal and overall method for throughput analysis of a dynamic MCDF graph. This implementation required the adding of new modules to support modules for conversion, algebra and state-space analysis. All these implementation steps will be addressed with detail at the end of Chapter 4.

- **SRDF Fixed priority analysis module:**

  We revisited the previously implemented model to correct some bugs and errors in the analysis, we extended the existing technique for the analysis of n applications, instead of just two, and implemented a new module to recreate the challenging fixed priority analysis algorithm proposed in [18]. All these implementation steps will be addressed with more detail at the end of Chapter 5.

Chapter 4

FSM-MCDF

Current methods for analyzing the throughput of a MCDF graphs rely on pessimistic approximations. The simplest way of obtaining a worst-case throughput analysis is to use the methods available for SRDF to evaluate every mode of operation of the application individually. Results will be conservative but pessimistic because a SPS scheduling is used to bound the start-times of actors.

Marc Geilen et al [15], purposed a new method for throughput analysis for SADF models that relies on searching the state space of all possible transitions using an automaton, a finite state machine, and simulating the transition times and dependencies. Therefore, it is possible to take into account transition times, overlapping modes and infinite sequences of modes.

SADF can be easily modeled as an MCDF graph, and so we decided to adapt the proposed technique to MCDF graphs, in order to improve the current analysis methods.

This chapter introduces the basic concepts behind the proposed method, shows the implementation steps and compares the results obtained with the previous methods.

4.1 Max-Plus Algebra

Max-plus algebra is a mathematical framework supported by the binary operations *max* and *plus* in a set $\mathbb{R} \cup \{-\infty\}$, which we will denote by $\mathbb{R}_{max}$. This is reference material and most of it can be found in [19].

The Max-plus approach emerged from the need to have a mathematical framework that would be adequate for Discrete Event Systems (DES). Discrete Event Systems represent any system in which its dynamics are made up of *events*, like Dataflow models. Furthermore, the use of Max-plus algebra is limited to certain classes of DES, those which involve *synchronization* of events and that events are *timed events*. During this section we will provide the reader with the basic concepts of Max-Plus algebra and how it can be used as a framework for Dataflow models.

We now present the basic concepts and definitions of Max-Plus algebra, as well as all the advanced concepts used throughout this thesis.

Let $\epsilon = -\infty$, $e = 0$ and $a, b \in \mathbb{R}_{max}$, we can define operations $\oplus$, *max*, and $\otimes$, *plus*, by

$$a \oplus b = max(a, b) \qquad \text{and} \qquad a \otimes b = a + b.$$

Also, for any $a \in \mathbb{R}_{max}$

$$max(a, \epsilon) = max(\epsilon, a) = a \text{ and } a + (\epsilon) = \epsilon + a = \epsilon,$$

Such that,

$$a \oplus \epsilon = \epsilon \oplus a = a \qquad \text{and} \qquad a \otimes \epsilon = \epsilon \otimes a = \epsilon,$$

Therefore, ϵ is the zero element and the absorbent element, and e is the unit element. Other algebraic properties of Max-Plus algebra are listed below:

To exemplify these concepts we illustrate with some numerical examples:

$$\begin{aligned}
&10 \oplus 3 = max(10, 3) = 10,\\
&10 \oplus \epsilon = max(10, -\epsilon) = 10,\\
&10 \otimes \epsilon = 10 + \epsilon = \epsilon,\\
&10 \otimes 3 = 10 + 3 = 13,
\end{aligned}$$

4.1.1 Vectors and Matrices

In this subsection we introduce the concepts of matrices and vectors in Max-Plus algebra. Matrices and matrix operations will be the basis of the algorithms presented in this chapter.

In Max-Plus algebra a matrix $A \in \mathbb{R}^{nxm}$, with n columns and m rows, can be written as:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \tag{4.1}$$

Where a_{ij} represents the element of the matrix in row i and column j. Occasionally, the element a_{ij} will also be denoted as $[A]_{ij}$, with $i \in n$ and $j \in m$. The sum of matrices A and B is denoted by $A \oplus B$, and is defined by:

$$[A \oplus B]_{ij} = a_{ij} \oplus b_{ij} = max(a_{ij}, b_{ij}), \tag{4.2}$$

for $i = 1, ..., m$ and $j = 1, ..., n$. The product of matrices A $\in \mathbb{R}^{n \times l}_{max}$ and B $\in \mathbb{R}^{l \times m}_{max}$ is denoted by $A \otimes B$, and is defined by:

$$[A \otimes B]_{ik} = \bigoplus_{j=1}^{l} a_{ij} \otimes b_{jk} = max_{j \in l}(a_{ij} + b_{jk}) \tag{4.3}$$

for i $\in n$ and k $\in m$. To illustrate matrix operations we present the reader with some examples:

Let A $= \begin{pmatrix} 2 & 4 \\ \epsilon & 10 \end{pmatrix}$ and B $= \begin{pmatrix} 10 & 0 \\ 4 & 14 \end{pmatrix}$ and that A,B $\in \mathbb{R}_{max}^{n \times m}$, then we can write the following expressions:

$$[A \oplus B] = \begin{pmatrix} 10 & 4 \\ 4 & 14 \end{pmatrix} \qquad [A \otimes B] = \begin{pmatrix} 12 & 18 \\ 14 & 24 \end{pmatrix}$$

$$[B \oplus A] = \begin{pmatrix} 10 & 4 \\ 4 & 14 \end{pmatrix} \qquad [B \otimes A] = \begin{pmatrix} 12 & 14 \\ 6 & 24 \end{pmatrix}$$

Notice that the matrix product in general fails to be commutative, while the sum operation holds that $A \oplus B = B \oplus A$.

As in regular algebra, vectors are a subset of the set of matrices $\mathbb{R}_{max}^{n \times m}$. Vectors are the elements of $\mathbb{R}_{max}^{n \times 1}$, and the jth element of vector $x \in \mathbb{R}_{max}^{n}$ is denoted by x_j.

$$X = [x_0, x_1, ..., x_n]^T : x \in \mathbb{R}_{max} \tag{4.4}$$

Throughout this chapter we will use these notions as explained in this section. We refer as vectors to column matrices. For simplicity all vectors will be written in their transposed form.

4.2 Max-Plus and Dataflow

Dataflow models behavior can be reduced to two fundamental operations: *synchronization* and *delay*. A dataflow graph, such as an SDF graph, running on a self-time schedule behaves in such manner that delays, or tokens representing data, can be exchanged between actors, or tasks, by following the firing rules. On a self-timed schedule an actor will fire as soon as all of its input tokens are available, which can be seen as a *synchronization* operation. The lapsed time between the consumption of the input tokens and the production of the output tokens, the firing of the actor, can be seen as a *delay* operation. We use $s(i, k)$ to represent the start time of actor i in iteration k and $f(i, k)$ to represent the finish time of actor i in iteration k.

Let N be a set of actors and D_i the set of direct dependency actors of actor $i \in N$. The start time of actor i in iteration k depends on the finishing time of all the actors that exist in D_i and can be represented as a *synchronization* operation:

$$s(i, k) = max_{j \in D_i}\{f(j, k)\} \tag{4.5}$$

And the finishing time of actor i can be defined as:

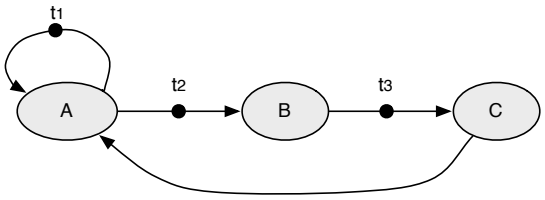

Figure 4.1: SRDF graph example

$$f(i,k) = s(i,k) + e(i,k) \tag{4.6}$$

Where e stands for the execution time of actor i in iteration k. It is easy to understand that both these expressions, *synchronization* and *delay*, are in fact Max-Plus expressions, *max* and *plus* as we have defined them. Therefore Max-Plus algebra can be used as a semantical framework for Dataflow models.

We can rewrite equations 4.5 and 4.6 in terms of token production times, instead of actor firings, in the following way:

$$t_i = max_{j \in T} t_j + e, \tag{4.7}$$

where t_i represents the time at which actor i produces its output tokens, T the set of tokens required by actor i to fire and e the execution time of actor i.

The behavior of a dataflow graph can also be characterized by the time at which tokens in channels are produced, thus allowing us to capture the data dependencies between iterations. We start by assuming a initial graph state of the application we wish to study, and build a matrix that represents the dependencies between initial tokens of the graph. Such matrix can be found by doing a symbolical execution of the model. The process of finding a matrix can be easily understood when accompanied by an example. Lets assume the SRDF graph of Figure 4.1, and that the tokens displayed are the initial tokens of the graph and that they are all available at time zero. The following explanation is inspired by the one in [15].

We start by introducing the concept of time-stamp vector γ. This vector is used to keep the productions times of all the initial tokens of the graph in each iteration of the graph. An iteration is finished when all the initial tokens travelled the graph and have returned to their initial positions. After one iteration there are the same number of tokens in the same channels, but with different times at which they were produced.

Time-stamp vectors have as many elements as there are initial tokens in the graph. Each element holds the correspondent token's production time in that specific iteration, for our example, the generic time-stamp vector is $[t_1, t_2, t_3]^T$. Taking as an example Figure 4.1 graph, we can state that the initial vector, under our assumptions, is $[0, 0, 0]^T$. After one iteration it becomes $[3, 3, 2]^T$, after two $[5, 5, 5]^T$, as demonstrated in the Gantt Chart of Figure 4.2.

We will now show that it is possible to find a Max-Plus matrix G that can characterize the behavior of a dataflow graph, as in the following equation:

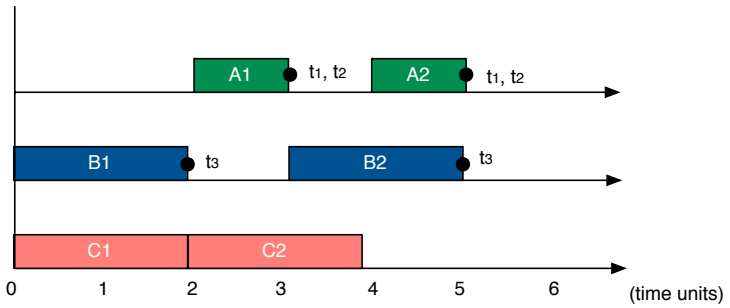

Figure 4.2: Gantt Chart of two iterations of Figure 4.1 graph

$$\gamma_{k+1} = G \cdot \gamma_k \tag{4.8}$$

where γ_k is the time-stamp of current iteration k and γ_{k+1} the time stamp of the next iteration of the graph. We associate each token with a representative number and a symbolic time-stamp vector t:

$$t_1 = [0, -\infty, -\infty]^T; \qquad t_2 = [-\infty, 0, -\infty]^T; \qquad t_3 = [-\infty, -\infty, 0]^T;$$

Each symbolic time-stamp vector represents an initial token and its dependency to other initial tokens. For instances, time-stamp t_1 represents initial token t_1, that in its initial state is already available and therefore the only dependency is with himself, thus the first element is zero. The same logic applies for all initial symbolic time-stamp vectors, the only dependency is an instant dependency with the actor itself.

Lets now run a symbolical execution of the model. We start by firing actor C, which will consume initial token t_3, and the tokens produced by actor C will carry the symbolic time stamp:

$$max([-\infty, -\infty, 0]^T) + 2 = [-\infty, -\infty, 2]^T, \tag{4.9}$$

which corresponds to the expression $t' = max(t_3) + 2$. Notice that after the firing of C generated a new symbolic time-stamp for the token produced in the edge C-A.

Now actor A can fire, consuming the newly produced token in edge C-A, and the token on its self-edge t_1. The produced tokens will carry the symbolic time-stamp:

$$max([0, -\infty, -\infty]^T, [-\infty, -\infty, 2]^T) + 1 = [0, -\infty, 3]^T \tag{4.10}$$

Now, because, actor A is responsible for the production of tokens t_1 and t_2, the new tokens will carry the time-stamp of $[0, -\infty, 3]^T$. At this point, tokens t_1 and t_2 have been replaced and therefore the new time-stamp represents the overall time dependencies from all initial tokens. We will associate these time-stamps with vectors g_1 and g_2, respectively, that will later be used to generate matrix G. Lastly, actor B will fire, consuming the initial token t_2

and producing a new token with the symbolic time-stamp:

$$max([-\infty, 0, -\infty]^T) + 2 = [-\infty, 2, -\infty]^T, \tag{4.11}$$

which will be associated with vector g_3, because it is the final time-stamp for token of interest t_3.
As we see that the graph has returned to its initial state, we can conclude that an iteration as passed and we now have the three vectors, g_1, g_2 and g_3, to build the iteration dependency matrix G. Matrix G will be the aggregation of the found token time-stamp vectors in a new vector $[g_1, g_2, g_3]^T$, obtaining:

$$\gamma_{k+1} = \begin{bmatrix} 0 & -\infty & 3 \\ 0 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix} \gamma_k \tag{4.12}$$

The resulting matrix G, represents all the initial token dependencies amongst them. An entry t at column k and row m in the matrix specifies that there is a minimum distance of t between time-stamps of token k of the previous iteration to token m of the new iteration, from dependencies in the graph. An entry $-\infty$ means that there is no dependency relation.

In summary, the behavior of any SDF graph can be characterized by a corresponding Max-Plus matrix G_F. This matrix can be computed by doing a symbolic execution of an iteration of the graph, as we did above. During our example explanation we stated that for the first three iterations the graph's time-stamp vector, γ, would be $[0, 0, 0]^T, [3, 3, 2]^T$ and $[5, 5, 5]^T$, corresponding to iterations 0, 1 and 2 respectively. Lets now verify that the matrix G_F matches these results. Notice that all the matrix operations are Max-Plus operations.

After one iteration,

$$\gamma_1 = G \cdot \gamma_0 = \begin{bmatrix} 0 & -\infty & 3 \\ 0 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} max(0+0, -\infty+0, 3+0) \\ max(0+0, -\infty+0, 3+0) \\ max(-\infty+0, -\infty+0, 2+0) \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} \tag{4.13}$$

After two iterations,

$$\gamma_2 = G \cdot \gamma_1 = \begin{bmatrix} 0 & -\infty & 3 \\ 0 & -\infty & 3 \\ -\infty & 2 & -\infty \end{bmatrix} \begin{bmatrix} 3 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} max(0+3, -\infty+3, 3+2) \\ max(0+3, -\infty+3, 3+2) \\ max(-\infty+3, 2+3, -\infty+2) \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix} \tag{4.14}$$

which is equivalent to what we wanted to show:

$$\gamma_{k+1} = G \cdot \gamma_k; \tag{4.15}$$

Algorithm 1 describes, step-by-step, the implemented methodology for the computation of the Max-Plus matrix representation of a specific SDFG, based on the proposition by Marc Geilen [15].

input : Set of Initial Tokens, Graph Schedule
output: Matrix
Compute Matrix (F);
/* T associates symbolic time-stamp i_k to initial tokens t_k */;
$T \leftarrow \{(t_k, i_k) | t_k \in InitialTokens\}$;
$\sigma \leftarrow$ Sequential Schedule of F;
for *j = 1 to Lenght(σ)* **do**
    Actor a $\leftarrow \sigma(j)$;
    Fire a consuming tokens $U \subset T$;
    Produce output tokens with time-stamp $g_p \leftarrow max\{g(t) | t \in U\} + E(a)$;
    add new tokens to T;
end
G $\leftarrow [g(t)]$ for all $t \in InitialTokens$;

Algorithm 1: Computation of Max-Plus matrix of a SDFG

4.3 Implementation

In this section we will adapt the throughput analysis technique of the FSM-SADF graphs, proposed by [15,32]. The novelty of the method is the ability to explore all possible transitions of an SADF graph by the means of a Finite State Machine (FSM).

Definition 15. *FSM-SADF $\mathcal{F}$ is a tuple $\mathcal{F}$ = (S,f). S is a set of scenarios and f is an FSM on S.*

To compute the throughput all possible scenario sequences have to be checked. Therefore, a state-space needs to be generated. The state-space is defined by:

Definition 16. *(FSM State-Space) is a tuple (C, c_0 , Δ). C is a set of configurations (q, γ) with a state q of $\mathcal{F}$ and a (max,+) vector γ. The initial configuration $c_0 = (q_0, 0)$,*

And labelled transition relation $\Delta \subseteq C \times \mathbb{R} \times C$ consisting of the following transitions:
$\{(q, \gamma), \|\gamma^{'}\|, (q^{'}, \gamma^{'norm})) \mid (q, \gamma) \in C, (q, q^{'}) \in \delta, \gamma^{'} = G_{q'}\gamma\}$

A state is then defined by a pair of a scenario of the SADF graph and a normalized vector indicating the relative distance in time of the time-stamps of the tokens. An edge represents a transition to another scenario, within the FSM possible sequences, and its weight represents the transition time between scenarios. Note that because of linearity, it is always possible to find an upper bound for any scenario sequence. In general, for any path leading to a state (q,γ), the exact tightest upper bound that can be given is the sum of all the weight along the edges of the path.

The state-space can be built in a depth-first-search (DFS) or breadth-first-search (BFS) manner, while checking for desired constraints if needed. With all the reachable state-space

found, the worst-case throughput of the SADF graph can be determined by a maximum cycle mean analysis of the state-space.

We will now explain, step-by-step, the implementation on MCDF graphs.

4.3.1 Converting MCDF to Max-Plus

In order to implement this method for MCDF graphs we first need to adapt the structure of MCDF. MCDF, as explained in Chapter 2, has special constructs that allow for data to flow differently according to the current mode of operation of the application. We want to study the transition between modes of operation and do an exhaustive temporal analysis for any possible infinite sequence of modes.

We start by dividing the MCDF graph into its modal graph representation, which is an SDF graph per mode of operation. To avoid complex modifications we keep the MCDF constructs in each modal graph, such as Switches, Selects and ModeControllers. We only alter the Tunnel constructs, replacing Tunnels by a self edge in both actors connected, but the token will be labelled the same in both self-edges. Doing this allows us to establish a FSM-based Mode-Controlled Data flow (FSM-MCDF) graph, that in similarity to FSM-SADF, can be defined as:

Definition 17. *(FSM-MCDF) FSM-MCDF is a tuple (M, f). M is a set of modal graphs and f is an FSM on M.*

The State-Space of an FSM-MCDF will be define as in Definition 16.

The next step is to assign to each modal graph its Max-Plus matrix representation to start the state-space generation. The process to compute the G matrix is the same described in the previous section and in Algorithm 1.

For simplicity purposes, we will look to an example application where all initial tokens are present in every modal graph and no tunnels exist in the original MCDF graph. This is a very simple example, but allows for an easy exemplification of the process of state-space generation. All other cases will be addressed later in this section.

This transformation, from MCDF to FSM-MCDF, can be done easily by following Algorithm 2 and creating a *transition graph* to serve as a FSM. Figure 4.4 shows an example of an FSM-MCDF graph and correspondent *transition graph*, of the MCDF graph in Figure 4.3

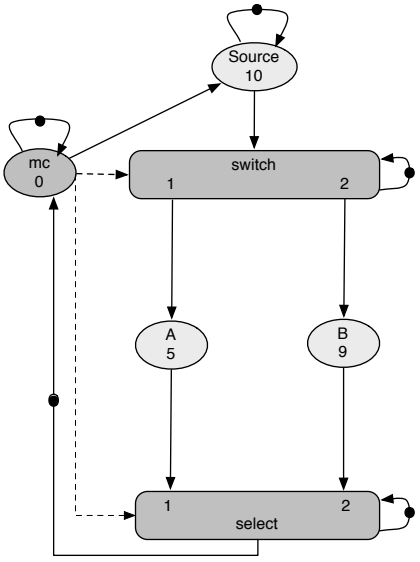

Figure 4.3: MCDF graph example

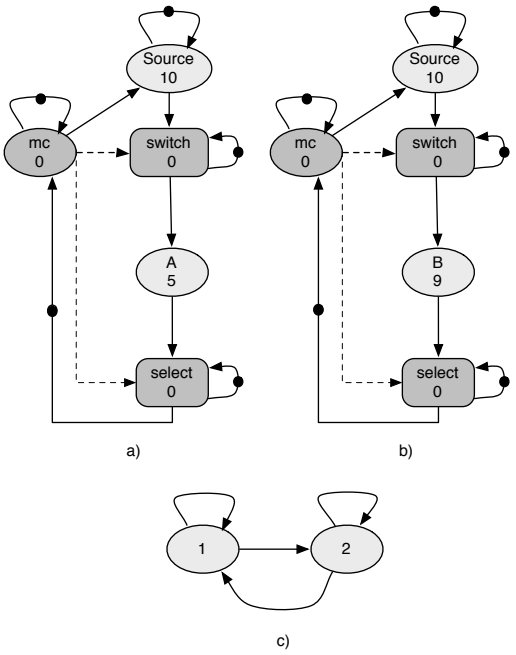

Figure 4.4: FSM-MCDF graph example. a) Modal Graph 1 b) Modal Graph 2 c) Finite state machine or transition graph

input : ModeSequence, MCDF graph
output: A hastable with all modal graphs
replace all Tunnels from input graph;
for *each mode in the* ModeSequence **do**
  $tMCDF \leftarrow$ input graph;
  $m \leftarrow$ current mode;
  for *each actor in the tMCDF graph* **do**
    if *actor mode != i and* empty **then**
      remove actor from tMCDF;
      remove edges of actor from tMCDF;
    end
  end
  add (m, tMCDF) to *modalgraphs*;
end

Algorithm 2: Converting MCDF into FSM-MCDF

.

Following the symbolic execution of the model method, express in terms of Algorithm 1, we will compute all matrices for the modal graphs. The results are the following:

$$G1 = \begin{bmatrix} 0 & 0 & 0 & -\infty & 10 \\ 5 & 5 & 5 & 0 & 15 \\ -\infty & 0 & 0 & -\infty & -\infty \\ 5 & 5 & 5 & 0 & 15 \\ -\infty & -\infty & -\infty & -\infty & 10 \end{bmatrix} \tag{4.16}$$

$$G2 = \begin{bmatrix} 0 & 0 & 0 & -\infty & 10 \\ 9 & 9 & 9 & 0 & 19 \\ -\infty & 0 & 0 & -\infty & -\infty \\ 9 & 9 & 9 & 0 & 19 \\ -\infty & -\infty & -\infty & -\infty & 10 \end{bmatrix} \tag{4.17}$$

Table 4.1: Initial Tokens (Graph to Matrix)

Tokens	Producer	Consumer	Matrix Identifier
t_1	Switch	Switch	1
t_2	Select	MC	2
t_3	MC	MC	3
t_4	Select	Select	4
t_5	Source	Source	5

We are now ready to start generating the state-space for our example FSM-MCDF.

4.3.2 State-Space Generation

Having a defined FSM-MCDF, with a modal graph per mode of operation and a *transition graph* (Finite State Machine), and having assigned to each modal graph a Max-Plus matrix representation we can then generate the state-space of the application.

The algorithm implemented is based on a breadth-first search exploration of the possible state transition, Algorithm (3). We defined a state as a pair (q,γ), a combination of a state mode and a time-stamp vector.

input : G Matrixes, MCDF graph, Transition Graph
output: State-Space
BFS:= new Queue();
StSp:= new Graph();
Insert initial states in BFS;
Insert initial states in StSp;
while *BFS not Empty* **do**
  (q, γ) := BFS.remove;
  for *q' = all possible transition from q* **do**
    γ' := $G'_q \times \gamma$;
    γ'_n = Normalize γ';
    w = Norm of γ';
    Add edge ((q,γ),(q',γ'_n), norm) to StSp;
    if *(q',γ'_n') state does not exist in StSp* **then**
      Add (q',γ'_n') node to StSp;
      Add (q',γ'_n') state to BFS;
    end
  end
end

Algorithm 3: Generating the State-Space

We start by adding all the possible initial states to the queue of the BFS algorithm. The algorithm will then run all possible transitions for all the queue states, adding new states if they haven't been visited yet. A state is visited when both the state mode and time-stamp vector already exist in the state-space. The process of adding a state to the state-space consists on verifying if the state does not exist, and calculating the distance between the departure state and the arrival state, to be set as the weight of the edge of the transition. In order to be able to verify that states are equal it is necessary to normalize the time-stamp vectors of the new state. Only by doing so can we compare two states equality. The value of normalization, or the value by which the vector is normalized, is the maximum distance between states.

Definition 18. *(FSM-MCDF Distance) Let t_s be the time-stamp of a state in the state-space, of an FSM-MCDF, the distance between two states is equal to the norm, in (max,+) of the t_s:*

$$\left\|t_s\right\| = max(t_s) \tag{4.18}$$

Every transition made into a new state, that will be added to the state-space, will have

associated a normalized time-stamp. This is because the values in the time-stamp will increase with time, but the distance between tokens of the same iteration will not. Normalizing the time-stamp vectors enables the algorithm to verify if the state with the exact token distances as been visited. Normalization of a (max,+) vector is defined as:

$$t_s \otimes - \left\| t_s \right\| \tag{4.19}$$

When the queue reaches an empty state we stop our state-space exploration and output the result as a graph.

Lets exemplify the building of a sequence in in our state-space. For example mode sequence [1,1,1,...,1], which will lead to a cyclical sequence. We start by transition to mode 1:

$$\left[G_1\right] \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 10 \\ 15 \\ 0 \\ 15 \\ 10 \end{bmatrix} Normalizing, \begin{bmatrix} 10 \\ 15 \\ 0 \\ 15 \\ 10 \end{bmatrix} \otimes (-15) = \begin{bmatrix} -5 \\ 0 \\ -15 \\ 0 \\ -5 \end{bmatrix} \tag{4.20}$$

As we can see this initial transition take 15 times units and is associated with the weight of the transition edge in the state-space. The new state will be added to the queue for later exploration. Again lets transition to mode 1:

$$\left[G_1\right] \begin{bmatrix} -5 \\ 0 \\ -15 \\ 0 \\ -5 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \\ 0 \\ 10 \\ 5 \end{bmatrix} Normalizing, \begin{bmatrix} 5 \\ 10 \\ 0 \\ 10 \\ 5 \end{bmatrix} \otimes (-10) = \begin{bmatrix} -5 \\ 0 \\ -10 \\ 0 \\ -5 \end{bmatrix} \tag{4.21}$$

The weight has now changed to a 5 time unit transition time.Finally lets transition one more time to mode 1:

$$\left[G_1\right] \begin{bmatrix} -5 \\ 0 \\ -15 \\ 0 \\ -5 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \\ 0 \\ 10 \\ 5 \end{bmatrix} Normalizing, \begin{bmatrix} 5 \\ 10 \\ 0 \\ 10 \\ 5 \end{bmatrix} \otimes (-10) = \begin{bmatrix} -5 \\ 0 \\ -10 \\ 0 \\ -5 \end{bmatrix} \tag{4.22}$$

And as expected we have reached the same state as before and we can stop our exploration of this sequence and close the cycle on further transition to mode one with the weight of 10 time units.

We follow the same procedure for all unvisited states in the queue until it is empty and we have reached our final state-space. The full state-space for this example, as well as the infinite mode sequence we exemplified, is depicted in Figure 4.5.

Inter-Modal Dependencies

So far we have described the process to compute the state-space of an FSM-MCDF application that has common initial tokens amongst all modes of operation. However, this is

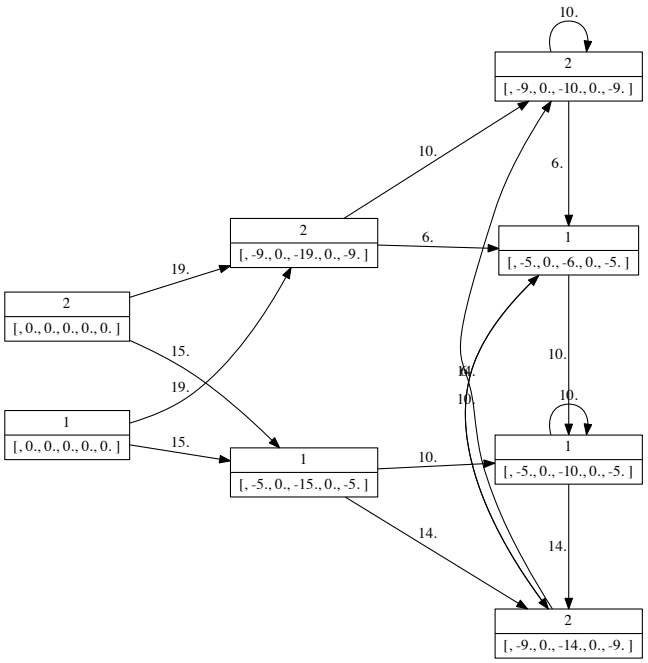

Figure 4.5: State-Space of the example FSM-MCDF

a restrictive requirement for many real-life application graphs. Many application have data dependencies between modes of operations or different initial tokens per mode of operation. For example, a radio application might have different inputs of data for synchronization, decoding and processing operations, and could need information on current frame length from one mode to the next. These constraints impact our technique and have to be addressed.

We have stated before that the matrix representation of each modal graph is done by doing a symbolic execution of the model and keeping time-stamps of the initial tokens present in the graph. However, if the application graph is like the one depicted in Figure 4.6, then different modal graphs will have different dimensions for each matrix, which cannot happen. Keep in mind that we remove all Tunnels and replace them by to self-cycles on each of the Tunnel's linked actors. Both the self-edges share the same initial token, and therefore both these tokens only count as one in our Max-Plus representation.

So the first step to extend our technique to Inter-Modal Dependencies is to dimension our matrices from the original MCDF graph. Therefore, any matrix that represents a modal graph will have dimension $n \times n$, where n is the number of initial tokens in the MCDF graph, in its entirety. For example, the graph in Figure 4.6 has seven initial tokens, so each modal graph's matrix will have dimension 7×7. We have now extended the matrix to accommodate every initial token in the MCDF application, but how do we do so and maintain the correct behavior of the model?

Initial tokens that do not exist in a certain mode are assigned no dependencies in that

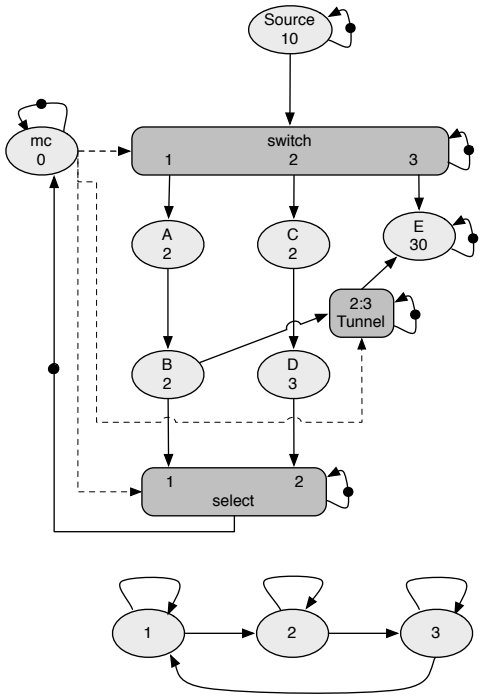

Figure 4.6: More complex MCDF example

mode, all the entries in the matrix are $-\infty$. However, we still need to guarantee that the time-stamps of the modal initial tokens are preserved for subsequent iterations. To do this we share the time-stamp of that tokens production time by allowing a self-dependency in the extended matrix, a zero entry in $i = j = t_i$, where t_i is the modal token's matrix index. This way if the largest time dependency is from a modal token the time-stamp vector will be normalized accordingly.

This is formalized in the following equation:

$$[G_m^{\mathcal{F}}] = \begin{cases} [G_m]_{i,j} & i,j \in I_m \\ 0 & i = j \wedge i,j \in I_m \\ -\infty & i \neq j \wedge i \in I_f \text{ or } I_m \vee j \in I_f \text{ or } I_m \end{cases} \tag{4.23}$$

If we did not consider tokens that are specific to a unique mode in the normalization, in cases where pipelining (overlapping execution of different modes) exists, transition times between modes would be incorrectly estimated. For example, consider MCDF graph in Figure 4.6 that exhibits on mode 3 a slow actor with execution time of 30 time units, while mode 1 and 2 will have a total transition bounded by the execution time of the source, 10 time units. If we travel from Mode 2 to Mode 3 it has a cost of 10 time units of transition time, but if we travel from Mode 3 back to Mode 1, we can also do so after 10 time units (because Actor E does not connect to the Select actor). However, Mode 3 still needs a total of 30 time units to finish the transition. Therefore, the total execution time of mode sequence 2-3-1 is: 10 + (10 + 30) + 0 = 50 time units. If the production time of modal token of Mode 3 is not passed in the time-stamp for the following transition, when traveling from mode 3 to mode 1, the

time-stamp of mode 1 will not consider the remaining execution time of actor E and it will infer that the total execution time of mode sequence 2-3-1 is: 10 + 10 + 10 = 30, which is incorrect as we have seen.

Furthermore, if we have a Tunnel, to model data dependencies between different modes, and don't pass the production time of the token associated to the Tunnel between all mode transitions, when traveling to modes that are not affected by the Tunnel's dependency this information will be lost completely.

For these reasons, we must always carry the modal tokens production times in the time-stamps of any transition in the state-space.

Lets introduce these extended concepts by using an example. Consider the FSM-MCDF graph in figure 4.6 and let G1, G2 and G3 be the (max,+) representation matrix of each mode:

$$G1 = \begin{bmatrix} 0 & 0 & -\infty & 0 & -\infty & 10 & -\infty \\ 4 & 4 & -\infty & 4 & 0 & 14 & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & 0 & -\infty & -\infty & -\infty \\ 4 & 4 & -\infty & 4 & 0 & 14 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 10 & -\infty \\ 4 & 4 & -\infty & 4 & -\infty & 14 & 0 \end{bmatrix} \tag{4.24}$$

$$G2 = \begin{bmatrix} 0 & 0 & -\infty & 0 & -\infty & 10 & -\infty \\ 5 & 5 & -\infty & 5 & 0 & 15 & -\infty \\ -\infty & -\infty & 0 & -\infty & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & 0 & -\infty & -\infty & -\infty \\ 5 & 5 & -\infty & 5 & 0 & 15 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 10 & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \end{bmatrix} \tag{4.25}$$

$$G3 = \begin{bmatrix} 0 & 0 & -\infty & 0 & -\infty & 10 & -\infty \\ -\infty & 0 & -\infty & 0 & 0 & -\infty & -\infty \\ 30 & 30 & 30 & 30 & -\infty & 40 & 30 \\ -\infty & 0 & -\infty & 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & -\infty & 0 & 0 & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty & -\infty & 10 & -\infty \\ -\infty & 0 & -\infty & 0 & -\infty & -\infty & 0 \end{bmatrix} \tag{4.26}$$

The initial tokens for each mode are: $Mode1 : [t_1, t_2, t_4, t_5, t_6, t_7]$, $Mode2 : [t_1, t_2, t_4, t_5, t_6]$ and $Mode3 : [t_1, t_2, t_3, t_4, t_5, t_6, t_7]$. Table 4.2 shows the relation between each matrix index and its corresponding initial token. Notice that, as explained, each matrix with modal initial tokens has a row and line with $-\infty$ entries, except for the self-dependency. As we did previously, lets assume a fixed mode sequence of execution, in order to demonstrate how the extended matrix allows for the modeling of inter-modal dependencies. Lets use the sequence [3, 1, 2, 2], which will provide us with an example with overlapping executions.

Figure 4.7 shows a Gantt Chart of the example's behavior for that specific Mode Sequence. It is easy to see the overlapping of modes, assuming that the task does not use the same resources, and that the execution time of the Mode Sequence is 45 time units.

Table 4.2: Initial Tokens (Graph to Matrix)

Tokens	Producer	Consumer	Matrix Identifier
t_1	Switch	Switch	1
t_2	Select	MC	2
t_3	E	E	3
t_4	MC	MC	4
t_5	Select	Select	5
t_6	Source	Source	6
t_7	Tunnel	Tunnel	7

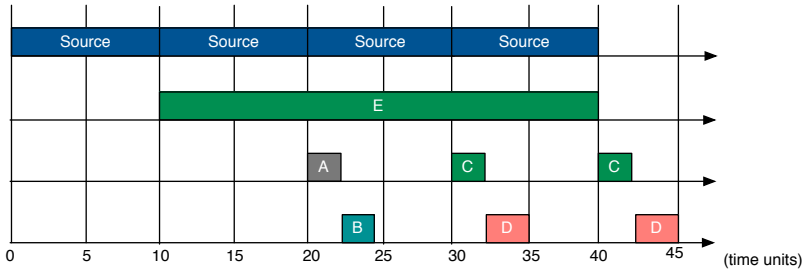

Figure 4.7: Gantt Chart of MCDF example in Figure 4.6 with Mode Sequence [3,1,2,2]

Lets now build the state-space for the sequence using matrices G1, G2 and G3. Lets start by transiting to mode 3

$$[G_3]\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}10\\0\\40\\0\\0\\10\\0\end{bmatrix} Normalizing, \begin{bmatrix}10\\0\\40\\0\\0\\10\\0\end{bmatrix} \otimes (-40) = \begin{bmatrix}-30\\-40\\0\\-40\\-40\\-30\\-40\end{bmatrix} \tag{4.27}$$

And from mode 3 to 1:

$$[G_1]\begin{bmatrix}-30\\-40\\0\\-40\\-40\\-30\\-40\end{bmatrix} = \begin{bmatrix}-20\\-16\\0\\-40\\-16\\-10\\-16\end{bmatrix} Normalizing, \begin{bmatrix}-20\\-16\\0\\-40\\-16\\-20\\-16\end{bmatrix} \otimes (0) = \begin{bmatrix}-20\\-16\\0\\-40\\-16\\-10\\-16\end{bmatrix} \tag{4.28}$$

And from mode 1 to 2:

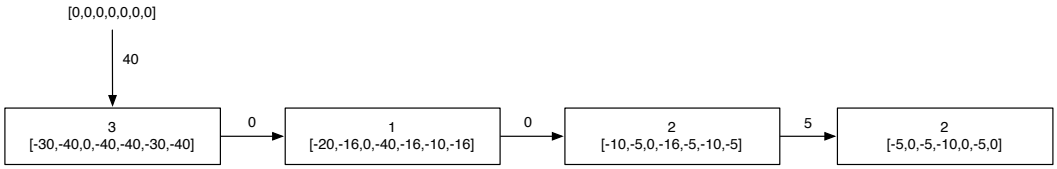

Figure 4.8: Generated State Space for Mode Sequence [3,1,2,2]. Total Execution Time = 45

$$\begin{bmatrix}G_2\end{bmatrix}\begin{bmatrix}-20\\-16\\0\\-40\\-16\\-10\\-16\end{bmatrix} = \begin{bmatrix}-10\\-5\\0\\-16\\-5\\-10\\-5\end{bmatrix} Normalizing, \begin{bmatrix}-10\\-5\\0\\-16\\-5\\-10\\-5\end{bmatrix} \otimes (0) = \begin{bmatrix}-10\\-5\\0\\-16\\-5\\-10\\-5\end{bmatrix} \tag{4.29}$$

And, from mode 2 to 2:

$$\begin{bmatrix}G_2\end{bmatrix}\begin{bmatrix}-10\\-5\\0\\-16\\-5\\-10\\-5\end{bmatrix} = \begin{bmatrix}0\\5\\0\\-5\\5\\0\\5\end{bmatrix} Normalizing, \begin{bmatrix}0\\5\\0\\-5\\5\\0\\5\end{bmatrix} \otimes (-5) = \begin{bmatrix}-5\\0\\-5\\-10\\0\\-5\\0\end{bmatrix} \tag{4.30}$$

Figure 4.8 show the state-space built for our sequence. Notice that some transitions have weight zero. This is due to overlapping of mode operations in the sequence [3,1,2,2], which is captured by preserving the time-stamp of non-initial tokens in every matrix representation of a modal graph. In total this sequence, according to the state space, takes 45 time units to complete which is the expected value according to the Gantt Chart of Figure 4.7.

Limiting the number of transitions

Although the results obtained from this analysis will be complete, all possible mode sequences are explored, they might be pessimistic when used with some applications. Most applications we wish to study fall in the category of real-time streaming applications, such as radio clients/servers. These applications, as we have stated many times, have a quite dynamic behavior with different operations for different phases of their execution. However, some of the modes will not execute in an infinite consecutive sequence, they can have a fixed or limited number of consecutive executions. Both these situations are not taken into consideration, so far, in our analysis. When designing the transition graph of our MCDF model we can easily imply that a specific mode can run consecutively, by adding a self-edge on the node mode,

but not fix or limit the amount of times it can actually run in a row.

In applications that have no such constraints or the *Source* is dominant (the rate of graph is imposed by the rate of the *Source*) this has no impact on the correctness of the results. However, if the application is similar to the one depicted in Figure 4.6, where Mode 3 has an actor E that will impose a rate of 30 in the graph, allowing Mode 3 to run in an infinite sequence or limiting the number of consecutive runs of Mode 3 has a big impact on the determination of the worst-case throughput sequence of the state-sapce. In the first case, allowing for an infinite sequence of Mode 3, will definitely result in an overall worst-case throughput of 30 time units. On the other hand if we limit the amount of runs Mode 3 can have to 3 this value will be lower. Therefore, we added the possibility of specifying a limit on the maximum number of consecutive runs for each Mode. If nothing is said for a specific mode, it is assumed it can run indefinitely.

Notice, that, although this is a small change in our algorithm, it is has a decisive impact on the performance of the worst-case analysis. Results obtained will be tighter to their real value and the analysis is still complete, since the prohibited mode sequences are also impossible to happen in the real application.

Technique Limitations

This analysis technique allows for a full exploration of all the possible mode sequences, according to the *transition graph*, of an MCDF graph. However, there are some limitations to the use of this technique. The building of the state-space can be unbounded due to ever growing differences in production/consumption rates over time. Consecutive transition between a faster and a slower mode will lead to ever growing distances in time-stamp's token production times. For example, if the Source of Figure 4.6 was to be twice as faster, or actor E twice as slower, it would be enough for consecutive transitions from mode 1 to 3 back to 1 would never have the same normalized time-stamp vector, thus the state-space being infinite. In terms of real applications, most times this might not be an issue because a faster production of tokens in one mode might be compensated for a slower production in another mode, or the number of consecutive transition to a specific mode might be limited. Therefore we can only guarantee correct results for *strongly connected* graphs, or if it is known that the graph's behavior is *self-time* bounded. In other words, the graph is periodic or will, eventually, reach a periodic behavior. All the examples and application used were *self-time* bounded graphs.

Solutions for this problem, as well as optimization proposals, are discussed in the conclusion of this chapter, but they were not implemented.

4.3.3 State-Space Analysis

With the state-space generated we can now discuss what results can be taken in a post-analysis.

Throughput Analysis

The most important result we want to have from this technique is the overall Worst-Case Throughput (WCT) for any mode sequence of a MCDF graph. This result can be obtained by calculating the Maximum Cycle Mean (MCM) of the entire state-space, which will compute

the mode sequence with the highest total execution time in the state-space. As the technique is based on simulation the results obtained are exact results of the graph's behavior.

Definition 19. *(Worst-Case Throughput) The worst case throughput of a FSM-MCDF is given as*

$$\min_{\forall \tilde{q}} \lim_{k \to -\infty} \frac{\tilde{q}^{\,k}}{c_{\tilde{q}^k}} \tag{4.31}$$

where $\tilde{q}$ is a sequence specified by the FSM, $\tilde{q}^k$ is the first $k \in \mathbb{N}$ elements of $\tilde{q}$ and $c_{\tilde{q}^k} \in \mathbb{N}$ is the completion time of $\tilde{q}^k$.

Although the WCT is the most important result we wanted to analyze, there are more interesting results that can be obtained through State-Space analysis. We can find the Best-Case Throughput, by determining the Minimum Cycle Mean or analyze a specific mode sequence's performance.

Latency Analysis

In terms of latency analysis, there are two interesting results one can obtain from the generated state-space. One is to compute the latency of a specific *mode sequence*, and the other is to compute the overall maximum latency for a mode sequence of length n.

The maximum latency for a *mode sequence* q of length n can be obtained by searching the state-space for the largest completion time of any sequence of length n that matches q.

Definition 20. *(Latency of a mode sequence of length* k*) Given state-space ss and let $\tilde{q}^k$ be a* mode sequence *with length k, the maximum latency is equal to:*

$$\max_{\forall \tilde{q}^k \in ss} (c_{\tilde{q}^k}) \tag{4.32}$$

where $c_{\tilde{q}^k}$ is the completion time of a sequence $\tilde{q}^k$ with length k.

4.4 Results

4.4.1 Validation

Before we can evaluate the performance of FSM-MCDF compared with current used analysis, we must first make sure it produces correct results. For this purpose, we designed three variation of the simple FSM-MCDF graph in Figure 4.3 and 4.4. This analysis of the graph will result in a small state-space where we can properly validate the correctness of every step. Furthermore, the variations of the example we will use will cover the two distinct *corner cases* in our analysis: intermodal dependencies and pipelining executions.

As we want to study both the effects of our initial analysis and the improvements made in section 4.3.2, we will run all examples with two different transition graphs. Transition graph T1 (Figure 4.9 a)) will allow for any transition for any consecutive amount of runs, while Transition Graph T2 (Figure 4.9 b)) will have a limit of 2 consecutive runs on Mode 2.

We now present all the cases we wish to analyze. All cases were manually solved, with exception of case 2, in order to have expected values for the outputted results. Case 2, due to generating a very large state-space, could not be analyzed in this fashion.

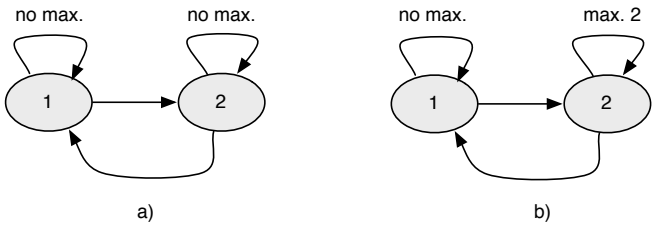

Figure 4.9: a) Transition Graph T1 - b) Transition Graph T2

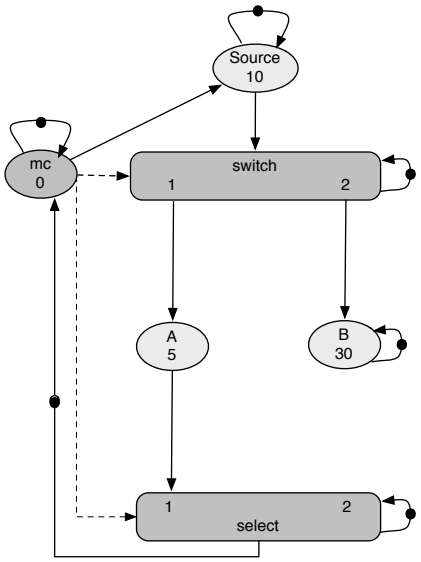

Figure 4.10: MCDF graph example for Case 2

Case 1: Simple MCDF Graph

This is the exact example case (Figure 4.3) we solved when explaining the analysis in the previous section. We expect the state-space graph of Figure 4.5 and an overall worst-case throughput of 10 time units, the same as the period of actor Source. As this example has no pipelining executions using the general transition graph (Figure 4.9 a)) or the transition graph with limited executions of Mode 2 (Figure 4.9 b)), should yield the same results.

Case 2: Simple MCDF Graph with Pipelining

Figure 4.10 is one of the example *corner cases* we wish to explore. Mode 2 has now a very slow actor B, 30 time units, that doesn't connect to the Select actor. This allows for Mode 1 to be able to run while Mode 2 hasn't finished an iteration. The expected results will now depend on how we assume our transition graph. If we use the transition graph T1 we expect an overall worst-case throughput of 30 times units, the same as the period of Mode 2. On the other hand, if we use transition graph T2 then we will be able to see the advantages of pipelining execution and have a tighter value for the worst-case throughput.

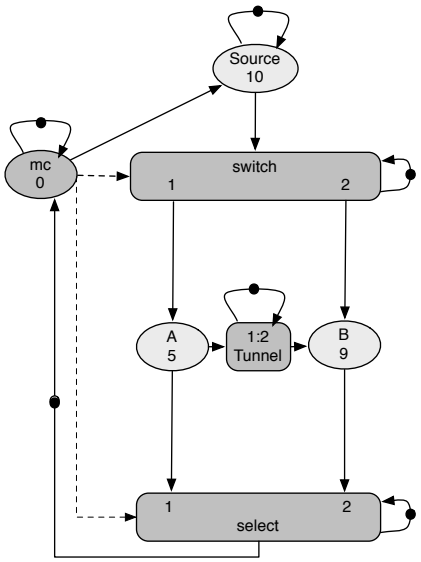

Figure 4.11: MCDF graph example for Case 3

	Transition Graph	Expected WCT	Number of Nodes	Number of Edges	Obtained WCT
Case 1	T1	10	8	16	10
	T2	10	9	13	10
Case 2	T1	30	91	182	30
	T2	<30	64	98	20
Case 3	T1	10	8	16	10
	T2	10	9	13	10

Figure 4.12: Validation results for FSM-MCDF State Space Analysis

---

Case 3: Simple MCDF Graph with Intermodal Dependencies

The example in Figure 4.11 is another of our *corner cases*. We have modeled an intermodal dependency, between Mode 1 and Mode 2, by adding a tunnel between actors A and B. As there is no added pipelining executions in this situation we are expecting the same results as in Case 1. However, we should be able to see on the resulting state-space evidence of the intermodal dependency between Mode 1 and Mode 2 transitions.

Results

Table 4.12 summarizes the results obtained by running the examples from the three chosen cases. We can conclude that the results are as expected in all scenarios. We can also notice that in Case 1 and Case 3 the state-space is quite small, while in Case 2 the number of nodes and edges is comparably higher. This can be justified by the fact that the *source cycle* is not dominant in the graph and, therefore, more states can be created until periodicity is reached. In other words, in Case 2 we will have more states due to a longer transient phase in the

graph's execution. Consequently analyzing the state-space is quite difficult when not done in an automated fashion. Although we could not place the state-space in this graph due to its size, when analyzed it was possible to find several sequences that showed the correct behavior of pipelining execution of Mode 2. All *mode sequences* with transitions 2-1 are executed with *transition time* zero, as expected.

As for the state-space outputs of Case 1 and Case 3, they are the same, with the difference of the extra-token modeling the intermodal dependency. As in both cases, the *source cycle* is dominant in the graph the temporal behavior is the same.

This initial run of experiments served as a validation step before the actual comparison with other throughput analysis techniques for MCDF. We concluded that the outputted results are as expected but that the algorithm will produce quite large state-spaces even for simple input graphs. This poses an issue on the scalability and performance of the algorithm on very large and complex applications.

4.4.2 Comparison

In this section we will show practical results of real applications using the FSM-MCDF analysis and compare it with the current available techniques. We will compare FSM-MCDF with other two methods of obtaining the WCT of an application: Static Dataflow Techniques (SDT) and a Static Periodic Schedule approximation (SPS-AP).

SDT is simply calculating the Maximum Cycle Mean of the original MCDF graph, while SPS-AP separates the MCDF graph into its modal graph representation and then approximates a Static Periodic Schedule for a specific Mode Sequence [28].

SDT and FSM-MCDF are directly comparable because both results are an overall result of all possible mode sequences [28]. However, FSM-MCDF and SPS-AP are not directly comparable because SPS-AP only outputs results for a given finite mode sequence. In the later case, we will compare both techniques by comparing latency analysis results output for specific finite mode sequences.

FSM-MCDF vs SDT

For this comparison we use all the examples used in the previous *validation* section, plus the example on Figure 4.6, which we will denominate Case 4, and two real application models of a WLAN radio (Figure 2.3), one with a slow source and one with a fast source for pipelining executions. In this situation we do not use the same transition graph T1 and T2 as in the previous subsection. Instead, we use each application appropriate transition graph with no limits on transition T1 and with correct real limits for each transition T2.

Analyzing Table 4.13 it is easy to see that both analysis have the same performance when determining the overall worst-case throughput (WCT) when FSM-MCDF does not take into consideration limits on the maximum number of consecutive transition per mode. However, when this factor is taken into account, applications that exhibit this behavior, Case 2, Case 4 and the Fast WLAN, have tighter WCT results with FSM-MCDF analysis.

	Transition Graph	SDT	FSM-MCDF	Improvement
Case 1	T1	10	10	0%
	T2	10	10	0%
Case 2	T1	30	30	0%
	T2	30	20	33%
Case 3	T1	10	10	0%
	T2	10	10	0%
Case 4	T1	30	30	0%
	T2	30	18	40%
Slow WLAN	T1	4000	4000	0%
	T2	4000	4000	0%
Fast WLAN	T1	4000	3000	25%
	T2	4000	3000	25%

Figure 4.13: Comparison results between FSM-MCDF and SDT analysis

FSM-MCDF vs SPS-AP

For this comparison we choose two real application models in MCDF: a WLAN radio (Figure 2.3) and a LTE radio (Figure 4.14). As it is impossible to directly compare both analysis, as SPS-AP is not a complete analysis, we opt to compare both techniques in terms of latency analysis. To perform latency analysis specific mode sequences will be chosen for each application. We will then test the worst-case latency that each mode sequence generates in both analysis. In the WLAN application we test a single mode sequence, while in the LTE model we tryout all the possible mode sequences for the current release of the model. The LTE model we tested can have up to 4 different static mode sequences, therefore we obtain results for each of them.

Observing the results gathered in Figure 4.15 we see that in all *mode sequences* analyzed FSM-MCDF returns tighter latency values. This is expected, since SPS-AP bases its analysis on a Static Periodic Scheduling for the graph and mode transitions, while FSM-MCDF uses a self-time simulator to characterize mode executions and mode transitions. On the particular case of the WLAN, we see that the improvement is quite insignificant. However, this is due to the WLAN application being having a periodic behavior which leads to an almost identical schedules for both Static Periodic and Self Time scheduling schemes. On the other hand, the LTE model we analyze exhibits several pipelining behavior between mode transition, therefore assuming a Static Periodic schedule to characterize mode execution and mode transition will lead to very pessimistic results, as we see in Figure 4.15.

4.4.3 Conclusions

We validated our FSM-MCDF implementation and verified its improvement in terms of temporal analysis of MCDF graphs. We consistently proved that if a MCDF graph is designed with pipelining behavior between modes, FSM-MCDF analysis returns much tighter results than other analysis, in terms of throughput and latency analysis. Furthermore, we concluded that FSM-MCDF is a complete analysis, in the sense that it returns an overall result for any *mode sequence* generated by the finite state machine (FSM). However, we also established that

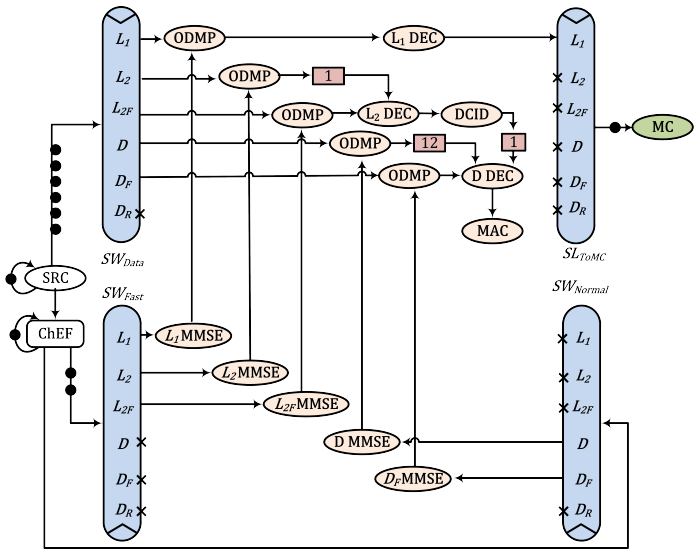

Figure 4.14: LTE MCDF model graph

	Latency Results		
	SPS-AP	FSM-MCDF	Improvement
WLAN	790920	786380	1%
LTE	3932	2424	38%
	3767	2424	36%
	3932	2514	36%
	3767	2513	33%

Figure 4.15: Comparison results between FSM-MCDF and SPS-AP analysis

FSM-MCDF is only an optimal analysis technique if the FSM that describes the MCDF graph is also optimal. For cases were consecutive mode transitions are limited we can simply add a limit on the transition number. However, we did not address the issue if there are particular cyclical mode transitions that have also limited consecutive occurrences. For example, cases where consecutive transition cycle 1-2-1 has limited occurrences. As future work, we propose that a consistent automated algorithm is designed to build the FSM of a MCDF graph. We suggest that the behavior of the graph is expressed as a general *mode sequence* with limits on each mode transition and each cyclical mode transition. Furthermore, it is important to generalize the analysis method to not strongly-connected graphs. A possible way to address this issue is to convert a MCDF graph into its equivalent Strongly Connected Components (SCCs) graph, as proposed in [32].

4.5 Software Implementation

As a result of the implementation of the analysis presented in this chapter, we had to add and modify certain aspects of the Heracles simulator and analysis tool. Specifically we had to introduce the following changes:

- **Create a Max-Plus module -** We added a module to Heracles to deal with matrix structures and all the Max-Plus algebra operations described throughout the chapter.
- **Create and implement the algorithms to build the Max-Plus matrix of a MCDF graph -** We developed and implemented algorithms to analyze the initial, modal and amodal, tokens of a MCDF graph. Moreover, we implemented an algorithm to build a Max-Plus matrix representation of a MCDG graph using the Heracles simulator and the initial tokens of the graph.
- **Edit the simulator to allow for a single iteration execution -** In order to build a Max-Plus matrix of a MCDF graph we need to alter the simulation to allow for a single iteration execution. To keep the versatility of the tool we only activate this functionality is specified by the user when calling the simulator.
- **Create an internal structure for transition graphs -** The basis of this analysis is the description of mode transition by a finite state machine. We opted to implement this as a transition graph that describes each mode as node and each transition as a graph arc. This graph is inputed as a file and parsed into a transition graph structure, within Heracles.
- **Create an internal structure for state-space graphs -** We create an internal structure for the generated state-space as a graph with nodes as mode states and mode transition as edges. Each node as the following parameters: *id, mode, rank and tokens production vector*. Each edge as the following parameters: *transition time and transition number*.
- **Implement the algorithms to build the state-space -** We implemented a breath first search algorithm to explore the transition graph of a MCDF graph and build a state space graph of all the possible mode transitions. The algorithm takes as input the transition graph, tokens table and Max-Plus representation of a specific MCDF graph and build the correspondent state space for future analysis.

- **Create a State-Space analysis module -** We added a new independent module that has the necessary function to analyze the maximum overall throughput and latency of a state-space, or a specific results for a given *mode sequence*.

4.6 Summary

In this chapter we adapted and implemented a state-space analysis algorithm for characterization of the temporal behavior of an MCDF graph, with the use of a finite state machine. We began by exploring the basic concepts of Max-Plus algebra and explaining with detail the fundamental logic behind the analysis algorithm. We then did a step-by-step description on how we implemented the algorithms, the optimizations done and the existing limitations. We validated the final implementation of the FSM-MCDF analysis and compared the results obtained with the current used analysis techniques. FSM-MCDF proved to achieve tighter worst-case throughput results than the other techniques when graphs are modeled with limited consecutive transitions in the same mode, or when pipelining was present in applications. In all other cases, FSM-MCDF proved to be as efficient as SDT analysis.

However, it is important to give emphasis to the fact that FSM-MCDF analysis is a complete analysis. Despite we were only interested in retrieving a worst-case throughput analysis, FSM-MCDF outputs an complete state-space analysis of the execution of a MCDF graph, which can be used to get throughput values for specific mode sequences or in order to achieve best-case throughput analysis.

Chapter 5

Fixed Priority Analysis for SRDF graphs

In a Multi-Processor System-on-Chip (MPSoC) platform running several applications simultaneously, resources must be shared between applications while the timing constraints of each one of them must be met. There are many resource arbitration strategies, such as Time Division Multiplexing (TDM), Round-Robin (RR) or Fixed Priority scheduling. In this chapter we analyze how a fixed priority assignment scheme affects the temporal behavior of streaming applications mapped on a MPSoC. We model streaming applications mapped on a MPSoC with a fixed priority scheme as Single Rate Dataflow graphs (SRDF), and propose a technique to characterize the temporal response behavior of the actors of a mapped application. Specifically, we define *load* of an actor on a processor and derive the worst-case response time analysis of a low priority actor, by characterizing the worst-case load that higher priority actors generate on the same processor, when executing in a self-timed fashion. We also show that in cases where the applications can be modeled using a periodic/sporadic event model, our approach provides tighter worst-case response time bounds than the technique proposed in [18], since we can implicitly take into account the effects of (a)cyclic precedence constraints.

We start by describing how we model streaming applications mapped on a multiprocessor platform with a local fixed priority schedule in SRDF. Then, in Section 2, we propose a technique to analyze the temporal behavior of the simple case of two fixed priority applications. In Section 3 we extend our analysis to n fixed priority applications. In Section 4 we introduce our methodology to apply the worst-case load conditions to an actual fixed priority SRDF graph. In Section 5 we present our experiments and results for interference and response time analysis of a set of fixed priority SRDF graphs. Furthermore, we compare our analysis technique with the one proposed in [18]. Finally, in Section 6 we conclude our work.

5.1 Fixed Priority in SRDF graphs

We model streaming applications as time-bounded SRDF graphs. We define a time-bounded SRDF graph G as:

$$G = (V, E, d, \hat{t}, \check{t}) \tag{5.1}$$

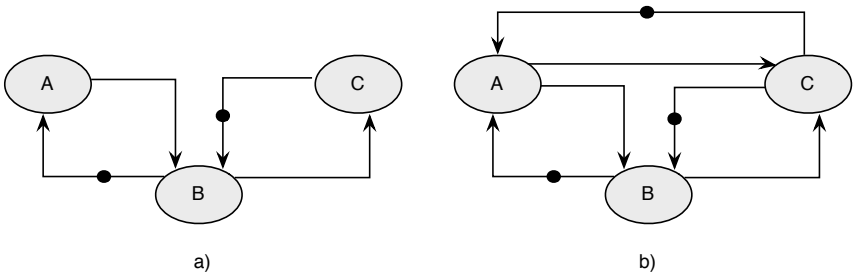

Figure 5.1: Example of enforced static order in actors execution

where V is a set of actors, $E : \{(i,j)|i,j \in V\}$ a set of directed edges and $d : E \to \mathbb{N}$ is a function that describes the initial placement of tokens on the edges; each actor has a worst and best case execution time, respectively, $\hat{t}$ and $\check{t}$. Furthermore, the elapsed time $t(i,k)$ between consumption and production of tokens for any firing k of actor i is defined by execution time $t : V \times \mathbb{N} \to \mathbb{R}^+$. We only consider valid execution times for actor firings such that: $\forall_{k\in\mathbb{N},i\in V} : \check{t}(i) \leq t(i,k) \leq \hat{t}(i)$.

We assume that executions of SRDF graphs always occurs in a self-timed fashion. For a given SRDF graph $G = (V, E, d, \hat{t}, \check{t})$ the start time of an actor firing is denoted by the start time function $s : G \times t \times V \times \mathbb{N} \to \mathbb{R}^+$, such that $s(G,t,i,k)$ denotes the start of the $(k+1)^{th}$ firing of actor $i \in V$ in graph G and execution time function t. When assuming a graph is self-timed, one can fully define $s(i,k)$ with a given graph G and a execution time function t.

Regarding the system platform, we define a MPSoC as a set of processors $\Pi = (\pi_1, \pi_2, ..., \pi_n)$ such that actors $i \in V$ of SRDF graph G are mapped to some processor given by the function $map : V \to \Pi$. Furthermore, we assume a fixed priority scheme where each streaming application running on MPSoC is assigned a unique priority, 1 being the highest value, such that all tasks of the application have the same priority. For simplicity, we associate a graph with a subscript representation G_i, where i is the assigned priority.

A processor, in this scheme, always executes the highest priority active task at any moment in time. Furthermore, to ensure mutual exclusion between tasks of the same application that share the same processor, we assume a pre-defined static ordering per application per processor with a mapping $so : (\Pi, G) \to \alpha^n$, where α^n is the set actor sequences of the type $[i_1, i_2, ..., i_n]$, where $i_1, i_2, ..., i_n \in V$. For instances, if Figure 5.1 a) represents an application such that actors a and c are mapped to the same processor, we model a pre-defined static ordering by appropriately adding edges between the actors to form a non-blocking cycle [28] as shown in Figure 5.1 b). Actors a and c have now a fixed order of execution.

For the rest of the chapter, we assume all SRDF graphs are defined as such and that they are self-timed scheduled. Again, we also assume that all actors, from the same graph, mapped on the same processor are statically ordered. Furthermore, since a processor cannot simultaneously execute more than a single task instance, we add a self-edge with a single token to every actor in the graph to model non self-concurrent execution of firings.

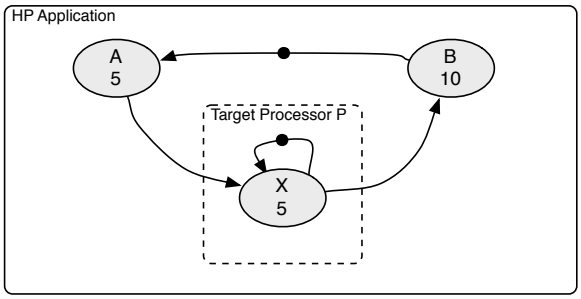

Figure 5.2: SRDF graph example with a single load actor

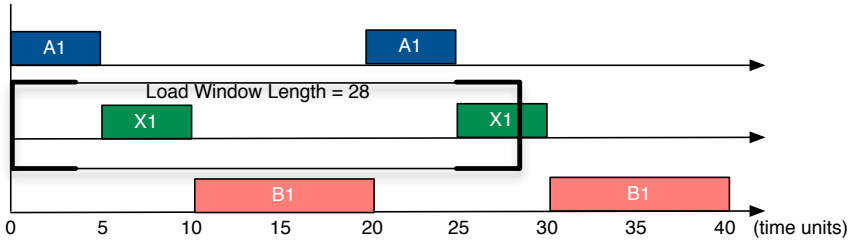

Figure 5.3: Gantt chart of SRDF example

5.2 Fixed Priority Analysis

We start by explaining our proposed interference and response time analysis strategy for the simple case of two self-time scheduled fixed priority applications: A high priority (HP) application and a low priority (LP) applications.

We want to address the issue of having several real-time applications mapped on the same resources. More specifically, analyze how higher priority applications interfere with the execution of lower priority applications. For this purpose, we will extensively use the concept of *load*.

Simply put, **load** is the amount of time a certain resource spends with a task, or a set of tasks, within a given time interval. However, as we are assuming that applications are modeled as SRDF graphs, it is important that we define the concept of *load* created by a graph.

Given a SRDF graph $G = (V, E, d, \hat{t}, \check{t})$ using resources Π, and $m \in \Pi$ a processor, we call *load actor* of m to any actor $i \in V$ with $map(i) = m$, and *non-load actor* of m to all remaining actors of G.

5.2.1 Load of a single load actor

For example, consider the SRDF graph in Figure 5.2. In Figure 5.2 we depict a SRDF graph with three actors, A,B and X. Actor X is a *load actor* of P, while A and B are *non-load actors* of P. Figure 5.3 describes in a Gantt chart two iteration of the self-timed execution of the SRDF example graph.

Now lets assume we want to study the load of processor P due to a single *load actor*, as

Table 5.1: Load generated by actor X

Time interval	Total Load
$[0, 5]$	0
$[0, 10]$	5
$[0, 40]$	10
$[5, 10]$	5
$[0, 28]$	8

in the example graph. In order to determine the load we must first define a time interval, that we refer to as **load window**. Table 5.1 show the total load generated by *load actor* X considering different values for the length of the *load window*.

Analyzing Table 5.1 we see that the *load generated* by a *load actor* is the sum of all firings, of that *load actor*, which occur within the specified *load window*. Moreover, we only consider the amount of time of a firing that occurs inside the window. For instances, in the case of interval [0,28], we consider as *load* of actor X, firing X1 and a partial firing X2 that occurs before the end of the time interval.

We can then define *load* of a single *load actor* on a processor as:

Definition 21. *(Load of a processor due to a single load actor with a schedule s) Given a SRDF graph G, with schedule s, running on resources* Π*, the load of processor m due to single* load actor *i is equal to:*

$$L(G, m, i, s, t, \delta_0, \delta) = \sum_k min\Big(s(G, t, i, k) + t(i, k), \delta_0 + \delta\Big) - max\Big(s(G, t, i, k), \delta_0\Big) \quad (5.2)$$

where k is a firing of i such that $k : \delta_0 - t(i, k) < s(G, t, i, k) < \delta_0 + \delta$*; $s(i, k)$ is the start time of the k_{th} firing of i and $t(i, k)$ the execution time of the k_{th} firing of i.*

For simplicity sake, we henceforth reduce equation 5.2 by assuming that graph G, schedule s and execution time function t to be implicit in our definition. Rewriting equation 5.2:

$$L(m, i, \delta_0, \delta) = \sum_k min\Big(s(t, i, k) + t(i, k), \delta_0 + \delta\Big) - max\Big(s(t, i, k), \delta_0\Big) \quad (5.3)$$

5.2.2 Load of a SRDF graph

Consider, now, that we have an application modeled by the graph in Figure 5.4. We now have more than one *load actor* mapped on the same processor. Figure 5.5 depicts the timeline execution of the actors mapped on processor P. As we assume that actors mapped on the same processor, and from the same applications, are statically ordered, the *load* generated in processor P can be determined simply by extending our load definition to all the firings of all the *load actors* of processor P. Therefore,

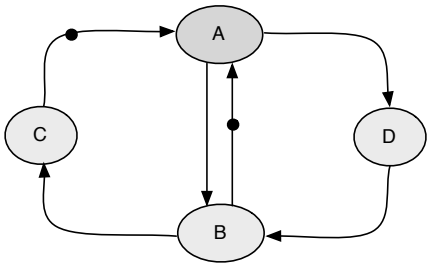

Figure 5.4: SRDF graph example with multiple load actors

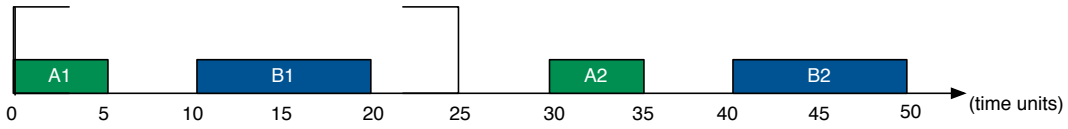

Figure 5.5: Gantt chart of SRDF example Figure 5.4

Definition 22. *(Load of a processor due to a SRDF graph) Given a SRDF graph G, with schedule s, running on resources Π, the load of processor m due to the* load actors *of m is equal to:*

$$L(G, m, s, t, \delta_0, \delta) = \sum_{k,i} min\Big(s(G, t, i, k) + t(i, k), \delta_0 + \delta\Big) - max\Big(s(G, t, i, k), \delta_0\Big) \tag{5.4}$$

where i is a load actor *of m, k is a firing of i such that $k : \delta_0 t(i, k) < s(G, t, i, k) < \delta_0 + \delta$; $s(i, k)$ is the start time of the k_{th} firing of i and $t(i, k)$ the execution time of the k_{th} firing of i.*

5.2.3 Maximum load of a single actor

Now consider an hypothetical characterization of graph $\hat{G}$, schedule $\hat{s}$, execution $\hat{t}$ and some time instant $\hat{\delta_0}$ such that:

$$L(\hat{Q}, m, \hat{s}, \hat{t}, \hat{\delta_0}, \delta) \geq L(Q, m, s, t, \delta_0', \delta), \tag{5.5}$$

that is, if we can realize such a bound for the worst-case load of the data flow graphs of all high priority applications, we can obtain a the worst-case response bound of a lower priority task.

Analyzing the *load function* of Equation 5.3 we can conclude that, in order to characterize the maximum load, we must analyze individually three parameters: 1) the start times of *load* and *non-load* actors, 2) the execution times of *load* and *non-load* actors and 3) the start time of the load window.

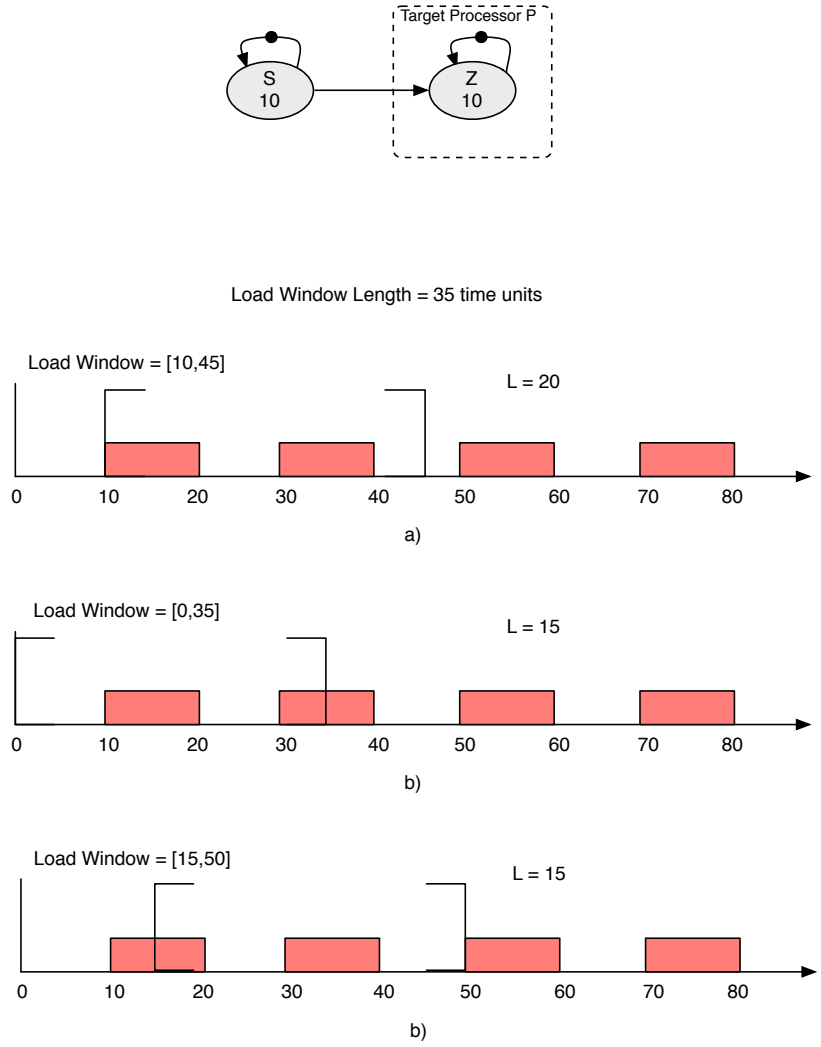

Figure 5.6: Influence of the start time of the load window

In the following subsections, we will analyze each of the parameters that may contribute to the *maximum load* of a single actor. We then summarize the necessary conditions and extend them for the general case of multiple load actors.

Start time of the load window

Given a *load window*, $[\delta_0, \delta_0 + \delta)$ all firings of the load actor within this time interval will contribute to the generated load, L. Therefore it is important to define the start time of the *load window* δ_0 so that the maximum number of firings of the load actor occur within the time window.

As an example, lets analyze three different situations: a) The first firing of the load actor begins at time δ_0, b) The first firing of the load actor begins after the time δ_0 and c) The first firing of the load actor begins before the time δ_0.

We use Figure 5.6 as an example. In the figure we have different starts for a load window

applied to the same timeline of *load actor* X of a processor P.

Case A:

We start by analyzing the most intuitive case, the start time of the load window δ_0 coincides exactly with the start of a *load actor*'s firing. This is the situation described in Figure 5.6 a). Applying Equation 5.3 to the example in 5.6 a) we determine that the total load is equal to 20 time units.

$$\begin{aligned} L(X,P,10,35) =& \Big(min(10+10,10+35) - max(10,10)\Big) + \\ & \Big(min(30+10,10+35) - max(30,10)\Big) = \\ & (20-10) + (40-30) = 10+10 = 20 \end{aligned} \tag{5.6}$$

Case B:

In this case we consider that the start time of the load window, δ_0, begins before the start time of a firing of the load actor. This situation is depicted in Figure 5.6 b). We see that by shifting the load window earlier in time, one of two things can happen: load will be pushed out due to the window's shifting or the load will remain constant because shifting of the window compensates the amount of load lost with load from previous firing.

For the example in 5.6 b):

$$\begin{aligned} L(X,P,0,35) =& \Big(min(10+10,0+35) - max(10,0)\Big) + \\ & \Big(min(30+10,0+35) - max(30,0)\Big) = \\ & (20-10) + (35-30) = 10+5 = 15 \end{aligned} \tag{5.7}$$

Case C:

Case C and B are quite similar. In this case, the start time of the load window is set after the start time of a firing of the load actor, as depicted in Figure 5.6 c). Again, we see that shifting the load window forward will result in two scenarios: the load is pushed out by shifting the window and it is either replaced by including load from a next firing of the load actor or by a slack interval between firings. Therefore, the amount of load will decrease or remain the same.

For the example in 5.6 c):

$$\begin{aligned} L(X,P,15,35) =& \Big(min(10+10,15+35) - max(10,15)\Big) + \\ & \Big(min(30+10,15+35) - max(30,15)\Big) = \\ & (20-15) + (40-30) = 5+10 = 15 \end{aligned} \tag{5.8}$$

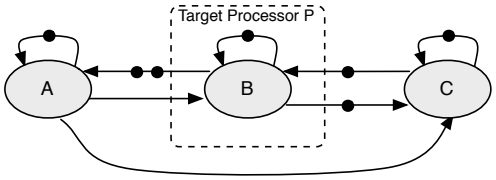

Figure 5.7: Example of two SRDF applications with actors B and X mapped on the same processor.

In summary, we conclude that the load is maximum when the start of the *load window* coincides with the start of a firing of the load actor. In all other scenarios the load can decrease but never increase. We can then characterize the start time of the *load window* that generates the maximum load in the following theorem:

Theorem 2. *For any load window* $[\delta_0, \delta_0 + \delta)$ *in the self-timed execution of a graph,* $\delta_0 \pm \Delta$ *denotes the start time of the first firing of the load actor such that* $L(r, t, \delta_0 \pm \Delta, \delta) \geq L(r, t, \delta_0, \delta)$*, where* $r \in \Pi$*.*

Proof. The start time of a *load window* $[\delta_0, \delta_0 + \delta)$ can either occur when the processor is idle, or when the processor is busy with the load actor. If the load window starts when the processor is idle, then we derive the a new load window of equal length δ whose start coincides with the start of the first firing of the load actor after δ_0 i.e. at time $\delta_0 + \Delta$.

Since the processor is idle from time δ_0 until $\delta_0 + \Delta$, no load will be lost by choosing the load window to start at $\delta_0 + \Delta$. Furthermore, as the load imposed from the end of the original timeline $\delta_0 + \delta$ to the end of the new load window $\delta_0 + \Delta + \delta$ must be non-negative we deduce that the load imposed within the new load window can never be less than the load imposed in the the original load window.

Similarly, if the *load window* $[\delta_0, \delta_0 + \delta)$ starts during the execution of the *load actor*, we derive a new load window whose start coincides with the start of that execution of the load actor at say $\delta_0 - \Delta$. As the processor is busy executing the load actor from time $\delta_0 - \Delta$ up to δ_0 it compensates for any load lost since the new load window ends at $\delta_0 - \Delta + \delta$ instead of $\delta_0 + \delta$. □

Since the above proof applies for any δ_0, we conclude that the maximum load can be characterized such that the start of the load window δ_0 coincides with the start of a firing of the load actor. We now re-write Equation 5.3 as:

$$L(m, t, \delta_0, \delta) = \sum_k \min(s(t, i, k) + t(i, k), t + \delta) - s(t, i, k), \tag{5.9}$$

where $k : \delta_0 \leq s(t, i, k) < \delta_0 + \delta$.

Execution time of the actors

To understand the influence of the execution time of both *load* and *non-load* actors in the *processor load* we will study the total load for a fixed length *load window* in different

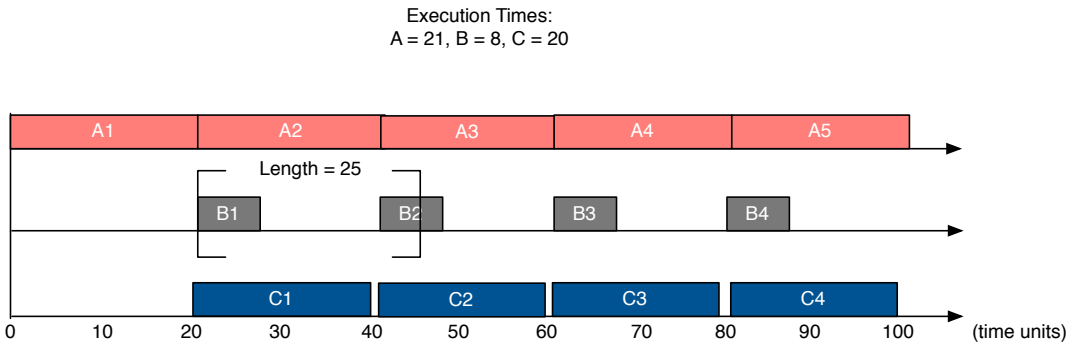

Figure 5.8: Gantt chart of the execution of example SRDF in Figure 5.7

situations. We assume that the start time of the *load window* coincides with the start time of the first firing of *load actor* B.

Considering the starting point of our analysis to be the gantt chart in Figure 5.8, we will assume that the *non-load* actors are executing at its fastest execution time and that *load actors* are running at their slowest pace. In this situation the total load within the defined *load window* is:

$$\begin{aligned} L(B,P,20,25) = &\Big(min(20+8,20+25) - max(20,20)\Big) + \\ &\Big(min(40+8,20+25) - max(40,20)\Big) = \\ &(28-20) + (45-40) = 8+5 = 13 \end{aligned} \tag{5.10}$$

We now derive conclusions by first analyzing the effects of varying the execution times of *non-load actors* and then of *load actors*.

Non-load actors

In this case, we slow down the execution of the *non-load actors* by increasing their execution time. The results can be seen in Figure 5.9. The total load in this situation equal to:

$$\begin{aligned} L(B,P,30,25) = &\Big(min(30+8,20+25) - max(30,30)\Big) \\ &(38-30) = 8 \end{aligned} \tag{5.11}$$

We see that by reducing the execution time of *non-load actors* the generated load is lower. Therefore we state:

Theorem 3. *Within any load window* $[\delta_0, \delta_0 + \delta)$ *slower execution for the firings of any non-load actor cannot increase the load imposed by the load actor.*

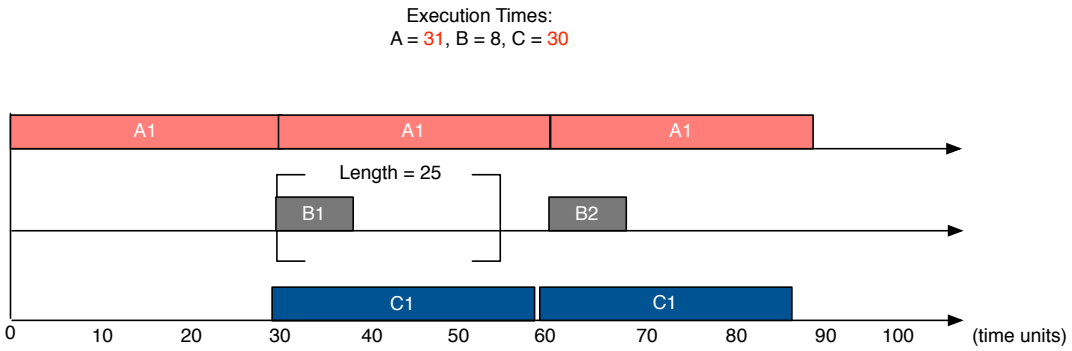

Figure 5.9: Execution of example SRDF in Figure 5.7 with slower execution of non-load actors

Proof. The time intervals between consecutive firings of the load actor i are caused by data dependencies of the load actor on some other actor(s). The start of a firing of the load actor is constrained by:

$$\forall_{j:(j,i)\in E} s(t,i,k) \geq s(t,j,i-d(i,j)) + t(j,k), \tag{5.12}$$

Monotonicity of actor firings implies that slower execution of an actor may only cause delayed firing of the next actor. Assume a firing $h = k - d(i,j)$ of actor j, where $(j,i) \in E$, executes slower such that $t'(i,h) = t(i,h) + \Delta_{i,h}$ and $\Delta_{x,h} \geq 0$. This may only imply that the k_{th} firing of the load actor is constraint such that $s(t,i,k) \leq s(t',i,k) \leq s(t,i,k) + \Delta_{j,h}$. If we assume that every actor except our load actor executes faster, it may only affect the start time of the load actor such that $s(t',i,k) \geq s(t,i,k)$. Due to this delay in firings of the load actor, there may be some load component that was previously within the given load window which may now be pushed outside it.

Let Δ_n denotes the shift in n-th firing of the load actor which is originally the first firing of the load actor outside the given load window such that $s(t',i,n) - s(t,i,n) = \Delta_n$ where $\Delta_n \geq 0$. The shift Δ_n implies that the last Δ_n time within the load window has now been pushed outside and therefore the load imposed within this Δ_n time must now be reduce from the original load to compute the new load.

$$L(t',r,\delta_0,\delta) = L(t,r,\delta_0,\delta) - L(t,r,\delta_0+\delta-\Delta_n,\Delta_n), \tag{5.13}$$

where $r = map(i)$. Since we know we can never impose a negative load for any load window, i.e. $L(t,r,\delta_0+\delta-\Delta_n,\Delta_n) \geq 0$ we conclude that $L(t',r,t,\delta) \geq L(t,r,t,\delta)$. □

Load actors

Similarly, we will start off from the gantt chart in Figure 5.8 assuming that the *load* actors execution time is at its slowest rate, and increase the execution time of *load actor* B. The results can be seen in Figure 5.10. The total load is now equal to:

$$\begin{aligned} L(X,P,20,45) = &\Big(min(20+5,20+25) - max(20,20)\Big) + \\ &\Big(min(42+5,20+25) - max(42,20)\Big) = \\ &(25-20) + (45-42) = 5+3 = 8 \end{aligned} \tag{5.14}$$

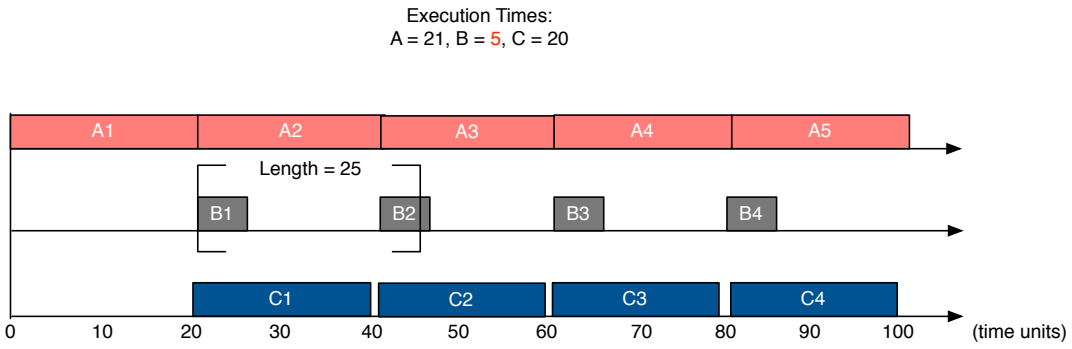

Figure 5.10: Execution of example SRDF in Figure 5.7 with faster execution of load actors

We see that decreasing the execution time of *load actors* does not increase the total generated load. Therefore,

Theorem 4. *Within any load window* $[\delta_0, \delta_0 + \delta)$ *faster execution of the load actor firings cannot increase the load imposed by the load actor.*

Proof. Consider that the firings of the load actor i may execute faster than originally defined by $t(i,k)$. We express these new execution times as $t'(i,k) = t(i,k) - \Delta_k$ where $\Delta_k \geq 0$. The *monotonicity of actor firings* implies that faster execution of an actor firing can only result in sooner arrival of input for the next firing:

$$s(t',i,k) \leq s(t,i,k) \wedge t'(i,k) \leq t(i,k) \Rightarrow s(t',i,k+1) \leq s(t,i,k+1) \tag{5.15}$$

We may also bound the earliest possible arrival of the next firing of the actor using *linearity of actor firing* as:

$$s(t,i,k+1) - s(t',i,k+1) \leq s(t,i,k) - s(t',i,k) + \Delta_k, \tag{5.16}$$

where $\Delta_k = t(i,k) - t'(i,k)$. Since we only consider faster execution for the load actor within the given load window $[\delta_0, \delta_0 + \delta)$, we observe that start of the first firing of the load actor (say $s(t,i,h)$) within the given load window does not change, i.e. $s(t',i,h) = s(t,i,h)$.

Let the h_{th} firing of the load actor be originally its first firing outside the load window (i.e. $s(t,i,h) \geq \delta_0 + \delta$) such that it does not impose load on the processor within the load window $[\delta_0, \delta_0 + \delta)$. However, since we now consider that the firings of the load actor to execute faster, it may be that the h_{th} firing also starts sooner such that $s(t',i,h) < \delta_0 + \delta$. We apply Equation 5.16 recursively to deduce that the soonest possible arrival of the h_{th} firing is expressed as:

$$\begin{aligned} s(t,i,h) - s(t',i,h) &\leq s(t,i,n) - s(t',i,n) + \sum_{n \leq k < h} \Delta_k \\ &\leq \sum_{n \leq k < h} \Delta_k \end{aligned} \tag{5.17}$$

Since the h-th firing is now inside the load window, we must also compute the load imposed by it as well as any other firing that might now be inside the load window. If originally the h-th iteration is just outside the load window ($s(t,i,h) = \delta_0 + \delta$), Equation 5.17 implies that the soonest possible start of h-th firing is $s(t',i,h) = \delta_0 + \delta - \sum_k \Delta_k$ where $n \leq k < h$. To compute the load imposed by the load actor within the given load window, we need to sum the new load components for all original firings within the given load window, and the load imposed by firings k onward until the end of the load window. The total load for the load window $[\delta_0, \delta_0 + \delta)$ for the faster execution setting can be expressed as:

$$L(t', \delta_0, \delta) = L(t', \delta_0, s'(i,k) - \delta_0) + L(t', s(t',i,h), \delta_0 + \delta - s(t,i,h)), \tag{5.18}$$

Considering faster execution times $t'(i,k) = t(i,k) - \Delta_k$ for each firing $n \leq k < h$ of the load actor, we deduce that:

$$L(t', \delta_0, s(t',i,h) - \delta_0) = L(t, \delta_0, \delta) - \sum_{n \leq k < h} \Delta_k \tag{5.19}$$

According to our definition of the load function we know that the maximum possible load that can be imposed within a load window is the length of that load window, i.e.

$$\begin{aligned} L(t', s(t',i,h), \delta_0 + \delta - s(t',i,h)) &\leq \delta_0 + \delta - s(t',i,h) \\ &\leq \sum_{n \leq k < h} \Delta_k. \end{aligned} \tag{5.20}$$

Placing these observations in Equation 5.18, we deduce:

$$\begin{aligned} L(t', \delta_0, \delta) &= L(t, \delta_0, \delta) - \sum_{n \leq k \leq h} \Delta_k + L(t', s(t',i,h), \delta_0 + \delta - s(t',i,h)) \\ &\leq L(t, \delta_0, \delta). \end{aligned} \tag{5.21}$$

We conclude that reducing the execution time of firings of the load actor cannot increase the load imposed in the given load window. In other words, the maximum load imposed by the load actor must assume the worst-case execution time for each firing of the load actor in the load window $[\delta_0, \delta_0 + \delta)$. □

Start times of the actors

In [28], Moreira defines *dependence distance* $dd(i,j)$ as the number of firings of actor $j \in V$ that can occur before the first firing of actor $i \in V$, in an admissible schedule. That is, for any admissible schedule, $s(j, dd(i,j)) \geq s(i,0) + t(i,0)$. Because SRDF actors have unary rates of production and consumption, it also means that $s(j, k + dd(i,j)) \geq s(i,k) + t(i,k)$.

Furthermore, [28] shows that, given a self-timed schedule s for a strongly connected SRDF graph we can construct a new schedule s' such that all firings are self-timed except the n-th firing of the an actor l, which is instead forced to fire at $s'(l,n) = s(l,n) + \phi$. The constructed new schedule is then such that for $i \in V$:

$$s(i,k) \leq \begin{cases} s'(i,k) \leq s(i,k) + \phi, & \text{if } k - n = dd(i,l) \\ s'(i,k) = s(i,k), & \text{if } k - n \neq dd(i,l) \end{cases} \tag{5.22}$$

If we forcibly delay the n-th firing of an actor j such that $s'(j,n) = s(j,n) + \phi$ considering a sufficiently large ϕ we can construct a schedule s' in which for all $i \in V$ and $k - n \leq dd(j,i)$, $s'(j,n) \geq s'(i,k)$. In other words, we intentionally delay the n-th firing of j until all firing that do not depend on its finishing such that any future firings must wait for the n-th firing of j. We then define the constructed schedule s' as **blocked** on the n-th firing of j. As any progress in execution of the graph is only possible after the n-th firing of j, any further delay in this firing will equivalently delay all future firings:

Theorem 5. *For the admissible schedules s' and s'' constructed from a self-timed schedule s of a given SRDF graph, if both s' and s'' are blocked on the n-th firing of an actor $l \in V$ such that $s'(l,n) = s(l,n) + \phi$ and $s''(l,n) = s(l,n) + \Phi$, it holds that for all $i \in V$ and $k - n \geq dd(l,i)$*

$$s''(i,k) = s'(i,k) + \Phi - \phi \tag{5.23}$$

Theorem 5 implies that the minimum distance between firings $s(j,h)$ and $s(i,k)$ where $k - h \leq dd(j,i)$ and $j, i \in V$ is obtained by constructing a schedule that is blocked on the firing $s(j,h)$. Without loss of generality we apply the blocking concept to the first firing to obtain the maximum possible firings of the load actor within a given load window:

Theorem 6. *For a given strongly connected SRDF graph, the minimum distance between two firings of an actor i in a self-timed schedule s is bounded by constructing schedule s' from s that is blocked on the first firing of that actor:*

$$s(i,k+h) - s(i,k) \geq s'(i,h) - s'(i,0) \tag{5.24}$$

From Theorem 6 we deduce that by blocking the first firing of the load actor we obtain the minimum distance to all its future firings and therefore maximize the load imposed by the load actor.

We explain with detail how we apply the blocking concept in practice, and how we determine the necessary blocking time to achieve the maximum load condition, later on, in section 5.4.2.

Gathering all conditions

We conclude our characterization by listing the conditions under which a *load window* within the self-timed execution of a SRDF graph gathers the maximum load for that actor: 1) the start of the load window coincides with a firing of the load actor; 2) by blocking the graph, all future firings activate at their natural distance from the first firing of the load actor in the given load window; 3) we consider the worst-case execution time for each firing of the load actor; and 4) we consider the best-case execution time for each firing of all other actors.

5.2.4 Maximum load of a SRDF graph

So far we have defined the necessary conditions for maximizing the load imposed by a single high priority load actor. However, an application may often have more than one *load* actor mapped on the same processor. If that is the case, then which *load* actor's firing should coincide with the start time of the *load window*?

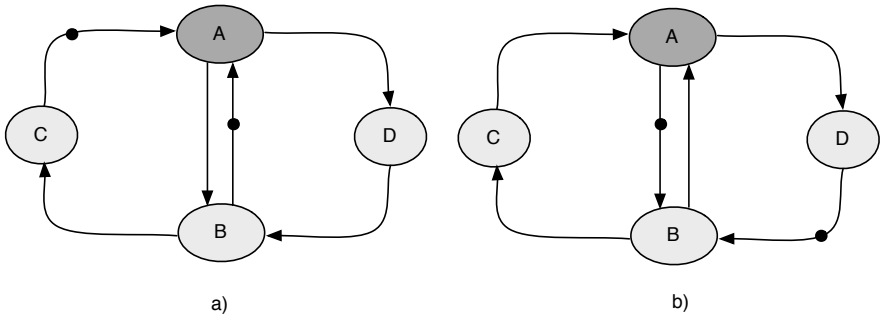

Figure 5.11: Example SRDF graph with cyclical dependencies and a blocked state

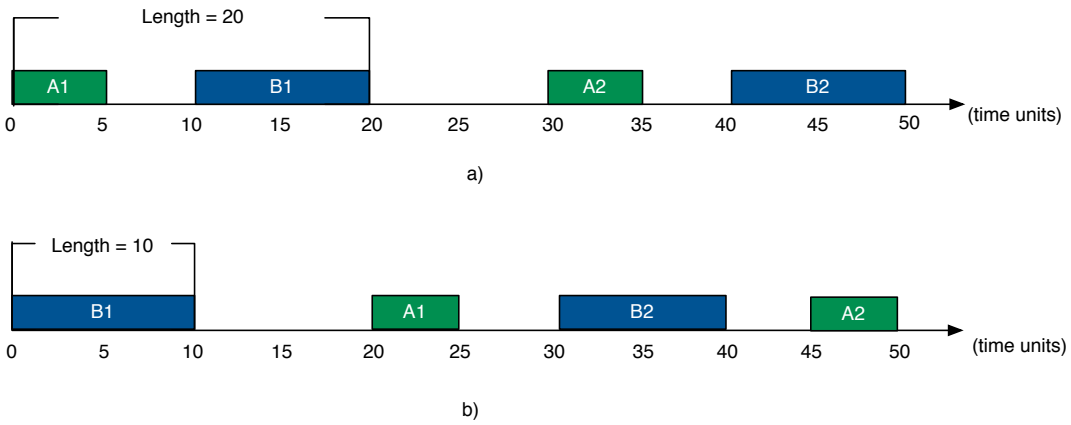

Figure 5.12: Timeline for the example of Figure 5.11

Consider the graph in Figure 5.11 such that actor a and b are mapped on the same processor while actor c and d are mapped to different processors. Lets assume constant execution times $t_a = 5$, $t_b = 10$, $t_c = 10$ and $t_d = 5$. Figure 5.11 a) shows the condition of the graph when all future firings are dependent on actor a, and Figure 5.11 b) shows the condition of the graph when all future firings are dependent on actor b. The timeline for both cases is depicted in Figure 5.12.

Lets now compute the load in each timeline, with the *load window* starting from the first firing of the load actor that fires first. We observe that the load imposed in each case is different and dependent on the length of the load window. For instances, for a *load window* of length $\delta = 20$ the load imposed in Figure 5.12 a) is greater, while for a length of $\delta = 10$ the load imposed is greater in the timeline of Figure 5.12 b).

Theorem 2 states that to obtain the maximum load within a load window, the start of the load window must coincide with the start of a load actor firing. Furthermore, Theorem 5 defines that by sufficiently delaying the n-th firing of a load actor u we minimize the distance of all future firings of all actors (including any other load actors) from this firing. We generalize Theorem 6 such that for all actors $v \in V$ and $k - n \geq dd(u, v)$

$$s(v, k) - s(u, n) \geq s'(v, k) - s'(u, n) \tag{5.25}$$

This implies that the maximum load imposed within a load window whose start coincides with $s(u, n)$. In case we have multiple load actors, we repeat this process for each of the load actors. Without loss of generality, we sufficiently block the first firing of one of the load actors to obtain the maximum load that can be imposed in a given load window assuming that the start of the load window coincides a firing of that load actor. The maximum of the above load value obtained gives us the maximum load imposed by the application in the given load window.

5.2.5 Response Time Analysis

In a fixed priority scheme, the execution of a low priority task can be preempted by the activation of a higher priority task, thus delaying its completion. We can use our maximized load function to build a worst-case response time analysis for any actor in a SRDF graph with a fixed priority scheme.

Lets consider we have a set of two SRDF graphs mapped on the same resources, as in Figure 5.13. We focus on a single resource: processor P. We see that actors X and Z are *load actors* of P. Furthermore, actor X belongs to a high priority (HP) application and Z to a low priority (LP) application. We also assume that the high priority application is running under all the necessary conditions (Section 5.2.3) such that the *load actors* of P produce their maximum load. If we look at the timeline of processor P, depicted in Figure 5.14, we see that the response time of a firing of actor Z is equal to 30 time units, instead of 20 time units. This is due to the interference created by the *load actors* of processor P with higher priority than Z, actor X.

What if the HP application has multiple load actors? Consider the example in Figure 5.15. In this case, our HP application is the same as in the example of section 5.2.5, Figure 5.11. Therefore, as stated in 5.2.5, to determine the worst-case response time, we need to analyze

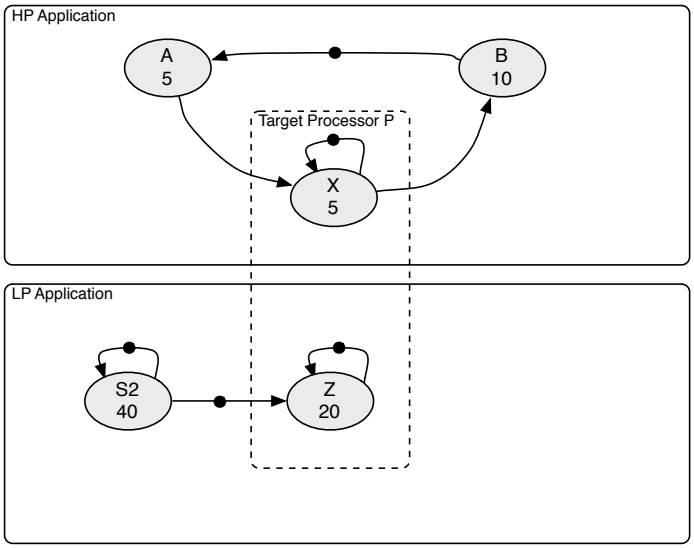

Figure 5.13: Example of interference between two SRDF applications

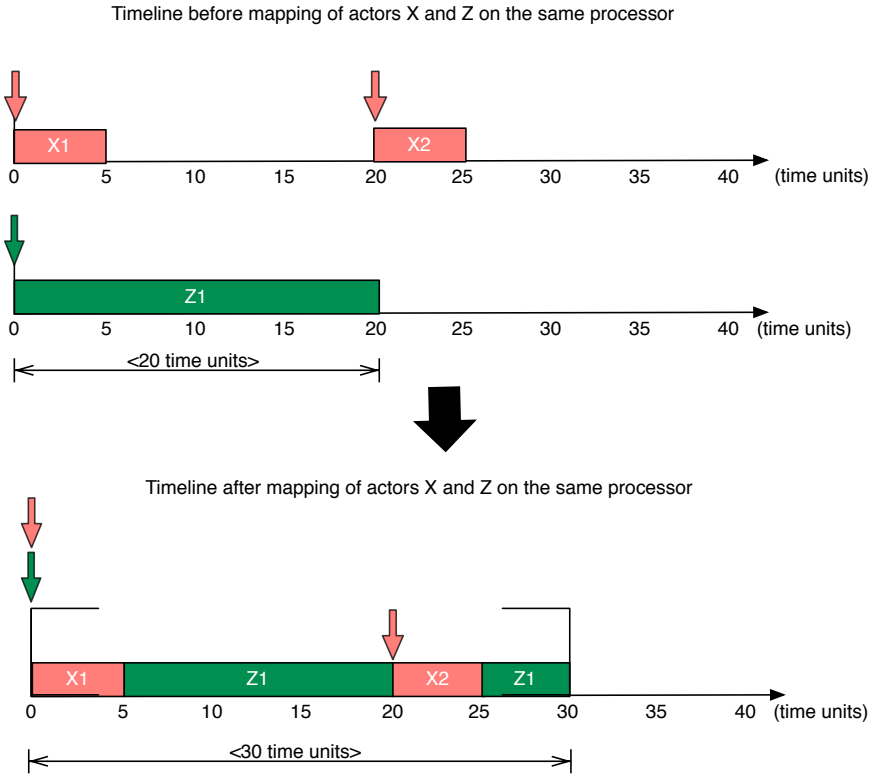

Figure 5.14: Execution of actors X and Z before and after being mapped on the same processor

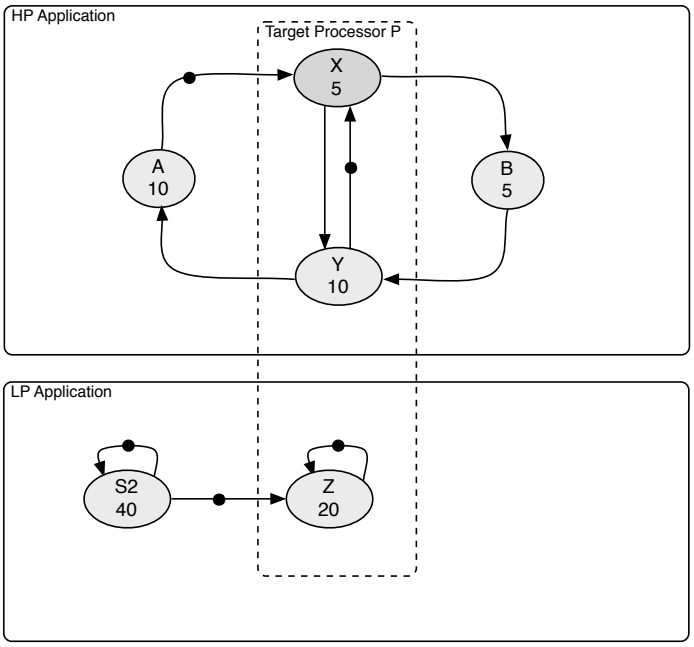

Figure 5.15: Example of interference between two SRDF applications

the maximum load conditions for each of the *load actors* of P, and retrieve the maximum load imposed by the application in the given *load window*.

Figure 5.16 depicts the timelines of the maximum load of processor P, due to different blocking of the *load actors*, and the respective worst-case response time of the low priority actor Z. In Figure 5.16 a), we depict the timeline of processor P, due to the HP application behavior being dependent on *load actor* A. While, in Figure 5.16 b), we depict the timeline of processor P, due to the HP application being dependent on *load actor* B.

Observing Figure 5.16, we conclude that the worst-case response time occurs when the graph is blocked due to *load actor* B, and it is equal to 45 time units.

However, if the execution time of actor Z would be 15 time units, both scenarios would give the same worst-case response time. Therefore, we also conclude that different low priority task may have worst-case response times due to different conditions of the maximum load.

At this point, we define $\hat{L}_i$ as the characterization of *load* function L that maximizes the load of a processor for a specific low priority actor i, such that Equation 5.5 holds.

We can then build the following equation to derive the worst-case response time of a load actor of a SRDF graph:

Definition 23. *(Worst-case response time of a actor) Let G be a high priority SRDF graph on resources Π with schedule s, execution time function t and m a processor of Π. Let i be an independent low priority task with execution time t_i. Then the response time of i, the elapsed time between the start (δ_i) and end of the execution of i, considering interference from the*

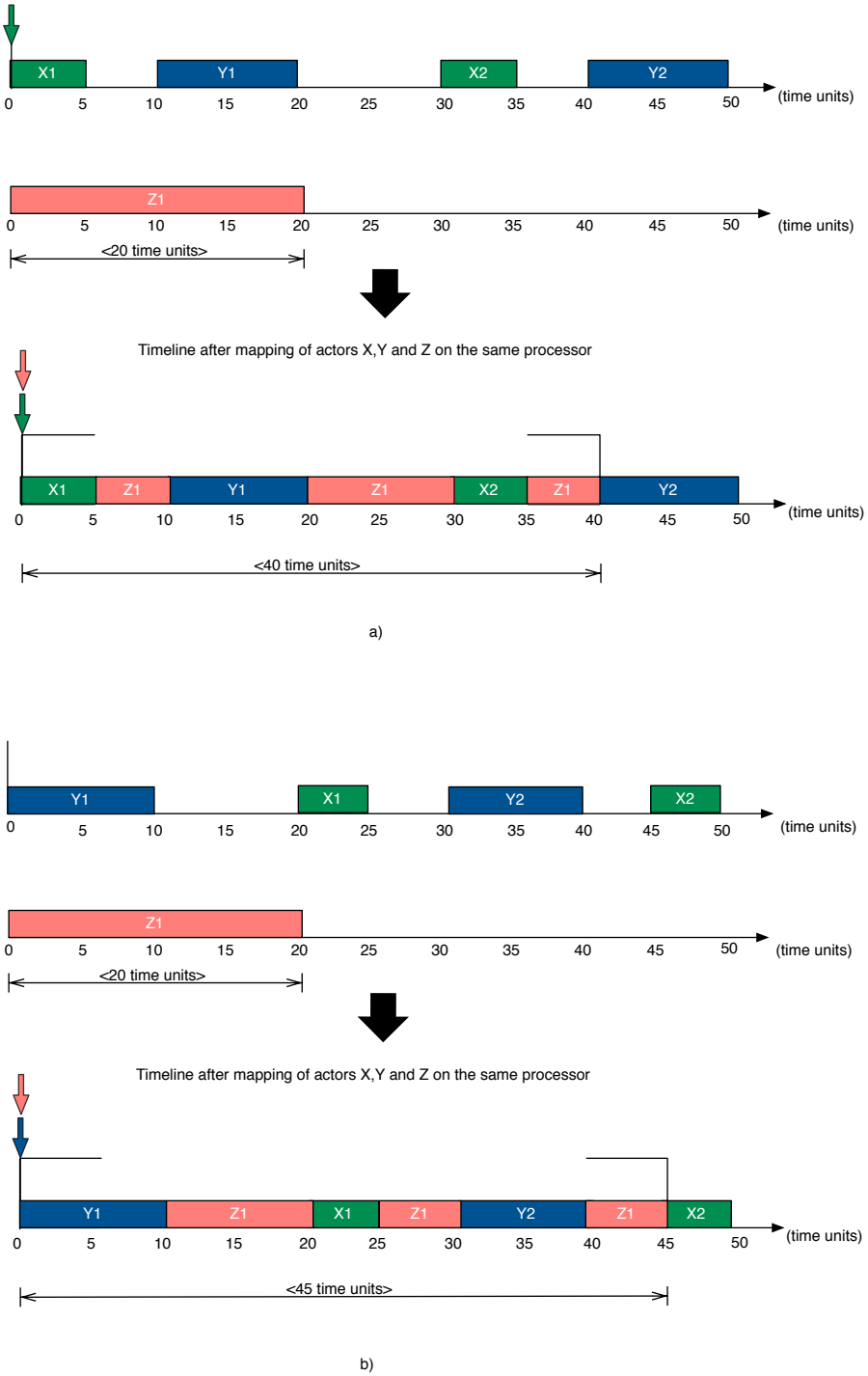

Figure 5.16: Execution of actors X,Y and Z before and after being mapped on the same processor

high priority application, is equal to the smallest positive value $\hat{r}_i$ that satisfies the equation:

$$\hat{r}_i = t_i + \hat{L}_i(G, m, s, t, \delta_i, \hat{r}_i), \tag{5.26}$$

where δ_i is equal to the schedule start time of actor i and $\hat{L}_i$ the maximum load characterization of graph G for actor i.

For instances, if we use equation 5.26 to determine the response time of load actor Z of P in the example of Figure 5.15, we get:

$$\begin{aligned} r_Z = t_Z + \hat{L}_Z\big(P, \delta_Z, r_Z\big) : \\ & 1) r_Z = 20 + \hat{L}_Z(P, 0, 0) = 20 + 0 \\ & 2) r_Z = 20 + \hat{L}_Z(P, 0, 20) = 20 + 10 = 30 \\ & 3) r_Z = 20 + \hat{L}_Z(P, 0, 30) = 20 + 15 = 35 \\ & 4) r_Z = 20 + \hat{L}_Z(P, 0, 35) = 20 + 20 = 40 \\ & 5) r_Z = 20 + \hat{L}_Z(P, 0, 40) = 20 + 25 = 45 \\ & 6) r_Z = 20 + \hat{L}_Z(P, 0, 45) = 20 + 25 = 45 \\ & r_Z = 45 \end{aligned} \tag{5.27}$$

5.2.6 Response Model

We define a simple response model for each actor of an SRDF application by using a single non-concurrent actor with a constant execution time equal to the obtained bound on the worst-case response time using equation 5.26. We can then use the response model to analyze the temporal behavior of an SRDF application.

By replacing each actor of an application graph by its fixed priority response model we obtain a SRDF dataflow graph that represents the worst-case temporal behavior of the entire application, when scheduled with a fixed priority scheme. Therefore, we can apply SRDF temporal analysis techniques, such as Maximum Cycle Mean (MCM) analysis, on the obtained graph to verify if it meets its temporal requirements.

For instances, in the example of Figure 5.13, due to the interference of X, the response time of the low priority actor Z is 30 time units, instead of the expected 20 time units. The expected MCM of the low priority application graph is 40. If we replace actor Z's execution time by its response time, we can evaluate if the application still meets its timing requirements, despite the interference from higher priority applications. Performing MCM analysis on the fixed priority response model application graph, returns an output MCM of 40. Since the MCM remains the same, we can conclude that, despite the interference of actor X, the low priority application still meets its timing constraints.

However, in the example case of Figure 5.15, we have interference due to the load of *load actors* X and Y. In this case, the worst-case response time of actor Z is 45 time units, instead of the expected 20 time units. If we replace actor Z for its fixed priority response mode, the execution time of actor Z is now 45 time units. Performing MCM analysis on the response

model applications graph, returns an output MCM of 45 time units. Since the MCM is above the expected value of 40 we can conclude that the low priority applications, when schedule with the HP application of Figure 5.15, cannot meet its timing requirements.

5.3 Fixed Priority Analysis for N applications

So far we established the necessary definitions and conditions to analyze a fixed priority scheduling of two streaming applications on a MPSoC platform. However, most real cases have more than two running applications. In this section we demonstrate how to extend our response time analysis for the case where we have n streaming applications scheduled with a fixed priority scheme.

Lets extend our response time computation, Definition 23, considering the interference from a set of n higher priority graphs. If $\hat{G}_j$, $\hat{s}$, $\hat{t}$ and $\hat{\delta_0}$ characterize the worst-case load of the j-th high priority application where $1 \leq j \leq n$, we define the worst-case response time bound $\hat{r}_i$ of the low priority task, as the lowest possible value that satisfies the following equation:

$$\hat{r}_i = t_i + \sum_{j=1}^{n} \hat{L}_i(\hat{G}_j, m, \hat{s}, \hat{t}, \hat{\delta_0}, \hat{r}_i), \tag{5.28}$$

where t_i is the execution time of task i and δ_i is equal to the scheduled start time of task i.

Simply put, in order to determine the worst-case response time of a low priority task, when considering interference from the load imposed by a set of n application, we need the following steps. First we characterize the maximum load condition of each application, using the set of conditions of section 5.2.3 and 5.2.4. Then add up the total maximum load, from all applications, within the *load window*, and determine the response time using Equation 5.28.

In the next sections we will describe the process of applying the maximum load condition on an actual SRDF graph and a step-by-step approach on how we implemented our analysis in the Heracles tool.

5.4 Maximum load set-up in a SRDF graph

In the previous section we characterized the conditions under which we can obtain the maximum load for a set of SRDF graphs. Now we need to describe how to apply these conditions to an actual SRDF model graph.

Some are quite easy to recreate: such as the start time of the load window and the execution times of the SRDF actors. All that is needed is to set the time window to start at the first firing of the load actor, and that the graph runs all *non-load* actors in their best-case execution time and all *load* actors in their worst-case execution time. However, to achieve the worst condition possible we still need to have the graph in a condition where all firings of the graph are dependent on the first firing of the load actor.

In this section we explain the method used in our study to have the graph in the necessary conditions for the load actor we want to study to generate the maximum load possible. Later we extend the analysis for the existence of multiple load actors.

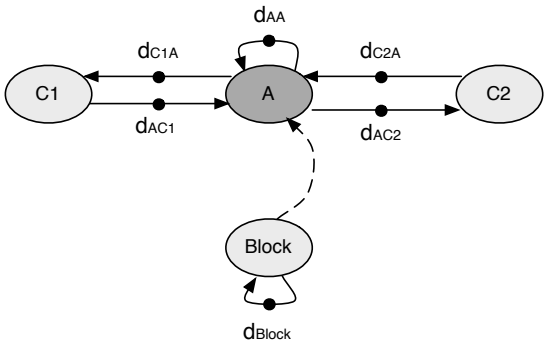

Figure 5.17: Example of the general SRDF graph we want to maximize the load

5.4.1 Methodology

In Figure 5.17 we have a SRDF graph with three actors, A,C1 and C2. This graph represents a condensed model of the type of graphs we intend to study. Actor A represents the high-priority *load actor* that will be responsible for generating the load in the processor it is mapped to, while *non-load* actors C1 and C2 represent all the cyclical dependencies that affect actor A. The graph is strongly connected graph and self-timed, therefore, as stated in Section 2.4, it will have a two phased behavior: when the graph initiates its execution it will go through a *transient phase* until it eventually reaches a *periodic phase*. As soon as the graph enters its *periodic phase* the computational load that a load actor imposes, per period, on its mapped processor becomes constant. Furthermore, the graph assumes that all the *non-load* actors are running on the best-case execution time, and all *load* actors are running on their worst-case execution time.

Actor A will be our load actor and in order to accumulate the largest number of tokens at its input edges before its first firing we will add a new actor to the graph: actor *Block*. The *Block* actor's purpose is to block the load actor A from firing for a certain amount of time, the blocking time, such that the rest of the graph will run and produce tokens in the input edges of the load actor until the graph can no longer run unless the load actor is fired. This way we can assure that the load actor has the maximum possible number of accumulated tokens at its input edges and that, consequently, it will generate the maximum load when it is fired.

With this method, all the necessary conditions for a *load actor* to produce the maximum load possible are met:

- Best-case execution time for *non-load actors*
- Worst-case execution time for *load actors*
- The *load window* starts at the first firing of the *load actor*
- The *load actor* is blocked until the graph is dependent on its firing

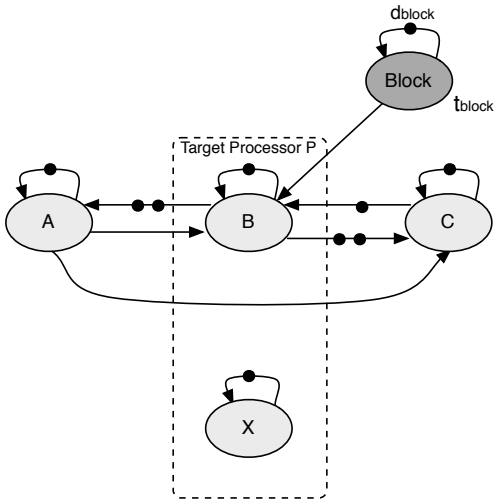

Figure 5.18: Example of Figure 5.7 with a Block actor

Example

Consider the SRDF graph in Figure 5.7 as an example graph. After adding our *Block* actor in the graph, as depicted in Figure 5.18, we will vary the *blocking time* of the *Block* actor and analyze the effect on the load generated by our *load actor* B.

Observing Figure 5.18, we can see that by blocking actor B and allowing the remaining actors to keep executing, the graph will reach a blocked state as depicted in Figure 5.19. We see that without any firing of actor B, the highest number of tokens at the input edges of B is 2. The state of the graph in Figure 5.19 reflects the best possible condition for actor B to generated the maximum load.

Figure 5.20 depicts the gantt chart of the execution of the blocked graph (5.18) with blocking times: 15,25 and 35. As expected the number of consecutive firings of actor B is at most 2, and it is the case that generates the maximum load. Furthermore, blocking the graph by 25 or 35 time units shows no difference in the generated load. Therefore, there is in fact a bound on the necessary blocking time for a strongly-connected SRDF graph.

It is important to give emphasis that, although this technique is conservative, some pessimism might be added if the blocking time reflects a state in the graph that would never happen. In the next subsection we will discuss with detail the parameters of the *Block* actor.

5.4.2 Blocking Actor

The blocking actor is connected only to the load actor and to itself by a self-edge. The two important parameters to configure the blocking actor are the *blocking time*, t_{block}, and the number of delays in the self-edge of the blocking actor, d_{block}. For simplicity sake, lets consider again the example in Figure 5.18. Once the graph is activated the *block actor* creates an extra dependency for the *load actor* B. While the *block actor* does not finish its firing, which will produce d_{block} tokens in edge $(Block - B)$, B cannot fire. Meanwhile, the remaining *non-load* actors will execute until, eventually the graph can no longer run due to pending dependencies from *load actor* B. At this point the whole graph is stopped waiting for the first

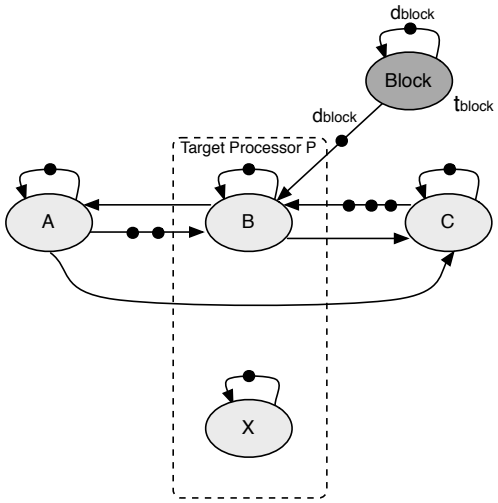

Figure 5.19: State that represents the maximum load in the SRDF graph

firing of B. This will guarantee that the maximum number of tokens is accumulated in the input edges of actor B.

Once the end of the *blocking time* is reached, the block actor produces a sufficient number of tokens on edge $(Block - B)$ to remove its influence for the remainder execution of the graph, and *load actor* B can finally fire. B will then produce the maximum number of firings with a minimum distance, creating the worst-case scenario in terms of processor usage - the **maximum processor load**.

General Case

In the general case, we assume we have a strongly-connected self-timed SRDF graph G and a mapped processor m we wish to study. We need first to determine the *blocking time* and the number of delays of the Block actor. As our assumption is that the remaining actors of the graph are mapped to different processors, and do not depend on our load actor, the *blocking time* can be derived from the maximum time it takes for the graph to stop to wait for our blocked *load actor*. Which means that the slowest cyclical dependency C of the load actor will be the maximum *blocking time* [1].

$$t_{block} \geq C_m \cdot \sum_{i \in C_m} d_i, , \tag{5.29}$$

where i is an actor of G and C_m is the cycle with the largest execution time, $C_m = max(t_{C1}, t_{C2}, ..., t_{Cn})$ $n \in \mathbb{N}$.

This time is an upper bound on the maximum execution time any of the cycles in the graph can run without needing a firing of the *load actor*.

On the other hand, the d_{delay} parameter should be chosen such as: the *load actor* does not have to wait for tokens from the Block actor after the blocking time. We want to avoid

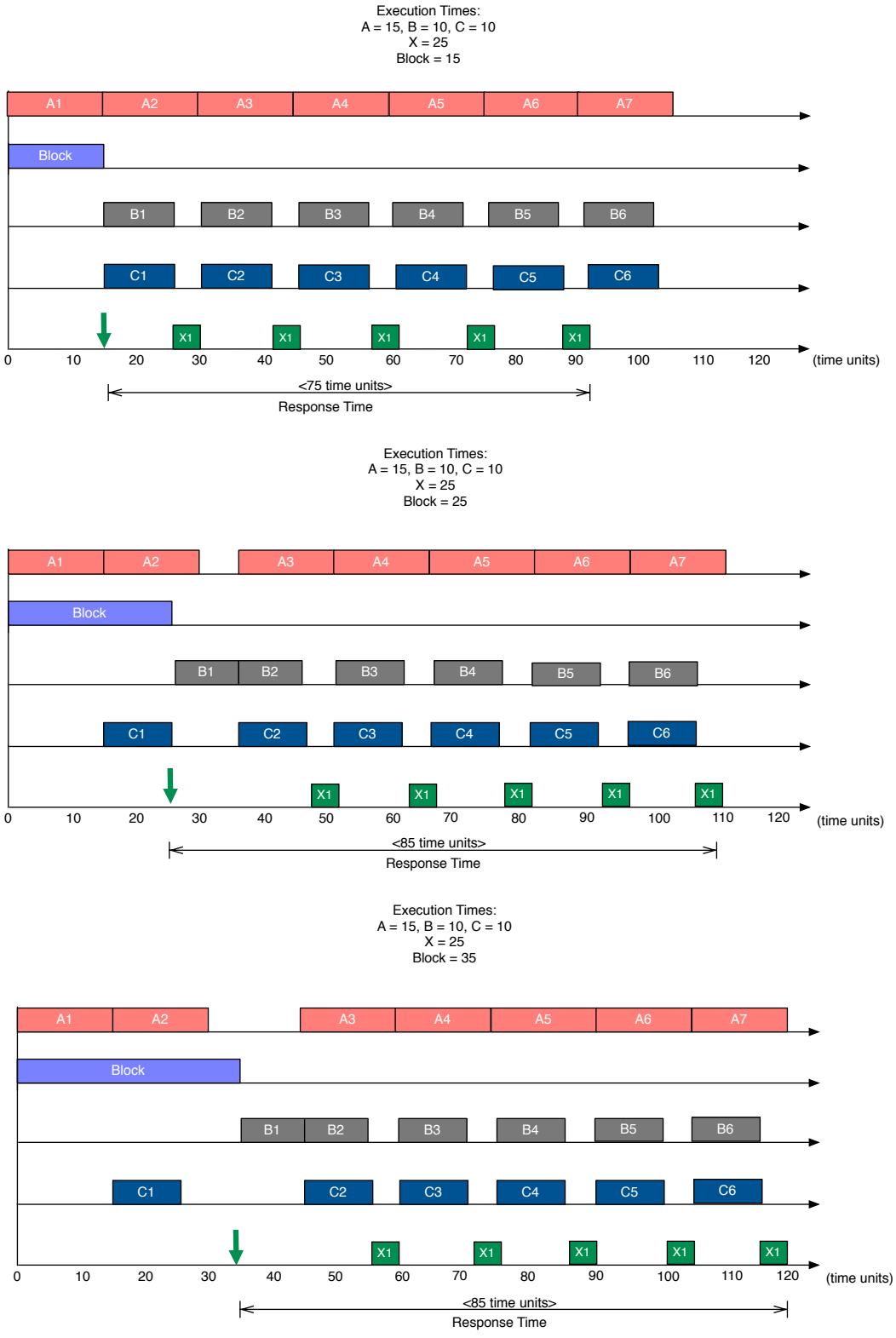

Figure 5.20: Timeline of the example SRDF with different block times (15, 25 and 35 time units)

a second delay of the *load actor* due to the Block actor. A rule of thumb [1] to achieve this is to assign:

$$d_{block} \geq \frac{t_{block}}{t_A} + 1 \tag{5.30}$$

For instances, *load actor* B in the example of Figure 5.18 (Execution times: A=15, B=10 and C=20) has three cyclical dependencies: 1) cycle A-B, 2) cycle B-C and 3) cycle A-B-C. With rates of 12.5, 15 and 11.25 respectively. Therefore, using equation 5.29 we compute:

$$t_{Block} = 15 \times 2 = 30 \tag{5.31}$$

This result is a conservative upper bound on the blocking time. However, a tighter bound can be found, as can be seen by the examples in Figure 5.18. Using a blocking time of 25 time units produces the same response time.

Optimization for applications with a *dominant* source

In a self-timed graph an actor will eventually fire as soon as its input tokens are available. By blocking the actor further than this point we are considerably adding pessimism to the results. Therefore the correct value for the *blocking time* should be exactly the largest distance, in time, between the *load actor* and any of its dependencies in the graph. In a self-timed graph this would correspond to the worst-case self-timed start time of the load actor. Unfortunately, even with current state-of-art analysis for SRDF, a bound on the worst-case self-timed start time is not always possible to determine because the graph might have a non-periodic behavior.

A **dominant source** in a SRDF graph, is an actor, or a cycle of actors, whose rate of execution imposes the rate of the graph. In other words, all actors are dependent on the *source actor* and the cycle containing such actor exhibits the maximum cycle mean (MCM) of the graph. In our application domain, most applications, such as radios, have a *dominant source*, that imposes the rate of the graph, and are periodic or sporadically periodic. If this is the case then it is possible to estimate the blocking time more accurately using the concept of *maximum latency*.

In such application the largest distance between the *load actor* and any of its dependencies, is always the distance between the *source actor* and the *load actor*. Using the latency concepts of section 2.4.4, we can estimate the maximum latency between the *source actor src* and the *load actor ld* for the following cases:

- For a periodic source:

$$t_{Block} = \hat{L}(src, ld, n) \leq \check{s}_{ROSPS}(ld, 0) - s(src, 0) + \mu(G) \cdot n, \tag{5.32}$$

where $\check{s}_{ROSPS}(ld, 0)$ represents the soonest start time of ld in an admissible ROSPS

- For a sporadic source:

$$t_{Block} = \hat{L}(src, ld, n) \leq \check{s}_{\eta} - s(src, 0) + \eta \cdot n, \tag{5.33}$$

  The latency $\hat{L}(src, ld, n)$ with a sporadic source has the same upper bound as the latency for the same source src, sink ld, and iteration distance n in the same graph with a periodic source with period η.

For example, in the case of Figure 5.18 we had previously established a bound for the *general blocking time* of 30 time units. However, analyzing the graph we see that at 15 time units, B as a token in the edge (A-B), and at 20 time units a token is produced at edge (C-B). At this points, actor B has all the conditions necessary to fire. Therefore, if we block *load actor* B more than 20 time units, we are creating a state in the graph that would never happen. Consequently, the results would be pessimistic.

5.4.3 Multiple load actors

In this subsection we address the case where a SRDF graph has multiple load actors. In section 5.2.5 we discussed this problem in terms of maximum load in a processor.

Our approach to extend the analysis done so far for a single actor, for multiple actors, is based on the definition of maximum load of section 5.2.4. In this situation we have several actors that can generated *load* in a processor. Therefore, a set of condition can be applied to a graph such that a particular *load actor* produces the maximum load. Furthermore, as we also stated in section 5.2.4, for a different low priority task different characterizations of the graph (one per *load actor*) can be responsible for the maximum load imposed on the processor that will lead to the worst-case response time of the task. Until the writing of this dissertation, we did not develop any analysis or heuristics to have an *a priori* knowledge of which *load actor* is responsible for the that specific maximum load. Therefore, our solution for multiple load actors is simple having a different characterization for the *blocking* of each *load actor* of the graph, and try each characterization to find the maximum worst-case response time of the task.

We explain with more detail how we explore the load imposed on the processor by the different *load actors* of an application, in section 5.5.3.

5.5 Implementation

Now that we have the necessary theoretical and practical concepts, we can implement the algorithms and functions to get results for our proposed SRDF worst-case response time and interference analysis. Furthermore, we are interested in comparing our analysis with the one proposed in [18]. Therefore, we will implement both analysis using the Heracles tool.

5.5.1 Our Algorithm Overview

Lets consider we have a set of n applications mapped on a MPSoC, with resources R, and that we want results on the interference and response time analysis of processor $P \in R$.

Notice that we are assuming that there are n schedule applications, and that each application may have m *load actors* of processor P. Since we don't know *a priori* which combination of maximum loads, of the *load actors* of m, will produce the maximum amount of total load for a particular low priority task Section 5.4.3 we need to explore all combinations of *blocked* load actors.

Therefore, we divided our approach in two major algorithms: a core analysis algorithm and a full-exploration algorithm. The core algorithm creates the load timelines, adapts the *load window* and performs interference and response time analysis. The full-exploration algorithm works at a higher level and simply runs the core algorithm recursively until all combinations of *blocked load actors* are explored. The full algorithm is described in pseudo-code in Algorithm 4. Because of this extensive exploration of all possible combinations the algorithm has exponential complexity. However, for the range of applications we study this complexity level is acceptable.

input : List of SRDF graphs, Target processor P
output: Response Times, Schedulability
final-results = new Hashtable();
for *each combination c of load actors* **do**
    fp-graph ← Connect block actors of (*c*,fp-graph);
    timelines ← simulate(fp-graph);
    mod-timelines ← adjust load window (timelines);
    results ← generate interference analysis (mod-timelines);
    final-results ← compare results (final-results,results);
end
print final results;
check schedulability;

Algorithm 4: Top-level algorithm

5.5.2 Core algorithm

In this subsection, we will describe the core algorithm of our implementation. In terms of the description in Algorithm 4, the core algorithm corresponds to the operations done within the main **For** cycle, for each combination of *load actors*.

Preparing the initial conditions

The first step is to prepared the SRDF model to have the necessary condition in order to obtain the maximum load from a *load actor*. Therefore, all the applications we model as SRDF graphs will have an assigned priority and a unique *Block* actor. When the selected applications are introduced into the Heracles tool, they are parsed into a SRDF graph structure and the block actor is connected to the first *load actor* of the processor we want to perform the interference analysis. Before we can obtain the load timelines for each *load actor* we must first determine what is the type of SRDF applications, in order to determine the correct *blocking time*. If the application is non-periodic we simply apply the set of equation of section 5.4.2, for the general case. Otherwise, we must first determine what is the Period (μ) and the Static Periodic Schedule (SPS) of that application. We hold these values in an

hashtable structure, with priority as a key.

Determining the maximum load

At this point, we call the Full Exploration algorithm to connect all the *Block* actors to their respective *load actor* candidates. The blocked applications are then ran in the Heracles simulator. After simulating, we collect all the simulated timelines, of each application, as a list of simulation events and pass them onto the next phase: setting the *load window*.

Setting the *load window*

Now that we have collected all the timelines we need to adjust them for our *load window* to reflect the *load* generated by the selected combination of blocked *load* actors. For such to happen, for each timeline we remove all the events that occur before the set *blocking time* for that application. In the end we obtain a list of timelines, whose events are all synchronized at the start of the load window (δ_0), and represent the maximum load imposed on the processor.

Interference analysis

This part of the implementation uses the fixed priority model of Heracles, created for the work in [1]. The fixed priority model contains a function for merging two different timelines, *merge_timelines*, and a function to find the response time of a specific task, *slot_fill_task*. Using these functions, we create a recursive algorithm that merges the received timelines, except the lowest priority (LP) application's timeline, into a single *maximum load* timeline. The last step is to get the response time for each task of the LP application by slot-filling it in the *merged timeline*.

The pseudo-code for the interference analysis algorithm is described in Algorithm 5, and a example of the *slot filling* procedure is depicted in Figure 5.21.

list = all timeline of *prio* < number of apps;
LP = all load actors of the application with *prio* = number of apps;
foreach *timeline (tl) in list* **do**
  merged_timeline = merged_timelines (merged_timeline) (*tl*);
end
foreach *load actor la in LP* **do**
  response_time = **slot_fill_task** *la* merged_timeline;
  add response_time to results_list;
end
return results_list;

Algorithm 5: Interference Analysis Algorithm

5.5.3 Full Exploration Algorithm

The **For** cycle in the top-level algorithm 4, is in practice a recursive function that receives an hashtable with the *load actors* per application and a the list of *load actors* of the highest priority application. Then it will iterate over the list of *load actor* candidates and connect,

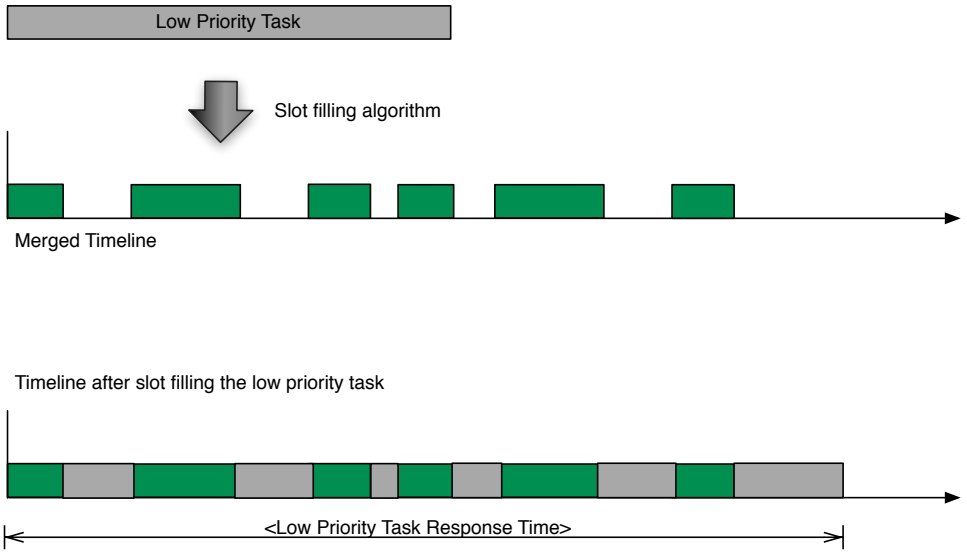

Figure 5.21: Slot filling algorithm example

in each iteration, a *Block* actor to each selected *load actor*. Every time the *Block* actor of an application is connected, the function is recalled with the list of candidates of the next priority application. This recursion is repeated until the n_{th} application is reached, the lowest priority application. At this point, the algorithm stops the recursion and calls the simulator for the current combination of blocked *load actors*. This recursive iteration over the load actors lists permits us to analyze all possible combinations. The pseudo-code for this particular is described in Algorithm 6.

```
let rec connect_block_actors list prio graph = begin
    if prio >= number of applications then
        call core algorithm graph
    end
    foreach load actor l in list do
        app_graph ← graph(prio);
        block_actor ← app_graph.block_actor;
        block_actor.blocking_time ← l.sps_start_time;
        app_graph ← add edge (block_actor,l);
        graph(prio) ← app_graph;
        connect_block_actors (graph(prio+1).load_actors) (prio+1) graph;
    end
end
```

Algorithm 6: Full exploration algorithm

5.5.4 Final Results and Tests

For every run of the core algorithm a list of response times is returned. Each value of the results is compared with the previous, and if they reflect a worst-case response time then they replace the previous value. Otherwise they are discarded. When all the combination of

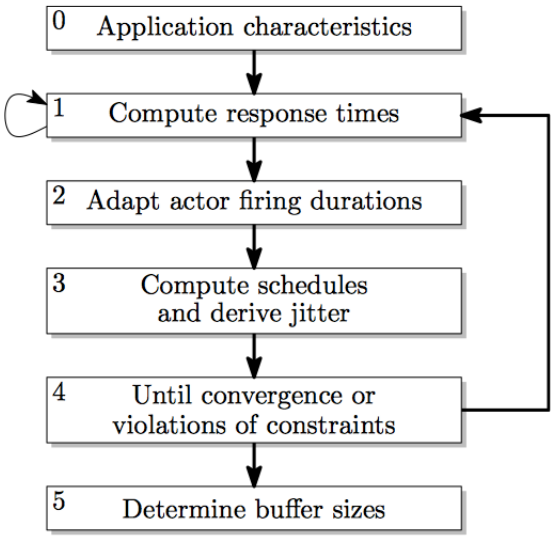

Figure 5.22: Analysis Flow of [18]

blocked load actors have been tested a list with the final results is printed and the temporal analysis using for the response model graph is called. It consists in replacing each *load actor* of the LP application by a single non-concurrent actor with a constant execution time equal to the obtained bound on the worst-case response time for that actor. Then temporal analysis tests are run on the modified LP application graph to assert if any constraint was violated.

5.5.5 Hausmans's Approach

Lets first introduce the approach in [18]. The presented temporal analysis technique is a dataflow based approach that uses a enabling-jitter characterization and iterative fixed-point computation. However, the application domain is restricted to periodic dominant source applications. The analysis flow is quite similar to the one in [20], where the enabling jitter of tasks combined with their period is used to compute the response time of task. However, the computation of the enabling jitter, in contrast with other methods, is done by using the best-case schedule and the worst-case schedule, for a dataflow graph. Convergence of the response times is guaranteed because the response times are monotonic in the enabling jitters of the task and because of the temporal monotonicity of dataflow graphs [40]. The analysis finishes when the enabling jitters converge or when a violation of the temporal constraints is detected.

Analysis Flow

The analysis flow starts by characterizing each application's tasks in terms of priority, period (P) and best and worst-case execution times (B and C respectively). Then a recursive algorithm will be ran for each application until the returned response times converge. After one application is analyzed, its results are passed onto the next lower priority application. The analysis flow ends when all applications have been analyzed. The analysis flow is depicted in Figure 5.22.

Response Times

The response times are calculated for each task by using an event model with enabling jitter (J), as in [3] [2]. The model used to determine the worst-case response times is formalized in the following equations:

$$w_i(q) = q \cdot C_i + \sum_{j \in hp(i)} \hat{\eta_j}(w_i(q)) \cdot C_j \tag{5.34}$$

$$\hat{\eta_j}(\Delta t) = \lceil \frac{J_j + \Delta t}{P_j} \rceil \tag{5.35}$$

$$\hat{R_i} = max(w_i(q) - (q-1) \cdot P_i), 1 \leq q \tag{5.36}$$

Equation 5.34 calculates $w_i(q)$ which is the maximum amount of time it takes to finish q executions of task i. Function $hp(i)$ returns the set of tasking running on the same processor as i, with a higher priority. The model for best-case response times is the same, except it uses the best-case execution time.

Compute Schedules and Derive Jitter

The next step in the analysis flow is to build an analysis model with the determined response times. The model chosen is a SRDF graph. A SRDF graph is then built for each application. One that consider best-case response times (BC-SRDF) and one that consider worst-case response times (WC-SRDF). Each dataflow graph model is then scheduled, using a Self-Timed schedule in the case of BC-SRDF and a Static Periodic Schedule in the case of the WC-SRDF.

The final step is to use the scheduled dataflow graph models to retrieve the jitter of each of the application's task.

$$J_i = \hat{s_i} - \check{s_i} \tag{5.37}$$

Convergence of the flow

The analysis flow ends when all applications have been analyzed, and when for each application there was a convergence in the determination of the enabling jitter of each task. The worst-case response time are then returned and, unless a violation is detected during the analysis, the set of applications is said to meet its temporal requirements.

5.6 Results

In this section we will perform different tests on our implementation to evaluate the performance and validity of our response time and interference analysis. Furthermore, we will compare our approach with the one in [18].

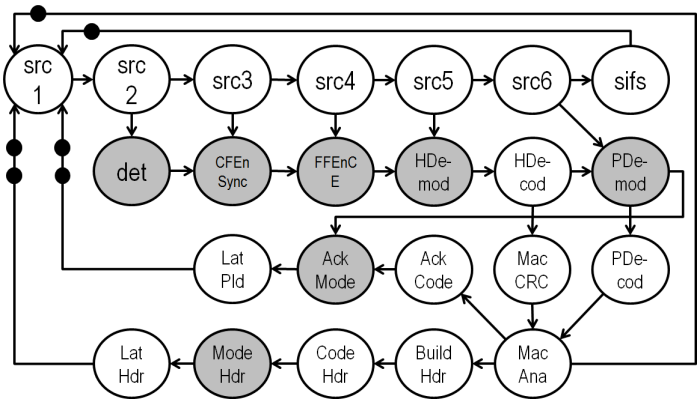

Figure 5.23: SRDF Model of a WLAN Radio

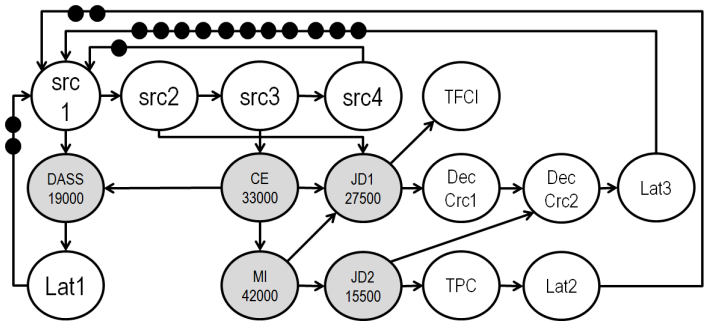

Figure 5.24: SRDF Model of a TDSCDMA Radio

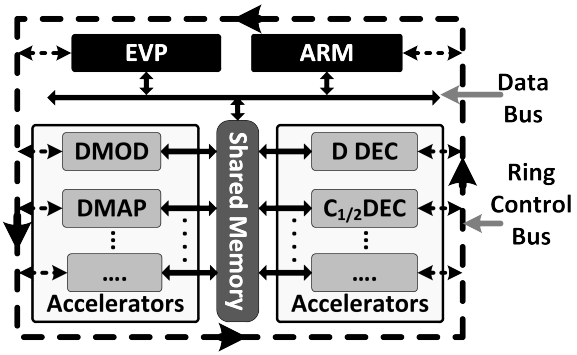

Figure 5.25: Abstract target architecture template

			Worst-Case Response Times		
		Execution Times	GA	OA	HA
LP Actors	CE	33000	45070	39035	39035
	MI	42000	55590	51395	54070
	DASS	19000	28630	24815	25035
	JD1	27500	39570	33535	33535
	JD2	15500	24210	20960	21535

Figure 5.26: Results for the analysis of experiment WLAN+TDSCDMA

5.6.1 Case Studies: WLAN and TDSCDMA models

For our experiments, we will use two SRDF models from real life applications: a **Wireless Local Area Network 802.11a** (Figure 5.23) and a **Time Division Synchronous Code Division Multiple Access (TDSCDMA)** (Figure 5.24). The system architecture for the MPSoC platform were the application are mapped is depicted in Figure 5.25. It includes one or more general-purpose **ARM** cores, to handle control and generic functionality, one or more of the **EVP** [6] core, to handle detection, synchronization and demodulation, and one or more **Software Codec** processors, that take care of the baseband coding and decoding functions.

Each actor in the graph contains the information on the task it represents and the execution time, in CPU cycles. As we will be focusing on studying only the task mapped on the **EVP** processor, all *load actors* of the **EVP** are shaded in the application graphs. The *EVP* is a vector processor designed and create by ST-Ericsson, with a clock speed of 300 MHz.

Prior to beginning of the experiments, each application graph is given a priority and a unique unconnected *Block* actor. For each experiment we will gather results from our proposed analysis and the analysis proposed in [18]. We will then discuss and derive conclusion from the results.

5.6.2 Interference Analysis (2 Applications)

For this case, we will run analyze the following experiments:

- **WLAN+TDCDMA**: A WLAN application with high priority and a TDSCDMA application with low priority;
- **WLAN+WLAN**: A Fast WLAN application with high priority and a Slow WLAN application with low priority;

WLAN - TDSCDMA

For this experiment we use a WLAN application, with a periodic source of 40000 CPU cycles, and a TDSCDMA application, with a periodic source of 675000 CPU cycles. We give high priority (HP) to the WLAN application and low priority (LP) to the TDSCDMA application. We run the same experiment for our proposed analysis, with the *blocking time* determined by the general case equation (GA), by our optimized approach (OA), and for the approach proposed by Hausmans (HA).

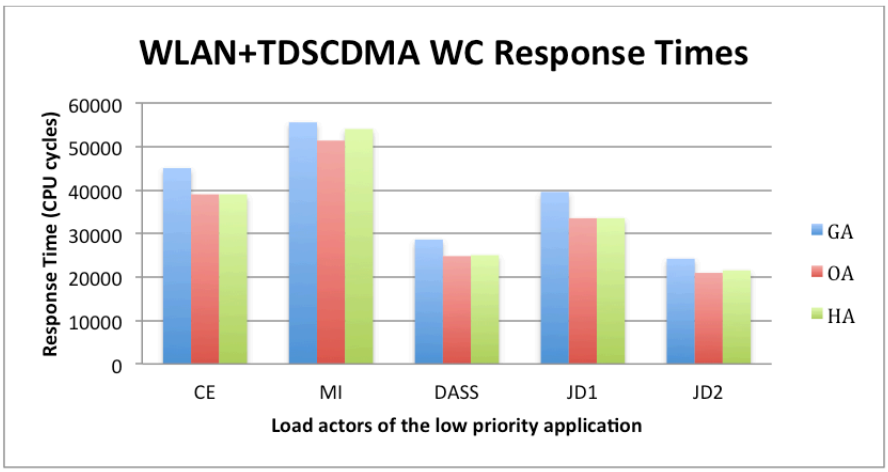

Figure 5.27: Results for the analysis of experiment WLAN+TDSCDMA

Looking at the results in Figure 5.26 and Figure 5.27, we can derive several conclusions. The most pessimistic results are delivered by our analysis using the general case rule for determining the *blocking time* (GA). This is expected since both applications have a periodic behavior and therefore the value for the *blocking time* can be tighter. This is reflected by the results returned by both periodic regime constrained analysis (OA and HA) that have tighter worst-case responses.
Between these two analysis we see that in the particular cases of LP actors MI, DASS and JD2, our optimized analysis (OA) returns tighter response times than Hausmans approach (HA). In all the other cases the results are exactly the same.

WLAN - WLAN

For this experiment we use a fast WLAN application, with a periodic source of 40000 CPU cycles, and a slow WLAN application, with a periodic source of 349600 CPU cycles. We give high priority (HP) to the fast WLAN application and low priority (LP) to the slow WLAN application. We run the same experiment for our proposed analysis, with the *blocking time* determined by the general case equation (GA) and by the periodic case equation (OA), and for the approach proposed by Hausmans (HA).

Looking at the results in Figure 5.28 and Figure 5.29, we see that in this case the most pessimistic results are the ones returned by Hausmans approach (HA). As expected, we still have tighter results returned by our optimized analysis (OA) than our general analysis (GA).

5.6.3 Interference Analysis (3 Applications)

For this case, we will run analyze the following experiments:

- **WLAN+WLAN+TDSCDMA**: A WLAN application with high priority, a WLAN application with medium priority and a TDSCDMA application with low priority;
- **WLAN+TDSCDMA+TDSCDMA**: A WLAN application with high priority, a TDSCDMA application with medium priority and a TDSCDMA application with low priority;

			Worst-Case Response Times		
		Execution Times	GA	OA	HA
LP Actors	Detect	220	3815	1140	6255
	CFenSync	355	3950	1875	6390
	FFenCE	680	4275	2200	6715
	Hdemode	920	4515	2440	6955
	Pdemode	920	4515	2440	6955
	ModHeader	920	4515	2440	6955
	AckMode	600	4195	2120	6635

Figure 5.28: Results for the analysis of experiment WLAN+WLAN

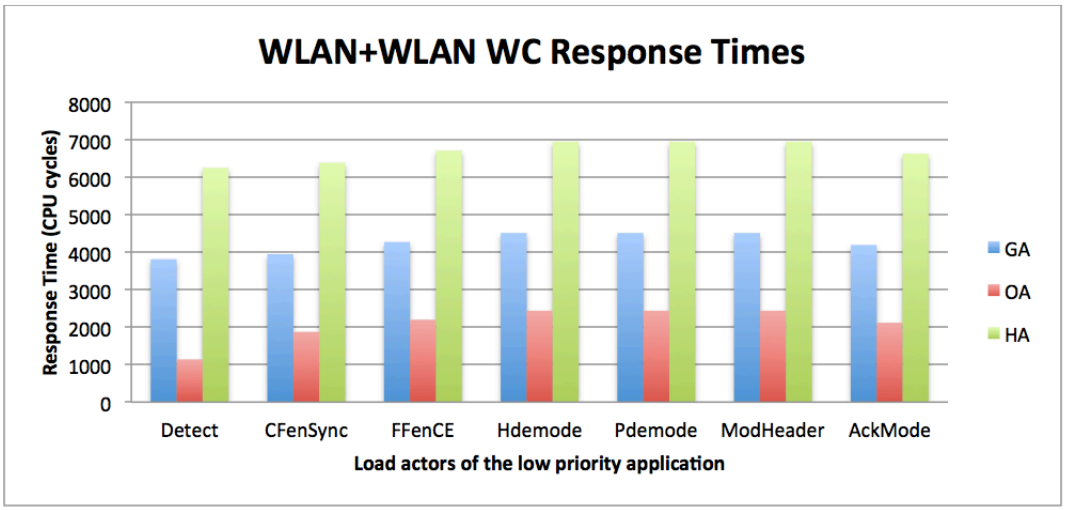

Figure 5.29: Results for the analysis of experiment WLAN+WLAN

			Worst-Case Response Times	
		Execution Times	OA	HA
LP Actors	CE	33000	42995	51105
	MI	42000	55790	60105
	DASS	19000	27475	31070
	JD1	27500	35975	39570
	JD2	15500	23755	27570

Figure 5.30: Results for the analysis of experiment WLAN+WLAN+TDSCDMA

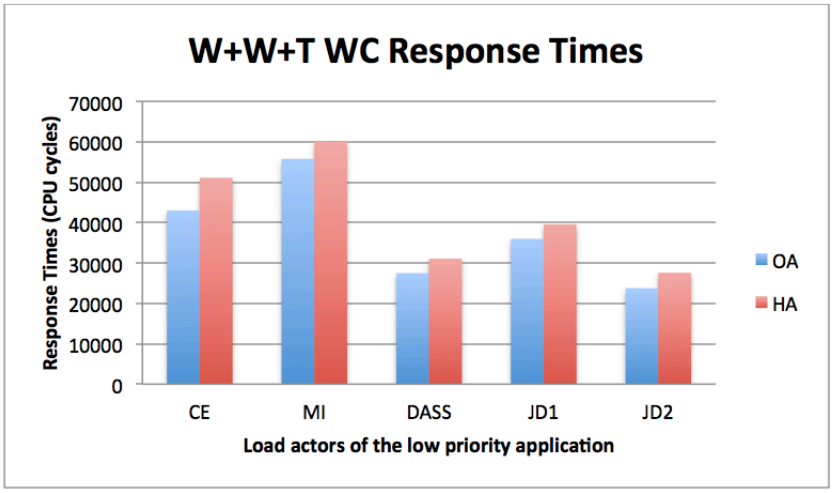

Figure 5.31: Results for the analysis of experiment WLAN+WLAN+TDSCDMA

WLAN - WLAN - TDSCDMA

For this experiment we use a fast WLAN application, with a periodic source of 40000 CPU cycles, a slow WLAN application, with a periodic source of 349600 CPU cycles, and a TDSCDMA application, with a periodic source of 675000 CPU cycles. We assign high priority (HP) to the fast WLAN application, medium priority (MP) to the slow WLAN application and low priority (LP) to the TDSCDMA application. We run the same experiment for our proposed analysis, with the *blocking time* determined by the periodic case equation (OA), and for the approach proposed by Hausmans (HA).

In this more complex case, we see that our optimized analysis returns tighter results than Hausmans approach for all LP actors of the TDSCDMA application.

WLAN - TDSCDMA - TDSCDMA

For this experiment we use a fast WLAN application, with a periodic source of 40000 CPU cycles, a first TDSCDMA application, with a periodic source of 675000 CPU cycles, and a TDSCDMA application, with a periodic source of 675000 CPU cycles. We assign high priority (HP) to the fast WLAN application, medium priority (MP) to the slow WLAN application and low priority (LP) to the TDSCDMA application. We run the same experiment for our proposed analysis (OA), with the *blocking time* determined by the periodic case equation,

		Execution Times	Worst-Case Response Times	
			OA	HA
LP Actors	CE	33000	178640	206210
	MI	42000	187640	215210
	DASS	19000	164640	186175
	JD1	27500	173140	194675
	JD2	15500	161140	182675

Figure 5.32: Results for the analysis of experiment WLAN+TDSCDMA+TDSCDMA

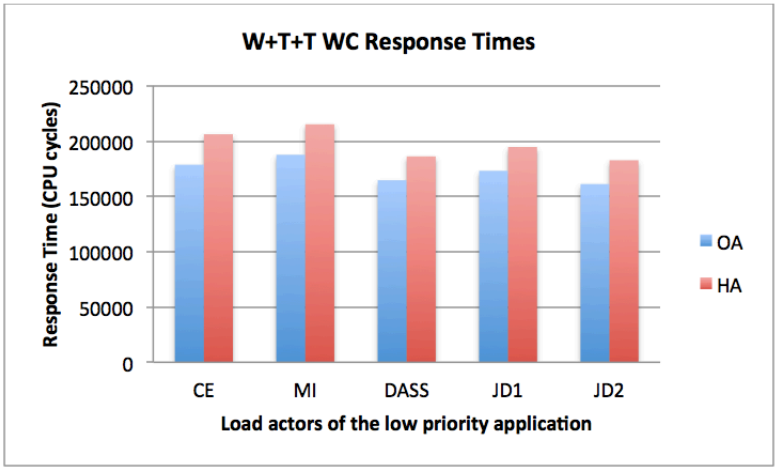

Figure 5.33: Results for the analysis of experiment WLAN+TDSCDMA+TDSCDMA

and for the approach proposed by Hausmans (HA).

Again, the results in Figure 5.32 and 5.33 show that our optimized analysis (OA) has tighter response times, than Hausmans approach (HA), for all the LP application's actor. However, in terms of the schedulability (Figure 5.34) of these three applications both methods, OA and HA, concluded that the throughput constrains of the LP application are not met.

5.6.4 Conclusions

In all experiments done the results returned by our optimized analysis (OA) are always tighter than our general analysis (GA) and Hausmans approach (HA). Regarding our general case analysis (GA), this was expected due to the fact that the analysis is pessimistic

	Expected MCM	Outputted MCM	
		OA	HA
W+W+T	675000	675000	675000
W+T+T	675000	687280	736385

Figure 5.34: Results for the schedulability of experiments

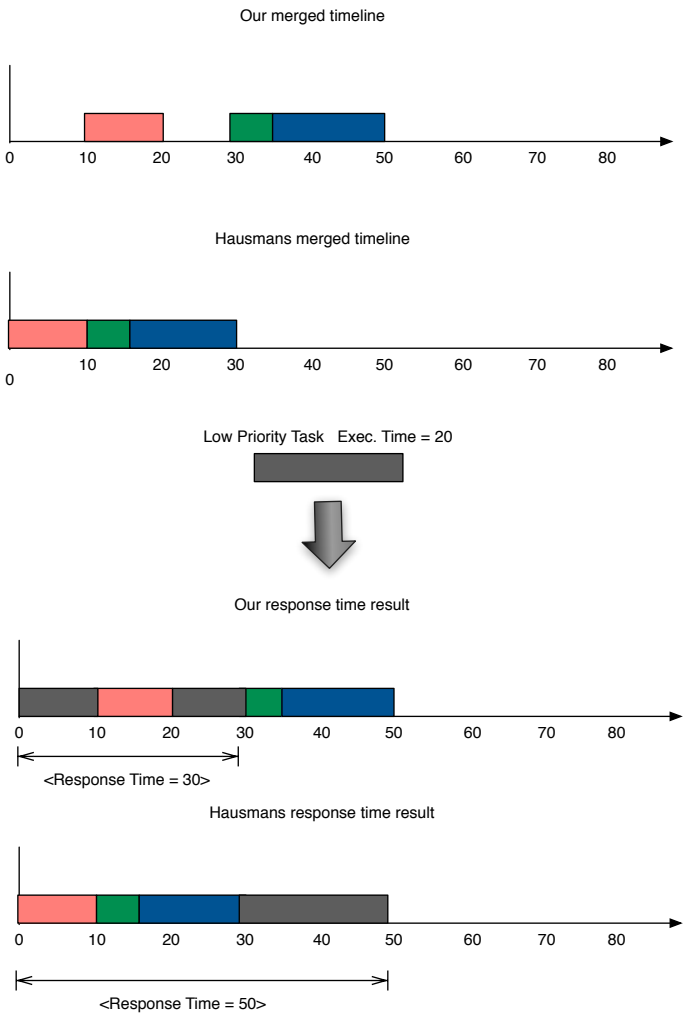

Figure 5.35: Response time analysis for both techniques

in determining the *blocking times* for the *load actors*. Leading to a state in the graph that would never be possible, under the graphs assumptions, and therefore, giving also pessimistic response times results.

On the other hand, Hausmans approach (HA) has as an application domain of strictly periodic applications, and therefore, the returned response times are tighter, in most cases, than our general case analysis (GA). However, since Hausmans approach does not consider offsets between tasks due to data-dependencies, for the applications we tested the results are never better than our optimized analysis (OA). This is strictly due to our analysis using a dataflow based approach in determining the load function that actually considers a simulation of the application to detect the (a)cyclical data-dependency offsets, which are already naturally described in a dataflow graph. This is depicted in Figure 5.35 where we show how each approach's final timelines format is, and how the response time are then determined.

All in all, we have proposed an approach for the interference analysis between fixed priority

streaming applications and response time analysis. The approach we suggest is for the general case of any SRDF graph. The same is not valid for the approach suggested by Hausmans, which focus only in periodic applications. Furthermore, our optimized analysis for periodic and sporadically periodic applications has proven to give the tighter response times.

5.7 Software Implementation

As a result of the implementation of the analysis presented in this chapter, we had to add and modify certain aspects of the Heracles simulator and analysis tool. Specifically we had to introduce the following changes:

- **Change the inputs of the tool -** Prior to the work done in this chapter, a set of applications were added in a single file. Graphs were manually merged. We added a simple option to import all files from a folder and automatically add the correspondent values and changes needed to be used as a set of fixed priority SRDF applications.
- **Adapt and improve the Fixed Priority module -** As result of the work done in [1], a fixed priority module was added to Heracles. However, most of its purpose was for a fixed 2-application analysis. Part of the implementation was to adapt and improve on this module to be able to perform interference and response time analysis for a set of n fixed priority SRDF applications, according to the techniques described in 5.2.3, 5.2.4, 5.3 and 5.4.
- **Implementation of the interference analysis -** As result of the work done in this dissertation, we designed, tested and implemented all the algorithms described in Section 5.5. As well as the algorithms described in [18] in order to be able to compare results.

5.8 Summary

In this chapter we presented a solution for the temporal analysis of fixed priority scheduled applications on a MPSoC, using a dataflow based approach. We proposed a characterization of the load imposed by actors, by exploiting the temporal properties of the self-timed execution of SRDF, and used this load characterization to derive worst-case response times for lower priority tasks. We first illustrated how the worst-case response time bound on a low priority task is obtained assuming a maximum load imposed by tasks with higher priority. We then demonstrated how we determine the conditions for the self-timed execution of a dataflow graph to impose the maximum load on a given processor. Furthermore, we proposed the concept of *blocking* to achieve the maximum load imposed, and showed how to reduce the optimism when determining the correct blocking time for applications with dominant periodic sources. We validated our observations with experiments of simultaneous execution of multiple radio applications on a MPSoC. Furthermore, our analysis presents tighter results than the current state-of-art approach in terms of worst-case response time computation, but for a cost of exponential complexity of the analysis flow. However, for the practical examples studied, this complexity was acceptable.

Chapter 6

Conclusions

MultiProcessor Systems-on-chip (MPSoCs) are an often sought solution for platforms to run several real-time streaming applications simultaneously. MPSoCs are versatile and powerful platforms designed to fit the needs of embedded applications. Embedded streaming applications mapped on a MPSoC, are often modeled using dataflow Models of Computation (MoC). Dataflow graphs have the expressivity and analytical properties to naturally describe concurrent digital signal processing applications [8]. Many scheduling techniques have been analyzed for dataflow models of applications, such as Time Division Multiplexing (TDM) and Non-preemptive Non-blocking Round Robin (NPNBRR) [24,28]. However, few attempts have been made to characterize the temporal behavior of dataflow modeled applications under a Fixed Priority (FP) scheduling scheme. Fixed priority scheduling is very popular, mostly because it is very easy to implement and predictable in overload conditions [10]. When a processor with a fixed priority schedule has a peak of load, the only affected tasks are the ones with the lowest priorities. Therefore, systems that have one or more critical applications find fixed priority scheduling a very attractive solution.

This dissertation set off with the objective of building the necessary concepts and techniques to analyze fixed priority scheduling for applications modeled with a state-of-art dynamic dataflow MoC, Mode-Controlled Dataflow. However, since fixed priority analysis of streaming applications, whose tasks do not exhibit periodic activations, is not trivial, and moreover, there were still issues with the analysis for static dataflow models, from the work in [1], our main objective could not be achieved within the time span of this dissertation. Instead, we opted to improve the existing fixed priority analysis for static dataflow models and temporal analysis of MCDF graph models, in order to establish the necessary ground work to extend the analysis for dynamic dataflow MoC in the future.

We presented an implementation solution for a temporal analysis technique of MCDF graphs, using a finite state machine (FSM) to describe the dynamic behavior of a MCDF graph, FSM-MCDF analysis. The proposed solution is able to consider intermodal dependencies, pipelining execution and provide a complete analysis of the temporal behavior of a graph. When compared with the current available analysis techniques, our solution proved to provide equal or better results for throughput and latency analysis. Furthermore, FSM-MCDF provides optimal results for the temporal analysis of an MCDF graph, but only if the description of the FSM matches exactly the natural behavior of the real application.

We propose an improved fixed priority analysis for SRDF graphs, based on the initial work done in [1]. Specifically, we demonstrated how to obtain the worst-case response time bound of a low priority task assuming maximum load imposed by higher priority tasks. We showed how to characterize a self-timed execution of a dataflow graph such that it imposes the maximum load on a processor. Furthermore, we proposed an optimization for the case of applications with a single dominant source on a fixed priority assignment. Finally, we validated and compared our analysis with the ones in [1] and [18]. We conclude that in all cases, our analysis provides the tightest results for the worst-case response time analysis of a low priority task. Moreover, the analysis technique we proposed is for a general case of any SRDF graph, while the analysis in [18] has a narrowed application domain of strictly periodic applications.

However, we were not able to devise a methodology, or heuristics, to determine the combination of characterization of a set of SRDF graph that would impose the maximum load on a processor. Therefore we base our analysis on a full exploration of all possible characterizations of the maximum load of *load actors* of the set of scheduled graphs, which leads to our proposed analysis to have exponential complexity. Nonetheless, our results for the practical cases for which we tested this technique, the execution time of the analysis tools was perfectly acceptable.

6.1 Future Work

As suggestions for future work we leave the following topics:

- **Automate the generation of the transition graph (Finite State Machine)**: One of the problems discovered when implementing the FSM-MCDF analysis was that the analysis was only optimal if the FSM description of the MCDF graph portrayed the correct behavior of the real application. We currently build our FSM manually and, although for simple graphs with few modes of operation this is sufficient, for more complex applications it is quite prone to errors. Furthermore, an automated generation of the FSM could solve our current issues with limiting the number of consecutive transitions of cyclical mode sub-sequences (ex. 1-2-1-2-1-2-...) that would never occur in the real application.

- **Adapt the FSM-MCDF analysis for non-strongly connected graphs**: The current implementation of FSM-MCDF only supports strongly connected graphs. Despite this being the case of most of the graphs we study, it is still a restriction in our application domain. As already suggested, a possible solution would be to convert non-strongly connected graphs to a a graph of its strongly connected components, as in [32].

- **Improve the performance of the fixed priority exploration algorithm**: Some adjustments can be made to the current implemented version of our fixed priority analysis technique to, at least, reduce the number of explored cases. For instances, finding a set of heuristic parameters to narrow down the number of combinations to explore. Or a conservative simplification of the analysis that provides slightly more pessimistic results, but is of less algorithmic complexity.

- **Fixed Priority for MCDF analysis**: The goal of this dissertation was to give the first steps in establishing an analysis strategy for the worst-case response time analysis of MCDF graphs with a fixed priority assignment. MCDF allows for a more realistic modeling of applications, since it captures the natural dynamic behavior of modern streaming applications, consequently reducing the gap between model and real application. Therefore, we expect that, with MCDF graphs, our worst-case response time analysis would give results closer to the actual values of the real application. Unfortunately we could not reach this objective within the time span of this dissertation. Ideally, if we can determine the *mode sequence* of a MCDF graph that imposes the maximum load on a processor, we can build an equivalent SRDF graph of that particular sequence and simply use our proposed fixed priority analysis for SRDF graphs. However, the problem lies in determining which is that particular *mode sequence*, since it is not clear what are the main factors that influence a *mode sequence* to impose the most load on a processor.

Bibliography

[1] Ricardo Almeida. Real-time fixed priority scheduling for multiprocessors (escalonadores de prioridade fixa em multiprocessadores de tempo-real). Master's thesis, University of Aveiro, 2012.

[2] N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[3] NeilC. Audsley, Alan Burns, RobertI. Davis, KenW. Tindell, and AndyJ. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[4] Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011. ACM.

[5] M. Bekooij et al. Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and Robust Streaming in and between Connected Consumer Electronic Devices*, volume 3, pages 81–108. Springer, 2005.

[6] K. Berkel et al. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, 2005(16), 2005.

[7] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *Computers, IEEE Transactions on*, 52(7):933–942, 2003.

[8] J.T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, Univ. of California, Berkeley, September 1993.

[9] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

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

[11] T.H Corman et al. *Introduction to Algorithms*. McGraw-Hill, 2001.

[12] Christof Ebert. Embedded software: Facts, figures, and future. IEEE Computer Society, April 2009.

[13] Pascal Manoury Emmanuel Chailloux and Bruno Pagano. *Developing Applications with Objective Caml*. O'Reilly, 2000.

[14] M. Geilen. Dataflow scenarios. In *IEEE Transactions on Computers*, 2010.

[15] Marc Geilen. Synchronous dataflow scenarios. *ACM Trans. Embed. Comput. Syst.*, 10(2):16:1–16:31, January 2011.

[16] Global Industry Analysts, Inc. . http://www.prweb.com/, 2013.

[17] Jon D. Harrop. *OCaml for Scientists*. May 2007.

[18] Joost P.H.M. Hausmans, Stefan J. Geuns, Maarten H. Wiggers, and Marco J.G. Bekooij. Dataflow analysis for multiprocessor systems with non-starvation-free schedulers. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, pages 13–22, New York, June 2013. ACM.

[19] Bernd Heidergott, Geert Jan Olsder, and Jacob W. van der Woude. *Max Plus at work : modeling and analysis of synchronized systems : a course on Max-Plus algebra and its applications*. Princeton series in applied mathematics. Princeton University Press, Princeton (N.J.), 2006.

[20] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):148–166, 2005.

[21] Jason Hickey. *Introduction to Objective Caml*. Cambridge University Press, 2008.

[22] International Business Times . http://www.ibtimes.com/worldwide-smartphone-users-cross-1-billion-mark-report-847769, 2012.

[23] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, 1987.

[24] Alok Lele, Orlando Moreira, and Pieter J.L. Cuijpers. A new data flow analysis model for tdm. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 237–246, New York, NY, USA, 2012. ACM.

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

[26] O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

[27] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proc. Embedded Software Conference (EMSOFT)*, October 2007.

[28] Orlando Moreira. *Temporal Analysis and Scheduling of Hard Real-Time Radios on a Multiprocessor*. PhD thesis, Eindhoven University of Technology, 2012.

[29] Moonju Park. Non-preemptive fixed priority scheduling of hard real-time periodic tasks. In Yong Shi, GeertDick Albada, Jack Dongarra, and PeterM.A. Sloot, editors, *Computational Science – ICCS 2007*, volume 4490 of *Lecture Notes in Computer Science*, pages 881–888. Springer Berlin Heidelberg, 2007.

[30] T.M. Parks and E.A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3235–3238 vol.5, 1995.

[31] R. Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, October 1968.

[32] Firew Siyoum, Marc Geilen, Orlando Moreira, and Henk Corporaal. Worst-case throughput analysis of real-time dynamic streaming applications. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '12, pages 463–472, New York, NY, USA, 2012. ACM.

[33] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker Inc., 2000.

[34] Marcel Steine, Marco Bekooij, and Maarten Wiggers. A priority-based budget scheduler with conservative dataflow model. In *DSD*, pages 37–44, 2009.

[35] The Statistics Portal. http://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/, 2013.

[36] Bart Theelen. Scenario-aware dataflow. Technical report, Technical University of Eindhoven, 2008.

[37] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104 vol.4, 2000.

[38] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[39] K. W. Tindell. Extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real-Time Systems*, 6, 1992.

[40] Maarten H. Wiggers, Marco J.G. Bekooij, and Gerard J.M. Smit. Monotonicity and run-time scheduling. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 177–186, New York, NY, USA, 2009. ACM.