Universidade de Aveiro
2012
Departamento de Eletrónica, Telecomunicações e Informática

Ricardo Daniel Lopes Almeida

Escalonadores de Prioridade Fixa em Multiprocessadores de tempo-real

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e de Telecomunicações, realizada sob a orientação científica pelo Dr. Paulo Bacelar Reis Pedreiras, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e co-orientação científica por Dr. Orlando Miguel Pires dos Reis Moreira, Principal DSP Systems Engineer na empresa ST-Ericsson.

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

o júri

Presidente

Professor Doutor José Alberto Gouveia Fonseca
Professor auxiliar do Departamento de Eletrónica e Telecomunicações da Universidade de Aveiro

Professor Doutor Paulo Bacelar Reis Pedreiras
Professor auxiliar do Departamento de Eletrónica e Telecomunicações da Universidade de Aveiro

Doutor Orlando Miguel Pires dos Reis Moreira
Principal DSP Systems Engineer em ST-Ericsson

Professor Doutor Luís Miguel Pinho de Almeida
Professor associado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Palavras-chave

Escalonadores de prioridade fixa, data-flow, tempo-real, multiprocessadores, carga computacional, sistemas embebidos, processamento digital de sinal.

Resumo

Devido evolução tecnológica observada nos últimos anos, os sistemas embutidos com capacidade de multi processamento tornaram-se comuns. Nestes dispositivos, a escassez de recursos obriga a uma distribuição otimizada dos mesmos pelas diversas atividades suportadas.

Este tipo de dispositivos contam normalmente com um processador de uso geral, tipicamente um processador da família ARM, e um ou mais processadores direcionados a tarefas específicas, como processadores vetoriais (EVP), utilizados em sistemas de processamento digital de sinal por exemplo.

A distribuição de recursos pelas tarefas do sistema é feita por um escalonador. Este pode fazer a distribuição de recursos obedecendo a uma das várias disciplinas conhecidas: Round Robin, First In First Out, Time Division Multiplexing, Fixed Priority, etc.

O presente trabalho tem como principal objetivo a investigação de escalonadores de tempo-real baseados em prioridades fixas, com especial atenção para a aplicações de streaming a executar em plataformas multiprocessador, utilizando dataflow.

Dataflow é um paradigma que utiliza teoria de grafos para realizar a modelação, programação e análise de aplicações e sistemas.

A primeira parte deste projeto é dedicada à análise e modelação de grafos de fluxo de dados onde a distribuição de recursos é feita com recurso a um escalonador de prioridade fixa. A segunda parte será dedicada ao estudo da interferência entre tarefas com níveis de prioridades distintos em grafos independentes, quando mapeados para execução no mesmo processador. Em sistemas embebidos, existem tarefas de alta prioridade (periódicas ou esporádicas) que têm de ser atendidas o mais rapidamente possível quando prontas a executar. Este atendimento irá interferir na execução de tarefas que corram na mesma plataforma com níveis de prioridade inferiores, pois estas serão bloqueadas durante a execução das tarefas de maior prioridade. Esta interferência tem como consequências diretas a diminuição do tempo de resposta das tarefas de alta prioridade e o aumento do tempo de execução das tarefas com níveis de prioridades baixos.

Com este trabalho pretendemos verificar quais as vantagens e desvantagens que um escalonador de prioridade fixa pode oferecer neste tipo de situações, quando comparado com outros escalonadores.

Keywords Fixed priority schedulers, data-flow, real-time, multiprocessors, computational load, embedded systems, digital signal processing.

Abstract

Due to the technological evolution that happened recently, embedded systems with multiprocessing capabilities are becoming common. Application requirements often impose resource constrains, leading to the necessity of distributing them in an efficient manner.

This type of devices counts normally with a general purpose processor, typically from the ARM family, and one or more task specific processors, such as vector processors (EVP), used in digital signal processing systems for instance.

The resource distribution through the tasks is done by a scheduler. The scheduling can be done through one of the known scheduling policies: Round Robin, Fist In First Out, Time Division Multiplexing, Fixed Priority, etc.

The main goal with this project is to investigate fixed-priority real-time schedulers, with special focus to streaming applications executing on multiprocessor platforms, using dataflow.

Dataflow is a paradigm that uses graph theory for modelling, programming and analysis of applications and systems.

The fist part of this project is dedicated to the analysis and modelling of fixed priority dataflow graphs with shared resources distributed through a fixed priority scheduler. The second part is dedicated to the study of interference between tasks with different levels of priority on independent graphs, when mapped to execution on the same processor.

Embedded systems frequently have high priority tasks (periodic or sporadic) that need to be dispatched as soon as they become ready to execute. This action is going to interfere in the execution of tasks that are running in the same platform but with lower priority levels, since they are going to be blocked during the execution of the high priority tasks. This interference has two direct consequences: a lower response time for the high priority tasks and an increase in the execution time for the tasks in lower priority levels.

With our work, we intend to investigate the advantages and disadvantages that a fixed priority scheduler can offer in this type of situations, when compared with other schedulers.

Contents

Chapter 1

Introduction

Multi processor systems are getting common nowadays. Due to the technological advances in this area, today it is more practical and efficient to create systems with more than one processor, relinquishing specific tasks to specific processors. This devices are known as Multi Processor Systems on Chip [MPSoC]. By doing this, not only we can benefit from the parallel execution of tasks but we can also use some unique traits of a processor to increase our processing capability. Most computing embedded systems that perform some digital signal processing possess at least two types of processing units: a general-purpose core and a vector processor. All the flow control decisions are performed by the general-purpose processor while the processing of vectors and matrix operations are done in the vector processor, taking advantage of its capability of handling multiply-accumulate operations on many input values simultaneously.

In order to maximize the productivity of such devices it is usual to map several applications on the same MPSoC device. With such computational power at our disposal, we need an efficient mechanism to distribute the computational load through the available platforms. Every computational system with limited shared resources, like memory, processor cores or peripheral access among others, needs a proper resource sharing mechanism. A scheduler is essentially a program that coordinates the access to resources. In most embedded systems, it is the scheduler who decides which task can be executed at some point in time. Since every task has a defined number of resources that it need to execute, the scheduler is the one responsible for ensuring that a given task can only be set to execution when the corresponding set of resources is available.

Due to the nature of the applications where embedded systems are designed to, it is expectable that they perform at least one or more tasks with real-time computing constraints.

This dissertation focuses in two major goals: the characterization of fixed priority graphs, i.e, determination of the worst-case response-time and start times for the tasks that compose the system, and the study of the interference between tasks with different priorities when mapped into the same processing platform.

In the remainder of this chapter, we will define the fundamental concepts needed to understand and define our problem.

1.1 Fixed priority scheduling: A historical review

A real-time system is one with explicit deterministic or probabilistic timing requirements. Historically, real-time systems were scheduled by cyclic executives, constructed in a rather *ad*

hoc manner. During the 1970s and 1980s, there was a growing realization that this static approach to scheduling produced systems that were inflexible and difficult to maintain [11]. More advance techniques where required for the design, analysis and implementation of hard-real time systems. It was hoped that these techniques would provide additional flexibility whilst enabling the predictability of such systems to be guaranteed.

Subsequently, a wide range of scheduling strategies have been proposed. These strategies can be characterized by their prescribed run-time behaviours and the forms of associated analysis provided for predicting/optimizing system behaviour. At one extreme, for a simple application model, *static scheduling* (cyclic executives) provides very deterministic yet inflexible behaviour. The other extreme is often known as *best-effort scheduling* [18]; it facilitates maximum-run time flexibility, but, at best allows only probabilistic predictions of run-time performance. Fixed priority scheduling falls between these two extremes: it is often criticised as being too static by the proponents of best-effort scheduling and as being too dynamic by the supporters of cyclic executives. However, it is a predictable approach: off-line guarantees regarding process deadlines can be afforded using appropriated analysis. In reality it represents a practical, highly effective approach to scheduling a large class of real-time applications.

Work in a fixed priority scheduling concentrated on two separate issues: policies for the assignment of priorities to processes and feasibility tests for process sets. The assumptions and constraints for much of this work are identical to those described by Liu and Layland in 1973 [24]:

1. All processes are periodic;
2. All processes have a deadline equal to their period;
3. All processes are independent;
4. All processes have a fixed computation time;
5. No process may voluntarily suspend itself;
6. All processes are released as soon as they arrive;
7. All overheads are ignored (assumed to be 0).

Development of real-time theory progressed steadily, before a resurgence in the 1980's. The motivation for this renewed interest stemmed for many diverse factors, including the realization that the requirements of hard (i.e safety critical) real-time systems outstripped available theoretical analysis (for example, formal methods, scheduling theory etc.) and implementation techniques. Typical real-time systems implemented prior to the mid 1980's included basic avionics control, laboratory control etc. Looking forward from this point in time, the future real-time systems were considered to be applications such as the space station, robots, intelligent manufacturing and advanced avionics control. The common requirements shared by these systems were the need for dynamic and adaptive behaviour, including elements of artificial intelligence, together with an increased demand for predictability and reliability.

Another factor in the renaissance of real-time systems research was the rapid development of hardware (e.g minicomputers in the 1970's and microcomputers in the 1980's) which led to renewed interest in real-time systems for many diverse applications. Powerful distributed systems,

with multiprocessor nodes, became available for use in the real-time domain. Individual processors became more complex, with the inclusion of pipelines and caches, and peripheral devices became more intelligent. The availability of such hardware within the context of hard real-time applications prompted further work in terms of analysis [2].

1.2 Streaming applications

A **streaming application** is an application that operates over a long (potentially infinite) sequence of input data items, also refereed as *data stream*. The data is fed into the application normally from an external source and each data token is processed in a limited time before being discarded [34]. This process outputs also a long (potentially infinite) sequence of output data items. This type of applications are common in signal-processing functions where we have always some type of antenna as the external source of data. In this situation, the application has no control over the incoming or volume of the data to be processed. As examples of streaming applications we can indicate software-defined radio, radar tracking, audio and video decoding, audio and video processing, cryptographic kernels or network processing [26]. Streaming applications follow a reactive model and, when the application requires synchronization with the data stream, temporal restrictions are also applied to it.

1.3 Real-Time applications

The validity of the results produced by a **real-time application** are depended on their functional correctness, as on any other type of application, but also from the time in which these results are produced. Although correct, an output from a real-time application may be irrelevant if it violates its temporal deadline. The term "may" used on the last sentence implies the existence of more than one type of real-time systems. A real-time application can be categorized into three types according to its temporal restrictions [6]:

- **Soft** - If this type of restriction is violated, the associated result maintains some of its utility to the application, although there is degradation in the quality of service.

  Let us consider an automated gate as an example. If there is a significant delay between the reception of the activation signal for the open button and the activation of the gate motor, it is annoying for a driver but the end result it is still usable.

- **Firm** - If a firm deadline is overdue, the consequent result is unusable but the integrity of the system and the user are not compromised. As an example we can refer data collected from a sensor array that it is used for autopilot navigation. If some of the samples arrived after the established deadline, they are useless. But as long as some other data arrives in a timely fashion, the system is still able to function correctly.

- **Hard** - For this type of restrictions, a deadline violation could also imply a catastrophic consequence to the system. Every critical security system is characterized by having at least one **hard** temporal restriction. As an example we can refer to a life support system or the traction control system of a car. If the control system is not able to meet its deadlines, the integrity of the user could be put in danger[19].

In a more broad sense, real-time systems can be now categorized into two major types according with the previous temporal restrictions:

- **Soft Real-Time** - These systems only possess *soft* or *firm* temporal restrictions
- **Hard Real-Time** - All the systems that possess at least one *hard* temporal restriction are categorized under this label.

1.3.1 Timing requirements

Timing requirements come in two basic types: throughput and latency. If the rate at which an iterative application produces results is important, then we are in the presence of a throughput requirement. If the minimum or maximum time interval between the arrival of an input and the production of the corresponding output are to be respected, then the application has a latency requirement. A heart rate monitor is an example of an application with throughput requirements. In order to return a correct value for this measure, all the heart beats in a given time interval must be read and the time between them must be respected, although the processing and output of the final value could suffer some delay resulting in a service degradation. The navigation and actuation signals in a car exemplifies a system with latency requirements. It is important that the maximum time between the actuation on the brake pedal and the actuation on the brake system is respected, for instance. In this case, due to the random nature of all the possible stimulus to the system, no throughput requirements are present, at least not in the systems considered.

Temporal requirements can also appear in the form of a required worst-case timing, best-case timing or both. If the worst-case timing coincide with the best-case timing, the result is designated as an on-time requirement. In this project we concentrate our attention in worst and best-case timing calculations.

1.3.2 Scheduling

A typical computational system is comprised of several resources (processors, memory, peripheral devices, etc.) that should be used concurrently by different tasks. These resources need to be assigned to the concurrent tasks in an orderly and efficient fashion. The set of predefined criteria that regulates the allocation of resources to tasks is called a *scheduling policy*. The set of rules that, at any time, determines the order in which tasks are executed is called a *scheduling algorithm*. The specific operation of allocating a resource to a task selected by the scheduling algorithm is referred as *dispatching* [6]. There are several known scheduling algorithms in existence: First In First Out, Round Robin, Shortest Remaining Time, Fixed Priority, Time Division Multiplexing, etc. Every one of these algorithms has advantages and disadvantages that had been studied throughout the years. Our project will be focused mainly on the Fixed Priority scheduling algorithm.

1.4 Fixed priority scheduling applications

In a fixed priority scheme, all tasks are characterized by an immutable priority value. Normally this value is a numeric one. The order in which these values are assigned depends essentially on the system specifications but conventionally higher priorities receive smaller values. Conceptually, if the tasks are ordered with decreasing priority, T_i has smaller priority that T_j if $i > j$,

where T designates a task, i and j indicate numerical priority values with $i, j \in \mathbb{N}_0$. The scheduler uses priorities to determine the next job to be scheduled. These are calculated at design time and never change during execution, hence the term *fixed* [33]. In fixed priority scheduling, the dispatcher will make sure that at any time, the highest priority runnable task is actually running.

1.4.1 Preemption

In a pre-emptive system, if we have a task with a low priority running, and a high priority task arrives, i.e, some event had occurred and the dispatcher needs to deploy a task into execution, the low priority task will be suspended and the high priority task will start running. If while the the high priority task is running, a task with a medium priority arrives, the dispatcher will leave it unprocessed and the high priority task will carry on running, finishing its computation in a later time. Only when both the high and medium priority tasks have completed can the low priority task resume its execution. This low priority task can then carry on executing until either more higher priority tasks arrive or it has finished its work [35]. If the platform in use does not support preemption, then the tasks with higher priorities are only set to execution ahead of the lower priority ones if they could be started at the same time instant. Otherwise, if a lower priority tasks is already executing in the platform when a higher priority task becomes ready for execution it just gets blocked, at least until the executing task finishes its current execution.

1.5 State of the art

1.5.1 Classical real-time theory

Real-Time is a subject that has been studied for some time, which led to the development of a considerable theory around it, known nowadays as *classical real-time theory*. In this introductory section, we are going to focus only on *on-line scheduling with fixed priorities* with special attention to the two main criteria for classical scheduling using fixed priority: the **rate-monotonic** and the **deadline monotonic** criteria [6] [25]. This type of scheduling has some advantages regarding off-line scheduling. Namely:

- Any alteration in the tasks characteristics is immediately taken into account by the scheduler.
- It can easily accommodate sporadic tasks.
- Deterministic behaviour on overloads since it only affects the tasks with lower priorities.

As expected, there are some disadvantages that go with the pros mentioned before:

- The on-line scheduling has a more complex implementation since it requires a kernel with fixed priorities.
- This type of scheduling requires the action of a *scheduler* and a *dispatcher*, which implies a higher execution *overhead*.
- Overloads, due to software or project errors, may block all tasks bellow a given priority.

1.5.1.1 Rate-Monotonic scheduling

The Rate Monotonic (RM) scheduling algorithm is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates, which means shorter periods, will have higher priorities and vice-versa. Since periods are constant, RM is a fixed-priority assignment: a priority P_i is assigned to the task before execution and does not change over time. For the remaining of this section, we will assume the existence of *preemption* by the platform. In the initial analysis performed in [24], the Rate Monotonic algorithm is intrinsically pre-emptive, and all the tasks are independent, i.e, there are no shared resources. In this context, a running task will be preempted by a newly arrived task with shorter period. Since the schedule is built on-line, it may be useful to know *a priori* if a given set of tasks respects its temporal requirements. To aid us in this subject there are two main types of tests that can be performed upon the task set:

- **Tests based on the utilization rate of the CPU** - These consists in inequalities applied to the tasks characteristics, such as their worst-case execution time, period and deadline. The verification of these inequalities allow us to conclude if a given task as guaranteed activations or not. The two reference criteria for this subject are the **Minor bound of Liu and Leyland** and the **Hyperbolic bound of Bini, Buttazzo and Buttazzo**. A more detailed explanation of each can be found in [24] and [3] respectively. Our project does not deal directly with local deadlines, so we will not progress any further in this subject.

- **Tests based in the response-time** - For systems with arbitrary fixed priorities, the analysis of the response-time allow us to perform a schedulability test that, assuming that the system allows preemption and synchronous activation, is necessary and sufficient. These tests consists in computing the *worst-case response-time*, i.e, the maximum elapsed time between the activation of a task and its completion, and then check if it is below the deadline. For further information, please refer to [1].

1.5.1.2 Deadline-Monotonic scheduling

The Deadline Monotonic (DM) priority assignment weakens the "period equals deadline" constraint within a static priority scheduling scheme. The application of the scheduling algorithm assumes that every task is characterized by a phase ϕ_i, a worst-case constant computation time C_i for each instance, a constant relative deadline D_i and a period T_i. According to the DM algorithm, each task is assigned a fixed priority P_i, inversely proportional to its relative deadline D_i. Thus, at any instant, the task with the shortest relative deadline is executed. Since relative deadlines are constant, DM is a static priority assignment. As in Rate Monotonic, DM is normally used in a fully pre-emptive mode. [6]

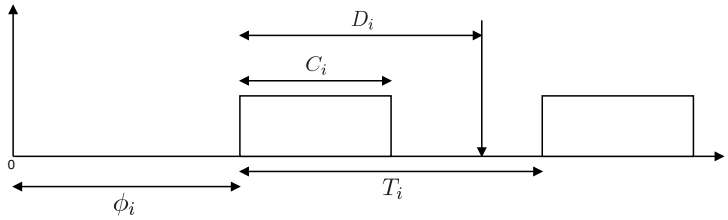

Figure 1.1: Gantt chart with indication of the various task characteristics

The Deadline-Monotonic priority assignment is optimal for independent tasks and when the period is equal to the deadline, meaning that, if a task set is schedulable by some fixed priority assignment, then is also schedulable by DM. The proof of this assumption and a more detailed explanation on this algorithm can be found in [23]. A more comprehensive overview on Rate-Monotonic and Deadline-Monotonic scheduling is available in [2].

1.5.2 SymTA/S

SymTA/S is a system-level performance and timing analysis approach based on formal scheduling analysis techniques and symbolic simulation. It is essentially a software tool used to determine system-level performance data such as end-to-end latencies, bus and processor utilization and worst-case scheduling scenarios. SymTA/S focus its utilization mainly on MPSoC designs, where the complexity level achieved due to all the concurring hardware makes manual analysis and optimization a very time consuming and prone to errors task.

The core of the SymTA/S tool is a technique to couple local scheduling analysis algorithms using event streams. For a more detailed description of these algorithms, please refer to [29] and [30].

In order to perform a system level analysis, SymTA/S locally performs existing scheduling analysis using a well know algorithm, like for example Rate-Monotonic, Time Division Multiple Access, Round Robin, etc., and propagates their results to the neighbouring components. This analysis-propagate mechanism is repeated iteratively until all components are analysed, which means that all output streams remained unchanged.

A more accurate description of this tool can be found in [16] and [15].

1.5.3 Real-Time calculus

Real-Time Calculus establishes a link between three areas, namely Max-Plus Linear System Theory [9] as used for dealing with certain classes of discrete event systems, Network Calculus [4] for establishing time bounds in communication networks, and real-time scheduling. In particular, it shows that important results from scheduling theory can be easily derived and unified using Max-Plus Algebra. In its essence, Real-Time Calculus focus on the characterization of sets of task $T_1, \ldots, T_i, \ldots, T_n$ by a request and a demand curve α_r^i and α_d^i respectively. These tasks are all processed by one processing unit characterized by a delivery curve β using a static priority scheduler with preemption. It is important to refer that the tasks are sorted with decreasing priority.

The algorithm consists in an iterative process to determine the tasks priorities such as the whole task system can be successfully scheduled. The process consists in selecting the tasks in increasing order of priority and perform a schedulability test based on the task deadlines, demand curve and the delivery curve of the processing unit. If any of the tasks fails this test, the whole set can not be scheduled. Otherwise, the schedulable task is removed from the set and the whole selection procedure is repeated until there is no more tasks left. A more comprehensive description of this algorithm can be found in [33].

1.6 Data flow graphs

In this project we intend to use the data flow paradigm to tackle the fixed priority scheduling problem. Data flow has developed into a useful tool, with extensive use in the analysis of streaming applications, modelling multiprocessor environments and dealing with concurrent applications. The application of data flow in the situations indicated is done through the use of graph theory to establish mathematical models for analysis using the tools provided by the paradigm.

In the most general sense, a data flow graph is a directed graph with actors represented by nodes and arcs representing connections between the actors. These connections convey values, corresponding to data packets, also designated as tokens, between the nodes. Connections are conceptually FIFO queues which permit initial tokens on them.

The operation in which an actor consumes a certain number of tokens from its incoming edges and then starts executing is known as an **actor firing**. The set of rules that control this firing, namely the minimum number of tokens present in the incoming edges, is know as the **firing rules**.

If actors are permitted to produce and consume only one token per activation, the resulting graph is designated as a *Single Rate Data Flow* graph. If, on the other hand, an actor can consume and produce multiple tokens in its activations, the graph is now known as a *Multi Rate Data Flow* graph. Independently of the rate of consumption and production of the actors, if the quantity of tokens in any actor operation is constant and well defined, we obtain a *Synchronous Data Flow* graph [5].

All these concepts will be addressed in greater detail in future chapters.

1.7 Problem description

Embedded platforms for streaming applications are expected to handle several streams at the same time, each one with its own rate. This functionality can be divided in jobs. A job is a group of communicating tasks that are started and stopped independently. The approach that has been taken so far for analysis resorts to the modelling of these systems using data flow graphs [21].

The overall scheduling strategy used mixes static (compile-time) and dynamic techniques (run time). The scheduling of tasks that belong to the same job, or intra-job scheduling, is handled by means of static order, i.e, per job and per processor, a static ordering of actor is found that respects the Real-Time requirements while trying to minimize processor usage. Inter-job scheduling is handled by means of local Time Division Multiplex (TDM) schedulers.

The biggest disadvantage of TDM schedulers is that they waste many resources for low-latency, low throughput tasks.

The goal of this project is to investigate how the flow must be changed to allow the usage of a non-budget-based scheduler, such as Fixed Priority. In order to achieve this goal, we must follow the following steps:

- Determine whether the data flow analysis is still possible under these conditions and under which conditions analysis can still be carried out.
- Propose a method for priority assignment per processor per job and design the scheduler in such a way that it works well for relevant applications.

- The resource manager has to be adapted to handle a Fixed Priority schedule.

The processor usage is an important factor to take into account. In shared resources platform, like the MPSoC devices that we refer in this document, the response-time of a task or a job is related to the capacity that a particular resource, specifically a processor, has to process that instance. On the other hand, this capacity or processor availability is related to the computational load required from other jobs or tasks with higher priority.

The analysis and characterization of the computational load of a shared processor is also one of the focal points of our project.

1.8 Developed work

The organization of this project followed the points established in the previous section. The contributions of this project to the state of the art can be summarized into the following points:

1. **Data Flow Analysis** - A comprehensive analysis of data flow models of fixed-priority systems comprised the bulk of our initial work. This analysis was centred in the characterization of *best* and *worst* case response-times for fixed-priority data flow graphs. Initially this analysis considered the whole system mapped on a single processing unit and later on, the behaviour of the same type of systems mapped on different platforms was studied, giving emphasis to the dependence and interference between tasks the same job but mapped on different processing units.

2. **Computational load analysis** - We formalized the concept that quantifies the amount of work required from a processor by a particular task. In a Fixed-Priority scheduling, it is useful to characterize the amount of time that a processor is busy with a high priority task, thus allowing us to determine the availability of the same processor to execute lower priority tasks.

3. **Extension of the tools available** - For the analysis of all the systems conceived to study the fixed-priority approach to this scheduling problem we had at our disposition a set of software tools, namely a data flow graph simulator.

   These tools did not contemplate either the simulation of fixed priority data flow graphs or the functionalities to perform load analysis of a processing unit. In order to obtain reliable results to support our study, it was necessary to add these functionalities.

   In order to simplify the readability of the results provided by this set of tools, we also included the necessary changes for an integration with an external visualization tool.

1.9 Thesis organization

The remainder of this thesis is organized as follows: in chapter 2 we review data flow computation models and their analytical properties. The mathematical notation for representing data flow graphs is also introduced in this chapter. The software framework used throughout this project is introduced in chapter 4, which includes a detailed explanation of the usage and functioning of the set of tools available. The changes and implementations made to provide the necessary functionalities for our project are described in 4. Chapter 5 details the analysis of

fixed-priority data flow graphs, which includes all the theory developed and respective software implementations to obtain results. Chapter 6 follows a similar template of the previous chapter but now relative to inter-graph fixed priority analysis. The practical results, either from software simulations or from analysis of practical examples, and their respective discussion are presented in chapter 7. Chapter 8 states our conclusions and suggests future work.

Chapter 2

Data Flow computation models

This dissertation uses data flow computation models for modelling and analysing various systems. In this chapter, we present the notation for the data flow model that we will use throughout this document and the properties of several data flow computation models that are relevant to our work. This is reference material and most of it can be found in [26] [5] [28] [32] [21].

2.1 Graphs

In this dissertation, we use data flow analysis, which in turn uses graph theory in its formalization. Therefore we need to first introduce graph theory.

2.1.1 Directed graphs

Definition 2.1. *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 is the **sink node** of e. We also denote the source and sink nodes of e as $src(e)$ and $snk(e)$, respectively.*

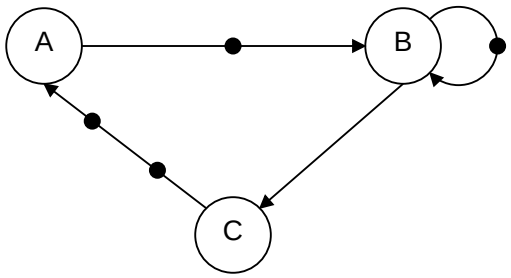

Figure 2.1: An example of a directed graph

The graph depicted on the previous figure is described by the following sets:

$$V = \{A, B, C\} \tag{2.1}$$

$$E = \{(A, B), (B, C), (B, B), (C, A)\} \tag{2.2}$$

It is also a directed graph: node A is directed to node B, node B is directed to node C and itself through a self-edge and node C is directed to node A.

2.1.2 Path 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 **transverses** $src(e_1), src(e_2), ..., src(e_n)$ and $snk(e_n)$; the path is **simple** if each node is traversed once, that is $src(e_1), ..., src(e_n), snk(e_n)$ 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 $(e_1, e_2, ..., e_{n-1})$ is a simple path and $src(e_1) = snk(e_n)$[26].

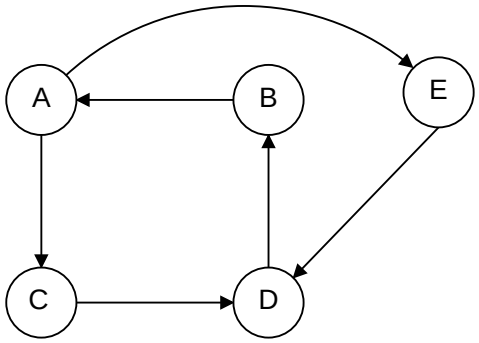

Figure 2.2: Example of a graph with a simple path

In the previous figure, the simple path $\{(A, C), (C, D), (D, B), (B, A)\}$ describes a cycle.

2.2 Data Flow

Data flow is a natural paradigm for describing Digital Signal Processing applications for the concurrent implementation on parallel hardware. Data flow programs for signal processing are directed graphs where each **node** represents a function and each **arc** represents a signal path. More specifically, in a data flow graph, **nodes** represent **actors**. An **actor** is a time consuming entity associated with **firing rules**. An **edge** or **arc** in a data flow graph represents a **First-In-First-Out** queue that directs values from the output of an **actor** to the input of another.

In data flow, data is transported in discrete chunks, referred to as **tokens**. When an **actor** starts an execution, it *consumes* a defined number of **tokens** from its incoming **edges**. Conceptually, this consumption is a *reading* operation of the data tokens that are needed for beginning the execution. These tokens remain in the edge (FIFO) during the execution of the actor. By the end of that execution, the **actor** *produces* a defined number of **tokens** into its outgoing edges. This production process is a *writing* operation onto the outgoing edges (FIFOs). It is also possible to perform a reservation of space in the outgoing edges during the start of the execution in order to assure that, once finished, the actor has enough memory available to write the processed data.

Synchronous Data Flow (SDF) is a special case of data flow in which the number of data tokens produced or consumed is specified *a priori*.

The data flow principle is that any **actor** can **fire** (perform its computation) whenever input data are available on all of its incoming **edges**. A **actor** with no input **edges** may **fire** at any time. This implies that many **actors** may fire simultaneously, hence the concurrency. Because the program execution is controlled by the availability of data, data flow programs are said to be *data-driven* [21].

2.2.1 Actor firings

At this point, it is useful to define the **firing** concept in the data flow context, since the same will be referred in the future.

As described in the previous section, in data flow, every **edge** has also two associated valuations: $prod : E \to \mathbb{N}$ and $cons : E \to \mathbb{N}$. For a given edge $e \in E$, $prod(e)$ gives the constant number of tokens produced by $src(e)$ on e in each firing and $cons(e)$ gives the constant number of tokens consumed by $snk(e)$ in each firing.

An **actor firing** is an indivisible quantum of computation. A set of **firing rules** give preconditions for a firing. Firing consumes tokens from the input streams and produces tokens into the output streams. The firings themselves can be described as functions, and the invocation of these firings is controlled by *firing rules* [20].

The **start time** of a firing refers to the time instant at which the firing rules are verified and the tokens from the input streams are consumed. We are going to use the following notation:

$$s(i,k) = m, \quad m \in \mathbb{N}_0 \tag{2.3}$$

where i is denotes the actor and k the instance of the activation.

As such, the **finish time** of a firing corresponds to the time instant at which the tokens resultant from the computation are produced into the output streams. Just like for the start time, to indicate a particular finish time we refer to a similar notation

$$f(i,k) = m, \quad m \in \mathbb{N}_0 \tag{2.4}$$

An **actor firing** can be designated as a **task instance** in some contexts. **Task instances** are used mostly in classical real-time theory while **actor firings** are their counterpart in data flow.

2.3 Temporal analysis

Execution time of an actor - τ Before the definition of Timed SRDF it is important to define the concept of *execution time* of an actor.

Definition 2.2. *The **execution time** $\tau(i)$ of an actor i is the elapsed time between the **start time of the firing** for that actor and the **finish time of the firing**, at the end of that execution. The execution time can be defined in a more general sense, as $\tau(i)$, in which it is assumed that all executions of actor i have a constant execution time, or it can be specified as $\tau(i,k)$, where k indexes the execution time of a particular firing, within an execution of a graph.*

Since the exact time of a particular instance of a task can be hard or even impossible to know in advance, for analytical purposes it is often convenient to use bounds to this value.

A given execution time of an actor i can be upper bounded by a *worst-case execution time* $\hat{\tau}(i,k)$ and be lower bounded by a *best-case execution time* $\check{\tau}(i,k)$. The following property must always hold:

$$\check{\tau}(i) \leq \tau(i,k) \leq \hat{\tau}(i), \quad \forall i \in G, \forall k \in \mathbb{N}_0 \tag{2.5}$$

2.3.1 Schedules

In the context of this problem, it is necessary to develop a concise definition of schedule that is consistent with the type of result that we plan to obtain. At this point it is important to make a distinction between *schedulers* in an implementation, as for example Fixed Priority, Time Division Multiplexing, Round Robin etc., and the execution of data flow graphs using a schedule, as for instance a Self-timed or a Static Periodic schedule. This section will address the latter. It is important to indicate from the beginning that, in this context, we will work with *Self-timed* schedules. A more intuitive designation for this type of schedules is *ASAP Schedules* (As Soon As Possible) since the start times vector for every actor is determined from the principle that every task should start as soon as it has conditions for it. So, in a similar way that schedulers had been defined in other situations, a scheduler is defined to a specific actor i, which, in our definition, is preceded by another actor j. The edge connecting both actors posses a number $d(i,j)$ of tokens on it, as the next figure illustrate:

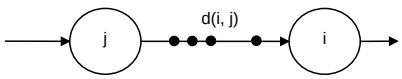

Figure 2.3: Simple arrangement of two actors connected through an edge

From this arrangement, we can write the following expression for the schedule of actor i:

$$s_{SelfTimed} = \begin{cases} -\infty, & k < 0 \\ \max(\max_{\forall (i,j) \in E}(s(j,(k-d(i,j))) + \tau(j)), 0), & k \geq 0 \end{cases} \tag{2.6}$$

For a two actor arrangement as the one in the previous figure, we can elaborate the following logic:

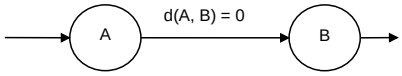

Figure 2.4: Two actors connected through an edge with no tokens

From this figure we can write that:

$$s(B,k) \geq s(A,k) + \tau(A) \tag{2.7}$$

The start time of the k^{th} iteration of actor B is going to be always $\tau(A)$ time after the start time of the k^{th} iteration of the precedent actor A. But if we have some tokens between the actors, then:

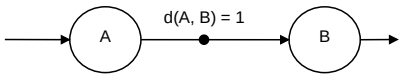

Figure 2.5: Two actors connected though an edge with one token

With a token in the edge connecting the two actors, the previous expression 2.7 needs to be adapted:

$$s(B,k) \geq s(A,k-1) + \tau(A) \tag{2.8}$$

Since now actor B does not need to wait for actor A to produce at least one token for it to start executing, the start time of this actor is now referenced to the $(k-1)^{th}$ iteration of the precedent actor. If we expand this logic to $d(A,B)$ tokens in the interconnecting edge, we reach the bottom branch of expression 2.6. Since a negative value for the start time of an execution does not make sense in the context of this problem, we included the 0 argument in the *max* expression, so that in such a case, the minimum start time of a execution is going to be zero.

The **Worst-Case Self-Timed Schedule** of an SRDF graph is the self-timed schedule of an SRDF where every iteration of every actor i takes $\hat{\tau(i)}$ to execute and where $\hat{\tau(i)}$ is the worst-case execution time of the actor. Note that the WCSTS of an SRDF graph is unique.

2.3.2 Single Rate Data Flow

If in a data flow graph we can verify that $prod(e) = cons(e)$ for every edge $e \in E$, then the graph is a **Single Rate Data Flow** (SRDF) graph. A SRDF graph is one where every actor in it consumes and produces the same number of data tokens. We can formalize this concept with

$$G_{SRDF} = (V, E, d, \tau) \tag{2.9}$$

V and E are already defined in definition 2.1. d is a valuation $d : E \to \mathbb{N}_0$. $d(i,j)$ is called the delay of edge (i,j) and represents the number of initial tokens in arc (i,j).

2.3.3 Timed Single Rate Data Flow graphs

We can now include the execution time of every actor of the graph into consideration and define a *Timed SRDF* graph:

$$G_{TimedSRDF} = (V, E, d, \hat{\tau}) \tag{2.10}$$

where $\hat{\tau}$ represents the worst-case response-time of an actor.

2.3.4 Application graphs

In the course of our work, we realized that we need to further specify the definition of Timed SRDF referred above, by including a new parameter into consideration. Our new graph instance differs just slightly from equation 2.10:

$$G_{app} = (V, E, d, \check{\tau}, \hat{\tau}) \tag{2.11}$$

Where $\check{\tau}$ represents now the best-case response-time.

2.4 Modelling schedulers in data flow analysis

In order to use data flow to analyse a particular schedule, first it need to be modelled using the data flow paradigm. In the present section we will present strategies to perform this modelling, using concrete examples as to illustrate the process.

2.4.1 Task scheduling

There are two types of task scheduling mechanisms that we are interested in modelling: Compile-Time and Run-Time Scheduling.

Compile-Time Scheduling (CTS) encompasses scheduling decisions that are fixed at compile-time, such as static order scheduling.

Run-Time Scheduling (RTS) refers to scheduling decisions that cannot be resolved at compile-time, because they depend on the run-time task-to-processor assignment, which in turn depends on the dynamic job-mix. This is handled by the local scheduling mechanism of the processor. Modelling the worst-case effect of the local scheduler on the execution of an actor is needed to include in the compile-time analysis the effects of sharing processing resources among jobs. If the WCET of the task, the settings of the local dispatcher, and the amount of computing resources to be given to the task are known, then the actor execution time can be set to reflect the **worst-case response-time** of that task running in that local dispatcher, with that particular amount of allocated resources [26].

2.4.2 Time Division Multiplexing scheduling (TDM)

The effect of a TDM scheduling can be modelled 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 fire i, when resource arbitration effects (scheduling, preemption, etc) are taken into account. This is counted from the moment the actor meets its enabling conditions to the moment the firing is completed. Assuming that a TDM wheel 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 $\tau(i)$ passes from the moment an 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, once the firing conditions of the actor are met. 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 [12]:

$$\hat{r}(i) = P - S \tag{2.12}$$

2.4.3 Non-Preemptive Non-Blocking Round-Robin scheduling

In a Non-Preemptive Non-Blocking Round-Robin (NPNBRR) scheduler, all clusters assigned to the same processor are put in a circular scheduling list. The run-time scheduler goes through this list continuously. It picks an actor from the list and tries to execute it. The actor (or the scheduler, depending on the implementation) checks for input data and output space availability.

If there are sufficient input data tokens available and available storage space in all output FIFOs such that the actor can consume and produce tokens according to its firing rules, the actor executes until the firing is over, if not, the actor is skipped. The process is repeated for the next actor in the circular scheduling list, and so on.

The worst-case arbitration time of an actor is given by the sum of the execution times of all other actors mapped to the same NPNBRR-scheduled processor. The processing time is equal to the actor's execution time, since there is no preemption. The total response-time is therefore equal to the sum of execution times of all actors mapped to the NPNBRR-scheduled processor [27].

2.4.4 Static-Order scheduling

A static-order schedule of a set of actors $A = \{a_0, a_1, ..., a_n\}$ mapped to the same processor is a sequence of execution $so = |a_k, a_l, ..., a_m|$ that generates extra precedence constraints between the actor in A such that from the start of the execution of the graph, a_k must be the first one to execute, followed by a_l and so on, up to a_m. After a_m executes, the execution restarts from a_k for the next iteration of the graph.

Any static order imposed to a group of Single Rate Data Flow actors executing in the same processor can be represented by adding edges with no tokens between them. From the last to the first actor in the static order, an edge is also added, with a single initial token. This construct reflects the fact that, the graph execution being iterative, when the static order finishes execution for a given iteration, it restarts it from the first actor in the static order for the next iteration.

Notice that the new edges represent a series of sequence constraints enforced by the static order schedule and do not represent any real exchange of data between the actors.

2.4.5 Static periodic schedulers

A **Static Periodic Scheduler (SPS)** of an SRDF graph is a schedule such that, for all nodes $i \in V$, and all $k > 0$:

$$s(i, k) = s(i, 0) + T \cdot k \tag{2.13}$$

where T is the designed period of the SPS. Please note that an SPS can be represented uniquely by T and the values of $s(i, 0), \forall i \in V$ [26].

2.5 Data Flow temporal analysis techniques

Temporal analysis is required in order to verify whether a given timed data flow graph can meet a required throughput or latency requirement of an application. In this section we will cover some of the analysis methods available in this regard.

2.5.1 Throughput analysis

In some systems, rate constraints are often imposed by designers on the execution rate of each process in the system in order to ensure correct timing behaviour and achieve performance goals. This type of restrains are known as throughput constraints.

The execution of a data flow graph can be divided into two phases: a transient and a periodic one. These phases occur in the same order as they were mentioned. When the execution of the graph is initiated, the transient phase begins. This phase has a limited duration where the initial tokens are distributed through the edges of the graph. Eventually the graph enters the next phase: the periodic one.

The state of a graph is defined by the amount of tokens present at each one of its edges. Whenever a graph enters in the periodic phase, the same sequence of states repeats itself recurrently. The time period required to repeat the same sequence of states is defined as the graph period.

In the transition phase, the throughput analysis can be derived by simulating the execution of the data flow graph, given worst-case execution times to all actors [26]. Another known technique for temporal analysis is the **Maximum Cycle Mean**. A simple explanation of these two techniques follows:

2.5.1.1 Simulation

This is perhaps the most direct approach to this problem. By running a reliable simulation of the data flow graph, it is possible to verify if the throughput requirements are met or not. The simulation tool that we used, which is going to be described in detail in chapter 3, provides enough information so that, in case of violation of the throughput specifications, one can adjust the graph characteristics (if possible) in order to obtain a throughput compliant graph.

2.5.1.2 Maximum Cycle Mean

The average weight of a directed cycle is the quotient between the summation of the execution time of all of its actors and the total number of initial tokens present in the cycle, and is called *cycle mean*. The *maximum mean cycle* problem for a directed graph with cycles is to find a cycle having the maximum average weight, called the *maximum cycle mean*, over all directed cycles in the graph. Such a cycle is called a critical cycle. The maximum mean cycle problem has applications in finding the iteration bound of a data flow graph for digital signal processing, in performance analysis of synchronous, asynchronous, or mixed systems, and on throughput analysis for embedded systems [10].

The **Maximum Cycle Mean (MCM)**, $\mu(G)$ of a SRDF graph G is defined as:

$$\mu(G) = \max_{c \in C(G)} \frac{\sum_{i \in N(c)} \tau_i}{\sum_{e \in E(c)} d_e} \tag{2.14}$$

where $C(G)$ is the set of simple cycles in graph G.

Theorem 2.1. *For an SRDF graph $G = (V, E, d, \tau)$, it is possible to find a **Static Periodic Schedule (SPS)** if an only if $T \geq \mu(G)$. If $T < \mu(G)$, then no SPS exists with period T.*

This theorem and respective proof are found in greater detail in [26].

2.5.1.3 Monotonicity

The monotonicity of a function, or in our case, of a self-timed execution, is a important concept to introduce at this stage since it had been proved very useful in this context.

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

Definition 2.3. ***Monotonicity:*** *A function $f(n)$ is* ***monotonic increasing*** *if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is* ***monotonically 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 $f(m) > f(n)$ [8].*

But we are more interested in the application of the monotonicity concept in a Single Rate Data Flow context, specifically when applied to self-timed schedulers. As such, we define monotonicity in this context as:

Definition 2.4. ***Monotonicity of a self-timed execution:*** *In a SRDF graph $G = (V, E, \tau, d)$ with worst-case self-timed schedule s_{WCSTS}, for any $i \in V$, and $k \geq 0$, it holds that, for any self-timed schedule s_{STS} of G*

$$s_{STS}(i,k) \leq s_{WCSTS}(i,k) \tag{2.15}$$

Because of the monotonicity of self-timed execution, if any given firing of an actor finishes it 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. This was defined as a theorem and proved in [26].

2.5.2 Latency analysis

Although throughput is a very useful performance indicator for concurrent real-time applications, another important metric is latency. Especially for applications such as video conferencing, telephony and games, latency beyond a certain limit cannot be tolerated. Usually, the dependencies on a SRDF graph allow some freedom in the execution order of the actors. This order determines performance properties like throughput, storage requirements and latency [13]. 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:

$$L(i,k,j,p) = s(j,p) - s(i,k) \tag{2.16}$$

where i and j are actors, p and k are firings. We say that i is the source of the latency and j is the sink. Another useful concept to take into account in this section is the maximum latency, here defined as:

$$\hat{L}(i,j,n) = \max_{k \geq 0}(s(j,k+n) - s(i,k)) \tag{2.17}$$

where n is a fixed iteration distance. Next we are going to refer some latency analysis techniques by presenting some concrete situations where this type of analysis is used.

2.5.2.1 Maximum Latency from a periodic source

In data flow, a source is an actor that models a generator of data, such as an antenna for example. Due to the unpredictable behaviour of the modelled device, a source can be represented as an actor that does not have a set of firing rules associated but produces tokens into its outgoing edges. A source can produce these tokens in a periodic or sporadic manner and the number of

tokens produced per activation can be fixed or variable. A periodic source is characterized by a period T that corresponds to the elapsed time between two consecutive productions.

As we have seen in section 2.4.5, the start times of a periodic source are given by:

$$s(i,k) = s(i,0) + T \cdot k \tag{2.18}$$

The period of the execution of the graph is imposed by the source since the source executes with a period T, the period of the execution of the graph is lower bounded by the period of the source. If on the other hand, the graph has a longer period, then it cannot keep up with the source, and infinite token accumulation on some buffer will happen for the WCSTS. Therefore, we will perform the latency analysis under the assumption that $\mu(G) = T$.

For the determination of the maximum latency for this case, we first need to establish the concept of **Rate-Optimal Static Periodic Schedule (ROSPS)**. This designation is attributed to a Static Periodic Schedule that has a period T equal to the MCM of the SRDF graph $\mu(G)$. Considering this concept, the maximum latency for a periodic source can be written as:

$$\hat{L}(i,j,n) = \max_{k \geq 0}(s_{STS}(j,k+n) - s(i,k)) \leq \check{s}_{ROSPS}(j,0) - s(i,0) + \mu(G) \cdot n \tag{2.19}$$

Where $\check{s}_{ROSPS}(j,0)$ represents the smallest time of j in an admissible ROSPS. We can determine the maximum latency for a periodic source just by calculating an ROSPS with the earliest start time j and a WCSTS for the earliest start time of i. A more extended approach to this subject, including a more detailed explanation on the logic behind expression 2.19 can be found in [26].

2.5.2.2 Maximum latency from a sporadic source

In reactive systems, it frequently happens that the source is not strictly periodic, but produces tokens sporadically, with a minimum interval μ between subsequent firings. Typically, a maximum latency constraint must be guaranteed. For any given graph with this type of source, it is mandatory that it has to be able to sustain a throughput of $1/\mu$ in order to guarantee that it cannot be overran by such a source, operating at its fastest rate. This means that the MCM of the graph, $\mu(G)$, is such that $\mu(G) \leq \mu$. The proof of this statement relies on the possibility of bounding the self-timed behaviour of a graph by static periodic μ, which is possible as long as $\mu(G) \leq \mu$.

In order to keep the simplicity of this chapter, we present the final expression for the maximum latency of a graph with a sporadic source, omitting all the steps that were taken in its deduction:

$$\hat{L}(i,j,n) \leq \check{s}_{\mu}(j,0) - s(i,0) + \mu \cdot n \tag{2.20}$$

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 μ.

For a detailed explanation for the logic behind the previous expression, please consult [26].

2.5.2.3 Maximum latency for a bursty source

A bursty source is characterized as a source that may fire at most n times within any T time interval, with a minimal Δt interval between consecutive firings. A job that processes such a source must have $\mu(G) \leq T/n$ to be able to guarantee its processing within bounded buffer space. Moreover, if $\mu(G) \leq \Delta t$, then we have the previous case, i.e, maximum latency from a sporadic source. If $\mu(G) \geq \Delta t$ then the latency may accumulate over iterations, as the job processes input tokens slower than the rate at which they arrive. The maximum latency must occur when the longest burst occurs, with the minimum interval between firings of the source, that is a burst of n tokens with Δt spacing. Because of monotonicity, making the source execute faster cannot make the sink execute slower, but it also cannot guarantee that it executes faster.

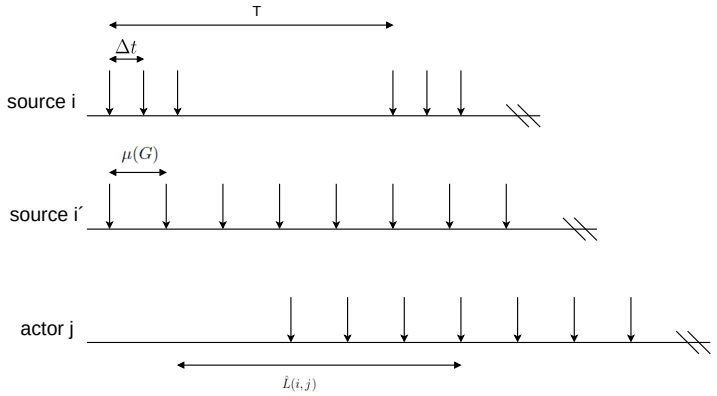

Figure 2.6: Arrival times from tokens of a bursty source relatively to strictly periodic source

As depicted in figure 2.6, the tokens of the bursty source i will arrive earlier than for the periodic source i'. Therefore, at iteration $n-1$ after the beginning of the burst (iteration 0) happens the earliest time:

$$s(i, n-1) = s(j, n-1) \leq \check{s}_{ROSPS}(j, 0) + (n-1) \cdot \mu(G) \tag{2.21}$$

As such, a bound on the maximum latency is given by:

$$\hat{L}(i, j, n) \leq \check{s}_{ROSPS}(j, 0) - s_{ROSPS}(i, 0) + (n-1)(\mu(G) - \Delta t) \tag{2.22}$$

2.6 Conclusion

By choosing a data flow model as a programming and analysis model, we can now use their analytical proprieties to our advantage. In this chapter were defined all the concepts essential to understand and use the tools provided by the data flow paradigm, which included a brief introduction to graph theory. Data flow is very useful for analysing streaming applications and modelling multiprocessor systems. All the systems that we are going to work with in future chapters fall into one of these categories, or both, and that is one of the main reasons why we choose this modelling tool as basis for our project.

Chapter 3

Software framework

The execution of our project relied heavily on the utilization of a set of software tools for obtaining results. The core of this set of tools is the Heracles data flow simulator. During the execution of this project, some questions were answered using the existing functionalities of this simulator, but as we progressed deeper into our problem, we had to extend the tools with a set of functionalities that address our specific needs. The simulator possesses a modular structure, which eases the insertion of new functionalities through integration of custom modules or even through the modification of existing ones, minimizing the necessity of tampering with the core of the program.

By default, the results provided by the simulation tool were presented in a text format. On the end of a successful simulation, a summary with the results from the data flow graph behaviour is shown in the console while a detailed list with the task executions characteristics was stored on a text file. This type of visualization was practical for some cases but created some confusion in others. For example, it is difficult to identify executions that overlap in time just by analysing a list of its start and finish times. So, in order to make the simulation results more readable, we decided to integrate an external visualization tool that allowed the timing behaviour of the tasks simulated to be presented in a coloured Gantt chart form.

In this chapter we explain in detail all the functionalities and changes mentioned so far.

3.1 The Heracles data flow simulator

3.1.1 Heracles tool flow

The Heracles simulator was written using **Objective Caml** (Objective Categorical Abstract Machine Language). OCaml is a dialect of the ML (*Meta-Language*) family of languages, which derive from the Classic ML language designed by Robin Milner in 1975 for the LCF (*Logic of Computable Functions*) theorem prover [14].

OCaml shares many features with other dialects of ML, and it provides several new features of its own. The main characteristics of this language are as follows:

- It is a **functional** language, meaning that functions are trated as first-class values.
- It is **strongly typed**, meaning that the type of every variable and every expression in a program is determined at compile time.

- Related to strong typing, OCaml uses **type inference** to infer types for the expressions in a program.
- The type system is **polymorphic**, meaning that it is possible to write programs that work for values of any type.

Although ML languages are mostly functional, they also include some imperative traits, which allows that a program written in OCaml can evidence both type of programming traits. The Heracles simulator was written using this paradigm. For more information regarding this programming language, please refer to [17], [7] and [22].

OCaml offers some distinct advantages when compared to other imperative languages, as for example C or Java. One of the most attractive features of this language is the **type inference** preformed by the compiler. This feature allows a user to write programs without explicitly indicate the type of data (integer, float, string etc.) of its defined variables: the compiler infers them by observing the context where the variable is inserted. This characteristic allows the compiler to detect and identify a great number of *bugs* that otherwise would be difficult to discover. This happens because the compiler, as it infers the type of a certain variable, will also check the consistency of the rest of the function regarding all the operations that this variable is included. This way, only programs that are able to maintain data type coherence throughout all the operations are able to compile successfully.

Another advantage of OCaml is the abstraction of pointers. While this feature is source for many troublesome bugs in languages as C or C++, in OCaml, the compiler handles all these references. The user is not allowed to change the intrinsic value of these references: only the variables that they point to. From a user standpoint, the syntax used for defining and changing memory positions when referred through a pointer (in OCaml we use the **ref** operator to define a reference or pointer) are relatively easier to manipulate than its imperative counterpart.

Finally, one of the most distinct characteristics of OCaml is the code compression. With this language is possible to write complex applications using a third of the code lines that the same program would need when written on a fully imperative language. Due to type inference and the functional nature of this language, a function of medium complexity can sometimes be written in a single code line.

3.1.2 Explanation of the software model

The Heracles simulation module was developed with the purpose of providing accurate timing simulations. The usage of this tool is mainly text based: the system to be simulated is inserted into the tool via a text file containing a description of the graph to be simulated, namely, a list with all the actors and their respective execution times, as some other relevant characteristics, as for example, the mapped processor, associated priority, etc, and a list of all the edges with the sources and sinks properly identified, the initial tokens, production and consumption rates. This text file is then parsed so that all the components and main characteristics mentioned before can be extracted into internal variables.

The simulator operates on a set of objects of a record type (commonly known also as *structure* or *class* in other programming languages) called *Event*. Along with plenty of useful information, an event also represents the start or finish of a scheduled task. These events are aggregated in a set, ordered through a specific function. This function, called *compare*, is particularly important

since the order in which the elements are put into the set is also the order in which they are processed by the simulator.

This function is essential to understand the basic mechanics under the simulation tool so we are going to explain it in detail. The *compare* function operates with *Event* structures. This structure is characterized by the following elements:

- **Event id** - identifies the type of event, namely if it is a *Start* or a *Finish* type of event. The *Finish* events should always precede *Start* events.
- **Start time** - This field indicates the time at which the *Event* should be processed. It contains a time value relative to the simulation clock.
- **Priority** - Although this field was already defined in the initial version of the code, it was being used in a limited way. As the name implies, it relates to the relative priority of the *Event*. We decided to retain the convention used in real-time that was described in 1.4, i.e, lower priority values mean higher relative priority.
- **Issue** - The issue of an *Event* is a unique identifier for that element. It has no operational significance other than uniquely identifying the *Events* in each simulation.
- **Multiplicity** - This field contains the number of simultaneous firings of a given *Event*. In *data flow*, when an actor has its firing conditions met, it can fire as many times as the number of tokens in its input edges permit. This field was created so that in a situation like this, instead of creating multiple copies of the same *Event*, the simulator creates only one *Event* with a multiplicity value equal to the number of simultaneously firings of the actor.

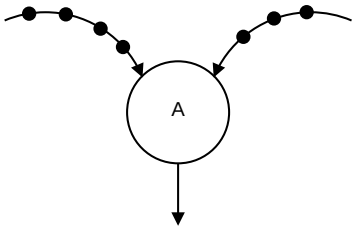

Figure 3.1: Example of an actor able to fire multiple times simultaneously

In figure 3.1, actor A has its firing rules met for three firings, so it will do it simultaneously (the actor is able to process three independent set of input tokens). If all the actor firings have the same execution time, in the simulation environment only one *Event* will be created regarding this firing. The multiplicity field in this *Event* will be equal to the minimum number of tokens in all of its input edges, i.e, 3 in this example.

- **Internal actor** - this last field contains an actor structure with the information relative to the actor that issued the event. In example 3.1, the internal actor field will contain a copy of the structure of actor A.

The *compare* function operates on the Event id, Start time, Priority and Issue fields of an *Event*. Whenever a new *Event* is to be inserted into the existing set, the *compare* function sequentially compares this event with all the events already inserted in the set, until it finds a suitable position for it.

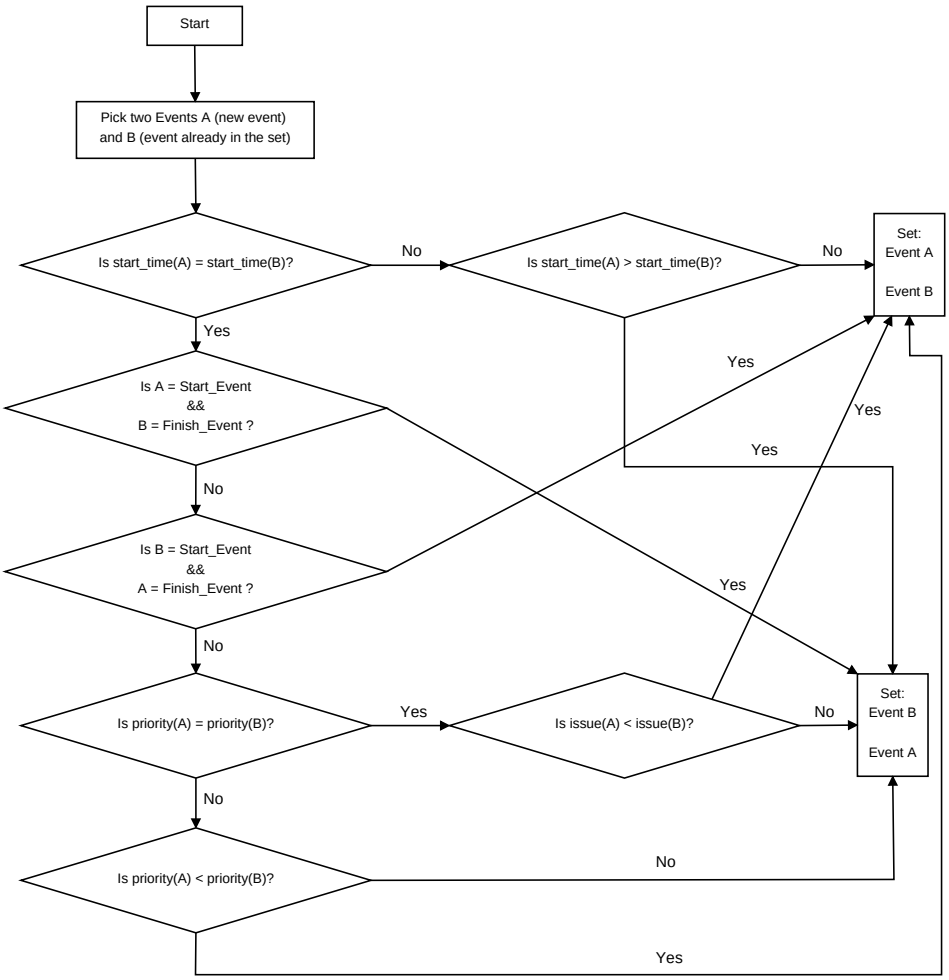

Figure 3.2: Flowchart of the *compare* function

Figure 3.2 depicts a schematic of the operation of the *compare* function elaborated for our project.

A poorly conceived *compare* function can easily originate *deadlock* situations. Lets consider for instance the decision regarding the Event ids. As it is defined, this function give precedence to the *Finish Events* relative to the *Start Events*. If several *Finish Events* are already on top of the set, they will be ordered according to the remaining parameters. This way, we ensure that the end of an execution always precedes the start of another. This is important because *Finish Events* release resources while *Start Events* reserve them. If this order was inverted, the simulation would reach a *deadlock* at some point, since the *Events* that reserve resources are being resolved before the *Events* that release resources. An imbalance between resource release and reserve is created and it is a matter of time until there are no resources available. Since the Event id comparison is made in the top, an Event that cannot be resolved is simply reinserted into the set in the same position as before.

At the beginning of an iteration, the simulator picks up the event on top of the set and operates based on it. A simplified scheme of the functioning of the simulator is presented in figure 3.3.

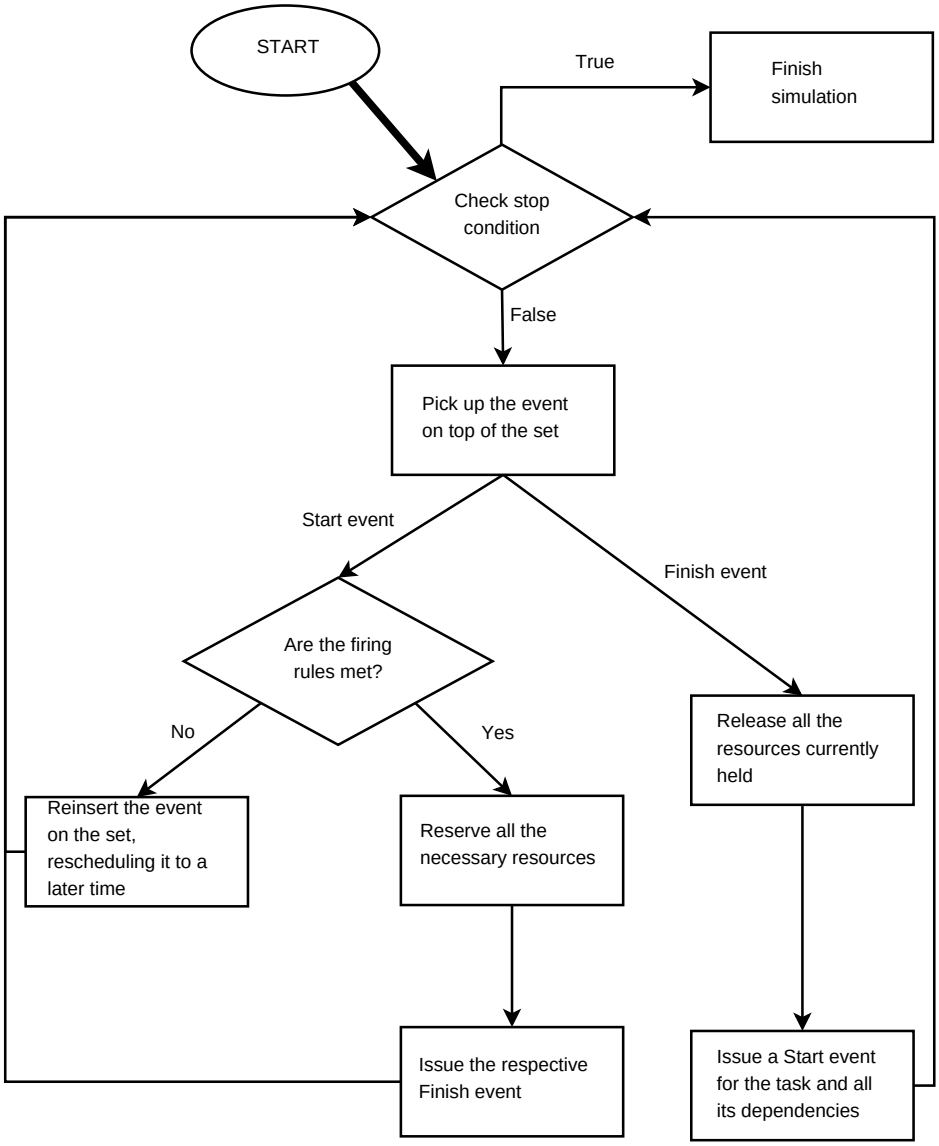

Figure 3.3: Flowchart for the simulator

This behaviour is assured by several functions and some complex data structures. All the extra functions or changes to the data structures that we intend to make will not alter the structure shown in figure 3.3.

In the remainder of this chapter, we are not going to make reference to any of the code changes made in the course of this project. These alterations were made to solve specific problems and we lack the context to explain them properly. Hence, they will be addressed in future chapters, when the problem that they were design to deal with is discussed.

3.1.3 Usage of the tool

The Heracles simulator is a text based application with its own parser and lexer for input of data.

In order to simulate a given system, one has to input it into the simulator in the form of a data flow graph. In Heracles, this is accomplished through the indication, via the command line, of a text file with all the data regarding the system to be simulated. This text file must obey the set of rules defined by the parser regarding its structure and the type of data admissible, as it was referred in section 3.1.2.

All other options of the simulator are passed to it through the command line, upon the calling of the main application. Whenever we added a new functionality to the system, the module that dealt with the available option had to be altered so that the new functionalities could be used, also in a modular and independent way.

If the data is inserted correctly, the simulator will run the graph until a stopping condition is achieved. This stopping condition can be the identification of a periodic behaviour or simply a predefined elapsed time. Once the stopping condition is reached, a summary of the graph execution is displayed in a console. This summary is comprised of information such as the total execution time, total number of steps taken, maximum number of tokens that each edge had during the graph execution, etc. Along with this summary, a text file containing a list with all the executions of all actor in the graph is created. In it we can consult information such as the relative times that each actor started and finished a certain execution.

After the integration of the visualization tool mentioned before, a successful simulation will also produce the script files to be interpreted by it. These files contain the same information of the previous file but formatted in way that the visualization can read and use to produce the respective Gantt charts.

3.2 Conclusion

In this chapter we intended to offer a concise explanation of the set of tools that we used during this project. The great majority of the results presented along this document were obtained through this tool, either by data flow simulation or by usage of other related functions. Given the importance of this software in the course of our work, it is important that we introduce to the reader the key functionalities of such a useful instrument.

Chapter 4

Implementations introduced in the software framework

4.1 Main changes to the code

During the remainder of this document, we will make reference to the various changes and additions to the Heracles source code. These references will appear in the appropriate context, so for now we are only going to indicate the alterations made to the general structure of the tool set, namely the inclusion of the visualization tool and other related alterations.

1. **Implement the fixed priority simulation option** - This project is centred around fixed priority schedulers. As so, one of the first and most important modifications that we implemented was the inclusion of a fixed priority data flow simulation. For it, we used the priority element already defined in the internal representation of the actors of the graph and created the necessary structures to correctly implement this feature. This functionality is activated by passing the respective option through the command line.

2. **Create the functions to build a script file from the task activations to be interpreted by the visualization tool** - During the course of our work, we realized that a tool that would allow us to see a graphical representation of the results of the simulation, such as a Gantt chart for instance, could be very useful, specially to detect patterns, like periodic behaviours. After some research we discovered the TimeDoctor tool. TimeDoctor is an open source application that it is primarily used to visualize the execution traces of tasks, queues, cache behaviour, etc. in embedded media processors [31] but it is flexible enough to be adapted to our needs. The usage of this tool is simple: the behaviour of the tasks, i.e, their start and finish times, mapped processor, etc, is inserted into a text file with a .tdi extension. A quick glance into one of these script files was enough to realize that the building process of one of these script files can be easily automatized through file writing functions. Not only this tool is practical for creating Gantt charts with the tasks executions but it also can be used to represent the state of the edges throughout time, specifically, the number of data tokens in each edge of the graph during the length of the simulation.

   One of the files created in each simulation will represent a Gantt chart of all the executions. Each task in this file is properly identified with its name and the processor in which it was

mapped along with a chart with the evolution of the number of data tokens in the edges of the graph.

A second file will do also a timing representation but now from the processor's point of view. This is a processor utilization chart and it represents all the activations that occur in a given processor over the simulation time. This is useful for studying the computational load of a given processor.

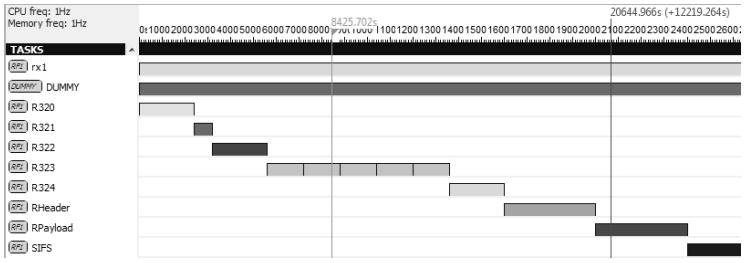

Figure 4.1: Example of simulation results interpreted by the visualization tool as a Gantt chart

3. **Elaborate the necessary functions to represent the state of the graph edges in time** - In addition to the mapping of the executions, we considered that a representation of the state of the graph edges will also be desirable. Fortunately, the TimeDoctor tool has a built in functionality for representation of queues, which was perfect for our situation. By adding the necessary functions to the code, along with a Gantt chart of the graph executions, we were also able to represent the state of the interconnecting FIFOs during the simulation. Figure 4.2 depicts the result obtained.

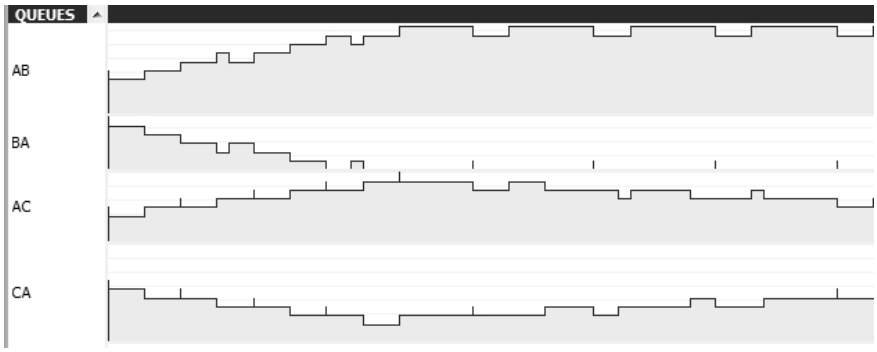

Figure 4.2: Example of the evolution of a graph edges interpreted by the visualization tool

4. **Create a set of functions for representation of the load curve of a processor** - This subject will be addressed in great detail in chapter 6, but in the meantime, we must make a reference to the load curve of a processor. This curve represents the evolution of the computational load on a processor and allow us to perceive how the work load is distributed throughout the simulation time. To build it, we need a time vector and a usage vector with a one to one correspondence in order to be able to do a trace. The output of the data should be on a simple text file, formatted in a way that can be interpreted by a chart building application, as for instance, Matlab or GNU Octave.

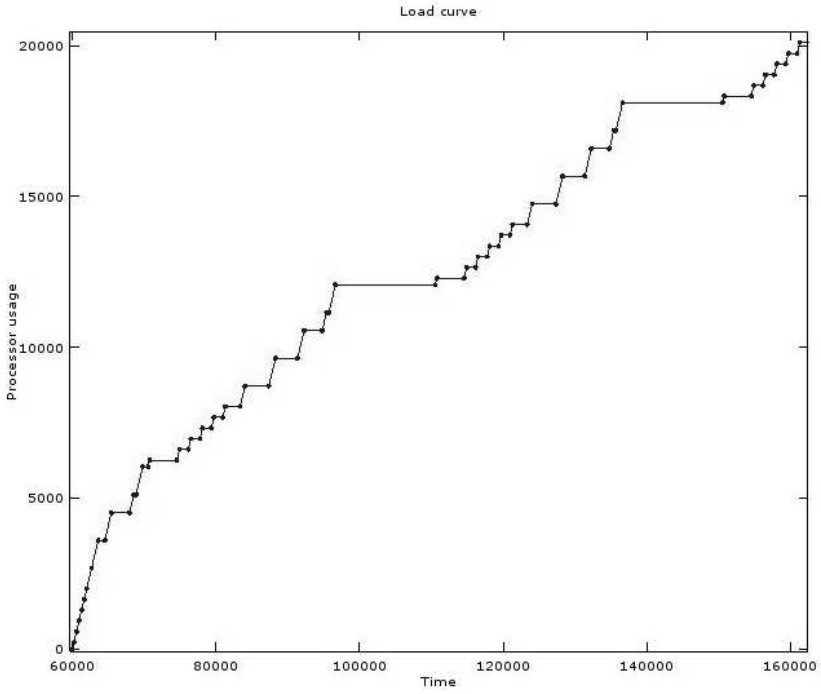

Figure 4.3: Example of a load curve plotted using GNU Octave

5. **Create a set of functions to compute the interference between tasks** - During this section of our project, we needed to perform calculations regarding the computation of the worst-case execution time of a low priority task when subjected to the interference from a high priority task mapped in the same processor. In order to achieve satisfactory results in this subject, it was necessary to take into account the preemption that the high priority task would impose on the running low priority task. Since the data flow simulator does not contemplate the use of pre-emption, we needed to develop a set of function that emulate this behaviour. Basically, our functions received a list with all the executions of both concurrent tasks in the same time referential and performed a merge of these lists taking into account the relative priority of the tasks and including preemptive effects when necessary. Along with the return of a merged list, these functions also compute and return the worst-case execution time of the low priority task.

6. **Insert an option to include the context switching times into the simulation** - In order to approach our simulations as close as possible to real cases, we need to take a critical factor into account: the context switch performed by the processor whenever a low priority task is preempted by a higher priority task. When a task is preempted, the processor need to store the task state and all the information that could be necessary to resume this task somewhere in the future. Also, the processor need to prepare the context for executing a different task. All these operations take a significant amount of time and, in order to make our simulations as realistic as possible, we need to take it into consideration. As so, we created a set of functions that insert a context switch task whenever they detect a preemption along a list of executions of several tasks. The activation of this functionality

and the definition of the context switch delay are passed as arguments via the command line.

All of the modifications referred above were inserted in a proper software module to avoid unnecessary additions to an already complex source code. Only the relevant functions and structures are exportable in order to maintain simplicity.

4.2 Conclusion

In this chapter we described the alterations that we performed to the software platform so that we can use the tools to verify and retrieve information from the simulation of data flow graph, using fixed priority or not. The actual implementation of the changes referred in this chapter was left out on purpose. In later chapters, when the correct context is build, we will make a more detailed explanation of the respective software changes.

Initially, all changes in the code were performed by editing existing sources. In order to maintain code integrity, all the additions were protected inside conditional branches, i.e, a new functionality can only be evoked if the respective option is indicated in the command line. On a later phase of development, we decided that it was more practical and simple to create our own software module and do the necessary imports into to the main source code.

Chapter 5

Intra-Graph fixed priority analysis for data-flow graphs

The pre-emptive fixed-priority schedulers are a popular real-time scheduler. The simplicity of its implementation can hide the difficulties in the study of its behaviour, particularly regarding the determination of best and worst-case start times of the tasks associated in a data-flow analysis.

It is possible to imagine a situation where a high priority actor is simply activated at a rate that causes starvation among the lower priority ones, even though this type of problem can also be observed in the absence of preemption - a high priority actor can simply be continuously scheduled ahead of a low priority one. Along with starvation, we can also point out to backlogging situations: whenever a high priority task is dependent of a product of a low priority one (processing data for instance), the rate of execution of the first one is undoubtedly limited by the rate of execution of the late.

These are some of the problems encountered during our analysis and will be properly addressed in the remainder of the chapter.

5.1 Problem definition

Given a set of actors, with different priorities, to be scheduled for execution, we want to establish a model based on data-flow that can conservatively predict the worst-case temporal behaviour for each actor. We want to determine what could be the largest response-time per firing, which in this case corresponds to the elapsed time between the moment in which an actor is set to start (by the scheduler for example) and its actual finishing time.

We will start by establishing a simple model with two actors with different priorities, that are interdependent from each other. This means that even though the high priority actor can preempt the low priority one, eventually the system will reach a point where the high priority actor is unable to fire because it needs the other to produce at least one token into its incoming edge.

Before we dwell into this problem, it is important to define the concepts that are used in the remainder of this chapter:

Definition 5.1. ***Execution time*** *τ_i of an actor a_i is a temporal defining parameter. τ_i is the amount of time required to complete the execution of a_i when it executes alone and has all the*

resources it requires. Hence, the value of this parameter depends mainly on the complexity of the execution and the speed of the processor used to execute it and not on how it is scheduled.

The actual amount of time required by an actor to complete its execution may vary for many reasons. As examples, the computation may contain conditional branches, and these conditional branches may take different amounts of time to complete. The branches taken during the execution depend on the input data. If the underlying system has performance enhancing feature (e.g., cache memory and pipeline), the amount of time a computation takes to complete may vary each time it executes even when it has no conditional branches. For these reasons, the actual execution time is unknown until it completes [25].

Definition 5.2. *In real-time systems, the* ***response-time*** *r of an actor is defined as the time elapsed between the release (instant of time when the actor becomes ready to execute) to the time when it finishes the execution (one dispatch). [6].*

response-time is different from **worst-case Execution Time**, which is the maximum time the actor would take if it were to execute without interference. It is also different from deadline, which is the length of time during which the actor's output would be valid in the context of a specific system.

For now, the analysis will be focused on this simple graph setup. After we find a satisfactory model for it, we will increase the complexity of the system.

5.2 Theory

In the next sections we are going to explore this problem by analysing some fixed priority arrangements using the concepts of data-flow analysis introduced in chapter 2. The purpose of this analysis is to illustrate some of the problems that appear when one tackles such subjects.

5.2.1 Data-Flow analysis of a fixed priority system

Lets start by considering a simple system composed by two actors, a high priority actor A and a low priority actor B, with dependences from each other. A and B are executing on the same processor. The data flow representation of this system is shown in figure 5.1.

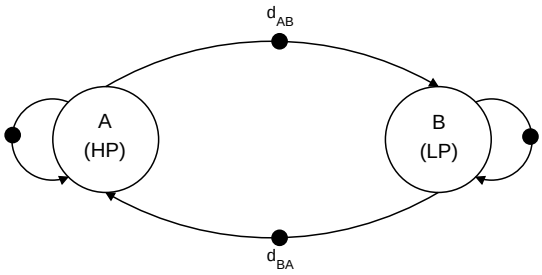

Figure 5.1: Simple fixed priority arrangement

In figure 5.1, d_{AB} represents the initial number of tokens present on the AB edge, while d_{BA} is the number of tokens on the opposite edge, with $d_{AB}, d_{BA} \in \mathbb{N}_0$. Both actors have self edges with a single initial token, which limits to one the number of firings that each actor can execute at a given moment.

The execution time of each actor is τ_A and τ_B. We also assume that after a token is produced by an actor, it can be immediately consumed by the dependent actor. This means that we consider that a zero time permanence in a intermediate buffer or FIFO.

We also assume that the processor allows preemption, which mean that actor A will run whenever its firing conditions are met.

5.2.1.1 Notation

We will be dealing with the **running time** of the various actors in study and we will use the letter r to indicate such quantity.

All other representations used respect the notation adopted in other relative publications and the notation introduced in chapter 2. We will use the first letter of the quantity name followed by the actors associated underlined. For example, a number of delays are indicated by a d_{edge}, the starting time of a given actor in a particular iteration is denoted by $s(i,k)$ and the respective finishing time respects the same notation, but using $f(i,k)$.

Also, in this chapter we will use the terms **actor instance** and **actor firing** interchangeably. The first term is mostly used in classical real-time theory while the second one is the preferred designation in data flow analysis.

5.2.1.2 Worst-case response-time for the first firing

For all the situations to be analysed in this case, its is important to point out that we assume that the system is started in $t = 0$ in order to avoid the starting time term, which will be common to both worst-case analysis. We assume that the system is in stand by, with all the respective data tokens steady on the nodes, and then it is turned on at time 0, with all the actors firing immediately, if able to do so.

5.2.1.2.1 High priority actor :

Since the high priority actor has the capacity to pre-empt all the low priority ones, whenever it is ready to run, it simply fires and executes without being preempted.

$$\hat{r}(A,0) = \tau_A \tag{5.1}$$

If we only take into consideration the cases where the actors are all able to run, then the one with higher priority need only to wait for its previous firing.

5.2.1.2.2 Low priority actor :

The analysis for the low priority actor is more complex since we must take into account the effects of preemption by the high priority actor. So, in a given moment, if we have both actors with enough tokens in its inputs to fire, actor B will always be blocked by the high priority actor A. The question is, how long does that block last?

Actor A will block actor B as long as it has tokens to activate itself. Eventually actor A will consume all available tokens and only then will actor B be allowed to run. So, considering the logic taken in equation 5.1, we still need to take into account the execution time of the actor, thus

$$\hat{r}(B,0) = \tau_A \cdot d_{BA} + \tau_B \tag{5.2}$$

Although, this is not the worst-case conceivable. If we look at figure 5.1, we can make an assumption in which actor B was activated for $d_{AB} - 1$ times before. This means that we can project a scenario where all but one of the initial tokens on the AB edge are now in the opposing edge. We maintain a single token on the AB edge in order to comply with the assumption made in the beginning of section 5.2.1.2, i.e, all actors have their firing conditions met at $t = 0$. Lets also assume that the previous happened because actor A was unavailable before but becomes active in this specific moment. Now actor B has to wait for actor A to consume all those accumulated tokens. Taking this into consideration, the worst-case response time for actor B will be

$$\hat{r}(B, k) = (d_{AB} + d_{BA} - 1)\tau_A + \tau_B \tag{5.3}$$

This is a conservative scenario. It is useful to define an upper bound for our analysis, although it is impracticable in this example.

5.2.1.3 Dynamic data-flow analysis

We are now ready to make a more realistic analysis of the system. We intend to determine what happens to each actor after the initial activation of the system showed on figure 5.1 regarding their response times.

5.2.1.3.1 High priority actor :

We begin to assume again that both actors are scheduled to execution at $t = 0$, with actor A being the first to execute, blocking B in the process. After A as consumed all the tokens in the BA edge, it will enter a new regime where it needs to wait for B to produce a token into the BA edge. Since we consider the response time as the time elapsed **since an actor has enough tokens to execute** to the time where it actually **finishes its execution**, actor A only takes τ_A time to execute always. For the k^{th} iteration, the running time of actor A is

$$r(A, k) = \tau_A, \quad k \in \mathbb{N}_0 \tag{5.4}$$

5.2.1.3.2 Low priority actor :

As for actor B, we can identify two distinct running times for it. In the beginning we have B blocked by A during exactly d_{BA} iterations, waiting τ_A time each. So actor B can only make its first execution after A has consumed all the tokens in the BA edge and after that we still have to wait for B to finish executing

$$r(B, 0) = d_{BA} \cdot \tau_A + \tau_B \tag{5.5}$$

But after that first execution of B, the system enters a state in which both running times stay constant. In this state we have actor B producing a token into the BA edge, followed by the consumption of a token by the actor A, and thus blocking the execution of actor B, and finally followed by the next execution of B, due to the fact that only one token exists at a time in edge BA, blocking actor A after every execution. But since B is ready for execution during the block from A, we consider that whole time. The running time of actor B for the k^{th} iteration after the first becomes:

$$r(B, k) = \tau_A + \tau_B, \quad k \in \mathbb{N} \tag{5.6}$$

Rewriting this into a more general expression:

$$r(B,k) = \begin{cases} d_{BA} \cdot \tau_A + \tau_B & \text{if } k = 0 \\ \tau_A + \tau_B & \text{if } k > 0 \end{cases}, \quad k \in \mathbb{N}_0 \tag{5.7}$$

5.2.1.4 Analysis of the actor starting times

We will now analyse the evolution of the starting times of each actor.

5.2.1.4.1 High priority actor :

Since it has the highest priority, the first execution of actor A is done in $t = 0$

$$s(A,0) = 0 \tag{5.8}$$

From then on, as long as A has tokens on the BA edge, it will be fired as soon as it finishes its last execution. So

$$\begin{aligned} s(A,1) &= s(A,0) + \tau_A \\ &= \tau_A \\ \\ s(A,2) &= s(A,1) + \tau_A \\ &= s(A,0) + 2\tau_A \\ &= 2\tau_A \\ &\vdots \\ s(A,k) &= s(A,k-1) + \tau_A \\ &= k \cdot \tau_A, \quad \text{if } k < d_{BA} \end{aligned} \tag{5.9}$$

After all the BA edge tokens are consumed, actor A must wait for at least one of them to be produced by actor B. But from this point on forward, we need to take into account the time spend executing B because now we are trying to define a relative time span and, relatively to the last execution of actor A, it had to wait for that τ_B interval.

So now we have that:

$$s(A,k) = s(A,k-1) + \tau_A + \tau_B \quad \text{if } k \geq d_{BA} \tag{5.10}$$

We can develop the previous equation further, specifically the $s(A,k-1)$ term. Lets consider the situation in which actor A as just finished the execution resultant of the consumption of the last token remaining on the BA edge. actor A is on the verge of waiting for a production from B for the first time. In that particular situation we have that $s(A,k-1) = (d_{BA}-1)\cdot\tau_A = s(A,d_{BA}-1)$, for $k = d_{BA}$. The -1 term is due to the fact that our initial k term is 0, i.e, our fist iteration occurs when $k = 0$.

We can now iterate from there:

$$\begin{aligned} s(A,d_{BA}+0) &= (d_{BA}-1)\cdot\tau_A + \tau_A + \tau_B \\ &= d_{BA}\cdot\tau_A + \tau_B \end{aligned} \tag{5.11}$$

In the previous expression, the term $(d_{BA}-1)\cdot\tau_A$ takes into account the time spent by actor A processing the initial tokens in its incoming edge. The term $\tau_A+\tau_B$ refers to the one cycle after actor A has finished this consumption. For the next iteration, the same logic is maintained: the start time for any iteration after the consumption of all the initial tokens from the incoming edge of the high priority actor is given by adding the time this actor takes to consume those tokens (which would be a fixed value) to the time spend in the iterations where both actors fire in alternatively (which depends on the number of iterations considered).

$$\begin{aligned} s(A, d_{BA}+1) &= (d_{BA}-1)\cdot\tau_A + 2(\tau_A+\tau_B) \\ &= d_{BA}\cdot\tau_A + \tau_A + 2\cdot\tau_B \\ &= (d_{BA}+1)\cdot\tau_A + 2\cdot\tau_B \\ &\vdots \\ s(A, d_{BA}+k) &= (d_{BA}-1)\cdot\tau_A + (\tau_A+\tau_B)\cdot(k+1) \\ &= (d_{BA}+k)\cdot\tau_A + (k+1)\cdot\tau_B \end{aligned} \tag{5.12}$$

But we want to maintain the notation used previously, so we rewrite the last expression by subtracting the d_{BA} term and include equation 5.9

$$s(A,k) = \begin{cases} k\cdot\tau_A & \text{if } k < d_{BA} \\ k\cdot\tau_A + (k-d_{BA}+1)\tau_B & \text{if } k \geq d_{BA} \end{cases} \tag{5.13}$$

5.2.1.4.2 Low priority actor :

The low priority actor seems to be a simpler because it is blocked for a fixed time in the beginning. But once it start firing, it is capable of maintaining some periodicity. Due to the initial activations of actor A, the start time of the first execution of actor B will be

$$s(B,0) = d_{BA}\cdot\tau_A \tag{5.14}$$

But after the first execution, actor B will have the same ratio of execution of actor A, as we saw on the previous section. So

$$\begin{aligned} s(B,1) &= s(B,0) + \tau_A + \tau_B \\ &= d_{BA}\cdot\tau_A + \tau_A + \tau_B \\ &= (d_{BA}+1)\tau_A + \tau_B \end{aligned}$$

$$\begin{aligned} s(B,2) &= s(B,1) + \tau_A + \tau_B \\ &= (d_{BA}+1)\tau_A + \tau_B + \tau_A + \tau_B \\ &= (d_{BA}+2)\tau_A + 2\tau_B \\ &\vdots \\ s(B,k) &= (d_{BA}+k)\tau_A + \tau_B\cdot k, \quad k\in\mathbb{N}_0 \end{aligned} \tag{5.15}$$

5.3 Multi processor mapping analysis

5.3.1 Overview

We are now in the position to increment the complexity of our previous graph.

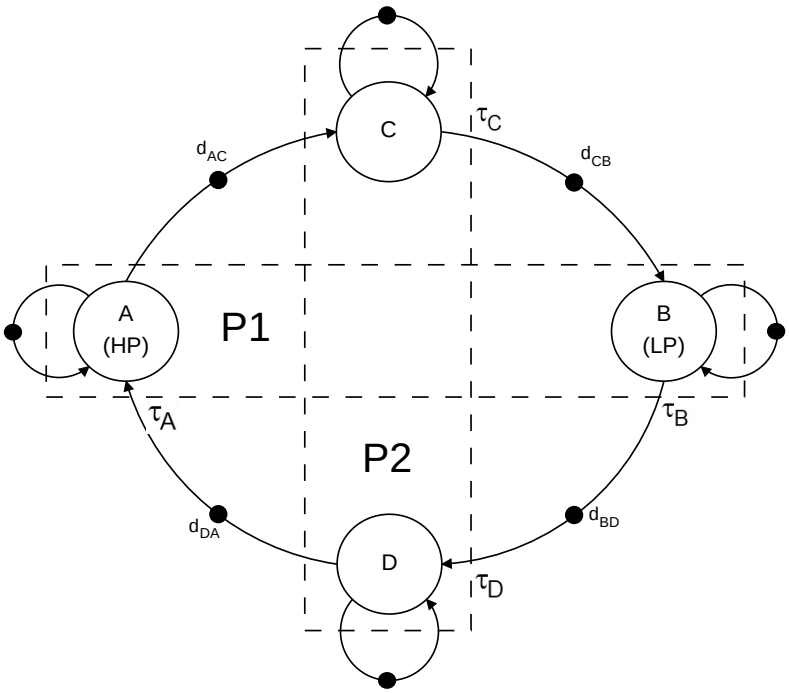

Figure 5.2: Example of a 4 actor graph with different processor dependencies and where $P_A > P_B$

In this situation, we inserted two additional actors to our default system. The priorities were maintained, but is important to point out that actors C and D are mapped on different processors as the remaining actors A and B. Because of that, we assume that the execution of actors C and D can occur simultaneously with the execution of other actors and they cannot suffer preemption from actor A or preempt actor B.

As before, each of the edges that interconnects the actors has an initial finite number of tokens, represented by the same notation used before, and each actor has a self edge with an initial data token. The execution time of an actor is characterized by a time τ_{actor} as usual.

Since actors C and D are not mapped on the same processor, we are only extending our analysis to actors A and B and only for the determination of the worst-case response time and starting times. Also, from this point on forward, actor C will be ignored for the moment, since we can see that actor B will originate a concentration of tokens in its input edges. Since we are interested in what happens until the first activation of the lower priority actor, we decided to ignore the "inactive" actor, for simplicity

5.3.2 Worst-case response time

5.3.2.1 High priority actor

We are going to maintain the same criteria of analysis used before. Assuming that $d_{DA} > 0$, for the first activation of high priority actor A we have that:

$$\hat{r}(A, 0) = \tau_A \tag{5.16}$$

Since actor A as priority over actor B and can execute simultaneously with actor C and D.

5.3.2.2 Low priority actor

In this section we are going to determine a conservative upper bound for the response time of actor B. By looking at figure 5.2, we can start by inferring that actor B has to wait for actor A to consume all the tokens in the DA edge, at least. But we also need to consider the tokens in the BD edge. In a worst-case scenario, we can have $\tau_A > \tau_D$ and $d_{DA} > d_{BD}$, which means that while actor A is consuming the tokens on the DA edge, actor D is consuming tokens from the BD edge and putting the processed tokens into the DA edge at a higher rate than the one which actor A can consume them. That will maximize the waiting period for actor B since the number of tokens consumable by the high priority actor is also maximum. So, for the first execution of actor B we have that:

$$\hat{r}(B, 0) = (d_{DA} + d_{BD}) \cdot \tau_A + \tau_B \tag{5.17}$$

This is a pessimistic approach. By inferring that $\tau_A < \tau_D$, it is easy to conceive some realistic scenarios in which actor B is executed faster.

5.3.3 Analysis of start times

5.3.3.1 High priority actor

An initial analysis of this system revealed two different scenarios regarding the relation between the response times of the high priority actor A and the actor that precedes it creating a direct dependence, actor D. So we need to fork our study one more time to analyse each scenario resulting from each assumption regarding the response times of the actors.

5.3.3.1.1 $\tau_A > \tau_D$:

For this case, we can simplify our study by assuming that all of the firings of actor A are consecutive, i.e, actor A will never be blocked by lack of input tokens in the DA edge. This general assumption is done by realizing that actor A depends on actor D and since this last one is faster to execute than the later, there will be more tokens produced than consumed in the DA edge. The only way to have actor A blocked in this case is to have actor D deactivated before by exhaustion of all the tokens available in the BD edge. So, the question here is, **in the situation depicted in figure 5.2, if $\tau_A \geq \tau_D$, is there any situation where, regarding the number of tokens in incoming edges of each actor in study, actor A could be blocked before actor D consumed all of its input data tokens?**

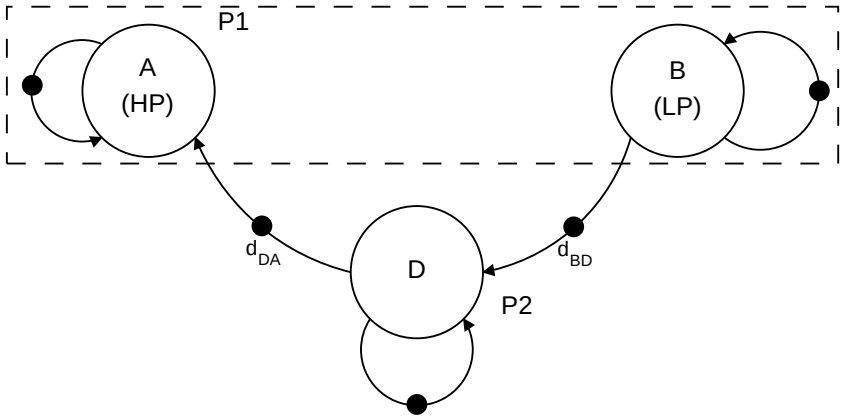

Figure 5.3: Representation of the system in study

For the system depicted in figure 5.3, lets start by consider that:

1. τ_A is a rational number greater than 0.
2. $\tau_D = \tau_A - \delta t, \delta t$ an infinitesimal time interval.
3. $d_{DA} = 1$, we start with just the minimum number of tokens possible in this edge.
4. $d_{BD} = \infty$.

Given all these assumptions, one can see that, if actor D and A are fired at the same time, at $t = 0$, because $\tau_A > \tau_D$ (δt is infinitely small but is still greater than 0), actor D will always finish δt sooner than actor A and, because of that, it puts a data token into the incoming edge of actor A. δt time after, actor A finishes it initial execution but finds an available token already deployed in its incoming edge, and so it initiates another execution right after the first one.

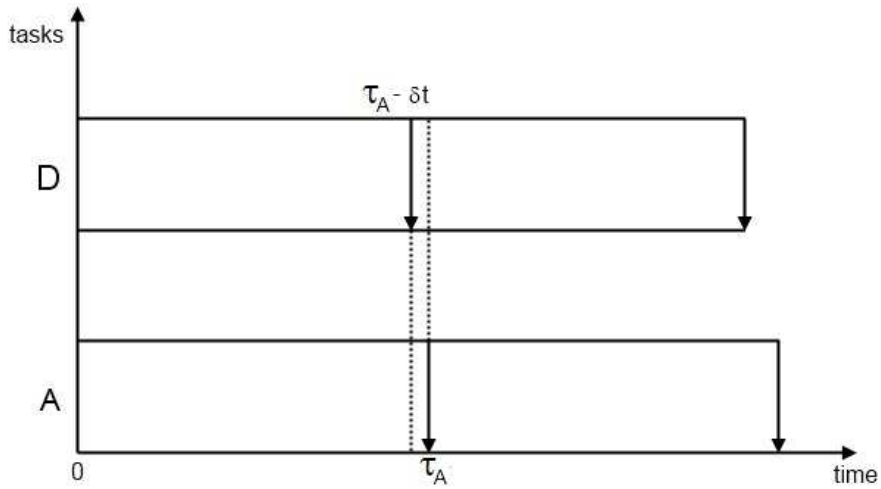

Figure 5.4: Activation diagram of both actors in study

But we cannot forget that actor D also finished its first execution δt sooner than actor A. So it also started its second execution δt time before actor A second execution start.

So actor D will finish its second execution $2 \cdot \delta t$ before the second finish of actor A and so on. As so

$$\begin{aligned}
s(D,0) &= 0 \\
\\
s(D,1) &= s(D,0) + \tau_D \\
&= \tau_A - \delta t \\
\\
s(D,2) &= s(D,1) + \tau_D \\
&= \tau_A - \delta t + \tau_A - \delta t \\
&= 2 \cdot (\tau_A - \delta t) \\
\\
&\vdots \\
s(D,n) &= s(D,n-1) + \tau_D \\
&= (n-1) \cdot (\tau_A - \delta t) + \tau_A - \delta t \\
&= n \cdot \tau_A - n \cdot \delta t, \quad \text{, where } n = \tau_A/\delta t
\end{aligned} \tag{5.18}$$

In the previous equations, n represents the number of delays δt that compose a time interval equal to τ_A.

So, in conclusion, after those n iterations we see that actor A will be delayed a full period of actor D:

$$\begin{aligned}
s(D,n) &= n \cdot \tau_A - n \cdot \delta t \\
&= n \cdot \tau_A - \tau_A \cdot \delta t/\delta t \\
&= n \cdot \tau_A - \tau_A \\
&= (n-1) \cdot \tau_A
\end{aligned} \tag{5.19}$$

This also means that after this period of time, actor A will be facing 2 extra tokens in the DA edge, when it finishes an execution, instead of just one. This allow us to conclude that the number of tokens in the DA edge is growing in time for this situation, which also means that actor A will always execute continuously and so it will never have to wait for actor D to continue processing. As the time progresses to infinite, so will the number of "extra" tokens in the DA edge, which allow us to conclude that actor A will never have to wait for tokens from actor D.

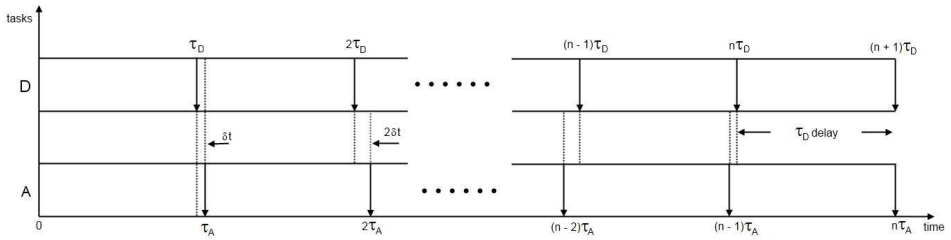

Figure 5.5: Temporal diagram of the system after n iteration of actor A

From the information previous figure, we can write the following expression:

$$s(A,k) = s(D,k) + k \cdot \delta t, \quad k \in \mathbb{N}_0 \tag{5.20}$$

In this case, the k^{th} firing of actor A is preceded by the k^{th} firing of actor D with a $k \cdot \delta t$ difference.

5.3.3.1.2 $\tau_A = \tau_D$:

This case is actually simpler than the last one and can be seen as a particular case of the previous case where we set $\delta t = 0$. If we take all the previous assumptions, then we can see that both actors are synchronized. Both actors finish at the same time, but since we assume that no time is spent in putting or collecting tokens from the input edges, actor A can simply start right after finishing an execution. So, for this case, we adopt the same conclusions as before.

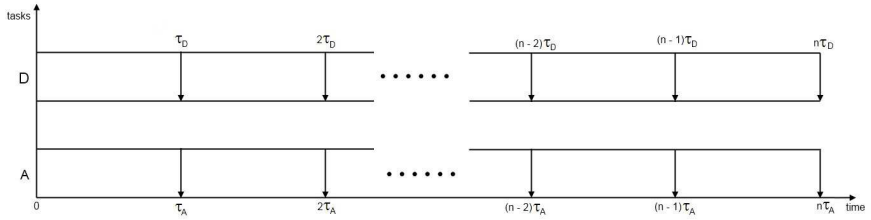

Figure 5.6: Temporal diagram of the response time of actors A and D for $\tau_A = \tau_D$

In this case, n represents any number of iterations.

5.3.3.1.3 $\tau_A < \tau_D$:

Like we did in a previous analysis, we start by determine the time of the first activation and elaborate from there in order to be able to write a general expression. As before, actor A does is first firing right after the start of the system, thus

$$s(A,0) = 0 \tag{5.21}$$

For the next activation, we will continue to assume that there are enough tokens in the DA edge to allow actor A to fire again. That means that

$$\begin{aligned} s(A,1) &= s(A,0) + \tau_A \\ &= \tau_A \\ s(A,2) &= s(A,1) + \tau_A \\ &= 2 \cdot \tau_A \\ &\vdots \\ s(A,k) &= s(A,k-1) + \tau_A \\ &= k \cdot \tau_A \end{aligned} \tag{5.22}$$

After all the tokens in the DA edge are consumed, the next firing of actor A can occur from two situations:

- Actor D finishes the production of a data token into the DA edge, where $0 \leq t_{waitA} \leq \tau_D$, t_{waitA} being the waiting time of actor A due to the absence of tokens in the DA edge. This happens because actor D is being processed in a different processor, which means that when actor A produces its last token, actor D could be in the verge of producing another token, just consumed an input token or be in a situation in between.

- There are no tokens left on the BD edge and actor D is currently inactive. In that case, actor B is put into execution. After it has put the first token into the BD edge, actor D can now fire again, producing a token into the DA edge, which will be finally followed by the execution of actor A.

Putting all this into an expression, we have, for $k \geq d_{DA}$, that

$$s(A,k) = \begin{cases} d_{DA} \cdot \tau_A + t_{waitA} + \tau_A & 0 \leq t_{waitA} \leq \tau_D - \tau_A \\ d_{DA} \cdot \tau_A + \tau_B + \tau_D + \tau_A & \text{if } d_{BD} = 0 \end{cases} \tag{5.23}$$

From this point on we have an increasing difficult in order to determine exact expressions. The term t_{waitA} is particularly complex to determine in a precise way. In order to illustrate the situations indicated above and to clarify the choice of limits for the term t_{waitA}, lets run some examples:

- **Case 1:** Lets consider a system as depicted in figure 5.2 with the following characteristics: $\tau_A = 2$, $\tau_D = 3.1$, $d_{DA} = 2$ and $d_{BD} = 5$.

  A simple analysis using a Gantt chart is shown in figure 5.7

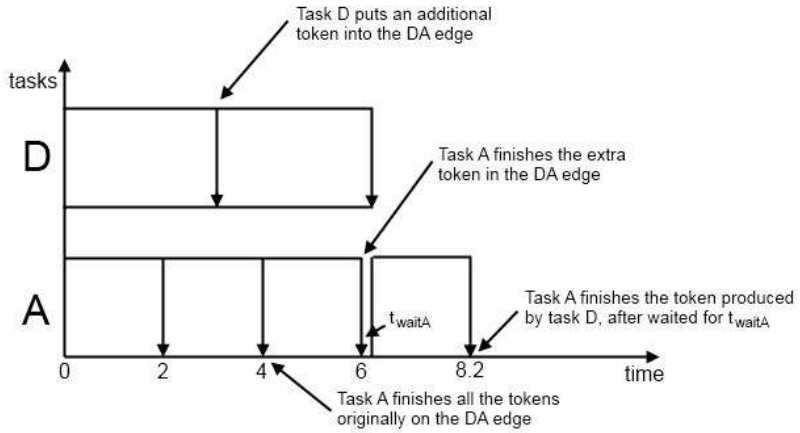

Figure 5.7: Gantt chart of the example used in case 1

  For this case we can explore it further and assume that $\tau_D = 3 + \delta t$, where δt is an infinitely small amount of time, which in this case will result in a waiting period of actor A also infinitely small, which makes $t_{waitA} \to 0$. This should be our best-case scenario for this type of situation.

- **Case 2** Considering the same scheme as before, lets now assume that $\tau_A = 2$, $\tau_D = 5.9$, $d_{DA} = 3$ and $d_{BD} = 5$. In this example we intend to explore the other extreme of the situation depicted in the beginning of this section.

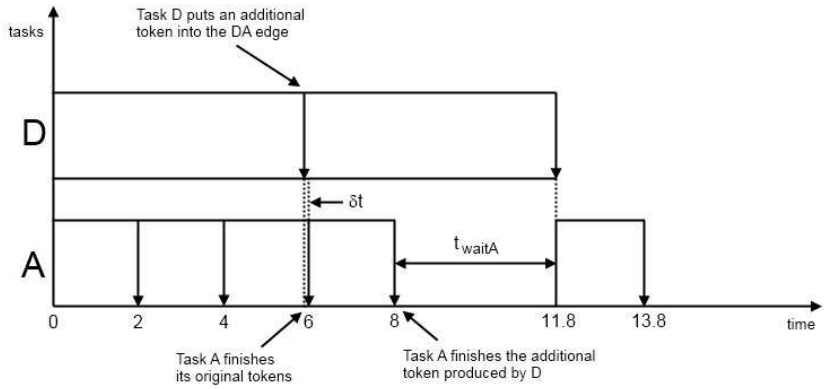

Figure 5.8: Gantt chart of the example used in case 2

As we can see in figure 5.8, we can elaborate a situation where the actor D picks up a data token right before actor A finishes the execution of its last token. But we also can understand that, if we assume that actor D operates continuously as long it has enough tokens for that, a token is put into the DA edge as a result, which means that actor A will always be able to do one last execution before the waiting period, hence explaining the subtraction of the τ_A term in the upper limit of t_{waitA}. As before, we can further elaborate by considering that $\tau_D = 6 - \delta t$ where again δt represents an infinitesimal quantity of time, which will make $t_{waitA} \rightarrow \tau_D - tA$.

Its is important to point out that all of the conclusions reached so far are build upon the assumption that actor D is running continually and in parallel with the actor in question. But in reality we cannot be certain of that, since actor D is assumed to be running on a different processor. Unless it is running solo, it can always be preempted by other actors with higher priority that run on the same platform, which will render our study inconclusive.

Also, we should not forget that every time that actor A is blocked by the absence of tokens in its incoming edge, actor B is immediately put to execution (assuming obviously that it has enough input tokens for it). But since the dependence on the last two cases is focused on actor D, its is not important to make reference to that.

- **Case 3** Consider now that $\tau_A = 2$, $\tau_B = 4$, $\tau_D = 3$, $d_{CB} = 7$, $d_{BD} = 1$ and $d_{DA} = 2$. The corresponding Gantt chart is presented in figure 5.9.

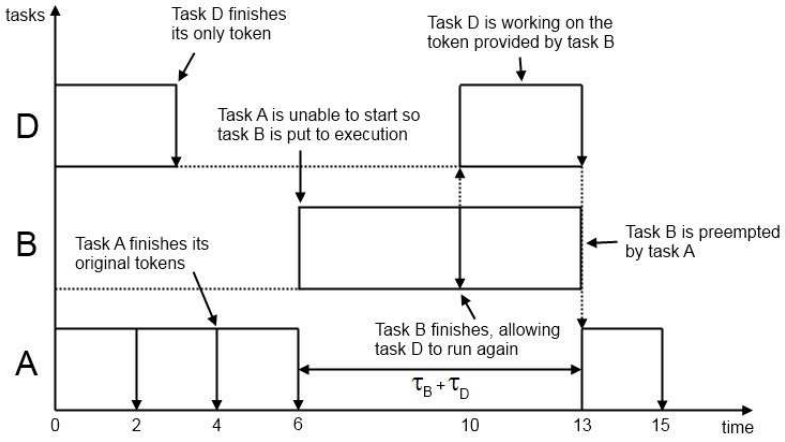

Figure 5.9: Gantt chart for the third case

In this example, we compute a situation were the low priority actor is allowed to execute, at least once, since the further execution of actor A indirectly depends on this one. As we can see, actor A and D exhausts all of its tokens, which allow actor B to finally be able to complete a full execution. After it, actor A still has to wait for actor D to produce a token into the DA edge. In the end, between the starting times of the previous execution and the actual execution, actor A had to wait for $\tau_A + \tau_B + \tau_D$.

All is left now is to analyse these cases and achieve a conclusion to what will be a correct expression. But first we need also to look again to all the cases studied before and elaborate a general expression to each one. In each of the last three cases, we can see that expression 5.22 is applicable as long as $k \leq d_{DA}$. The point where actor A finishes all the tokens that were originally in the DA edge is also indicated in every Gantt chart presented so far. But for now it will be useful to extend the previous Gantt charts in time. So, for case 1 we have that

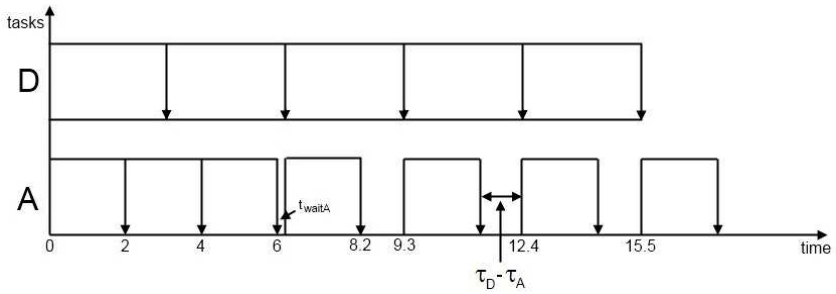

Figure 5.10: Time extension for the Gantt chart in case 1

In a similar fashion, for the second case we have that

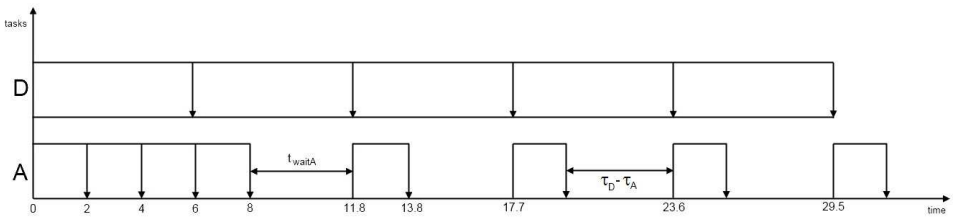

Figure 5.11: Time extension for the Gantt chart in case 2

In both of the previous figures we can see that, after the initial delay, which we designated as t_{waitA}, the system enters into a cyclic behaviour characterized by a waiting period for actor A equal to $t'_{waitA} = \tau_D - \tau_A$. Since this is also the upper bound for our initial quantity t_{waitA}, then we conclude that, as long as actor D as enough tokens to operate, i.e, $d_{BD} \geq 0$, and considering only the iterations after the exhaustion of all the original tokens in the DA edge, we have that

$$\begin{aligned} s(A,0) &= d_{DA} \cdot \tau_A \\ s(A,1) &= d_{DA} \cdot \tau_A + (\tau_D - \tau_A) \\ s(A,2) &= d_{DA} \cdot \tau_A + 2 \cdot (\tau_D - \tau_A) \\ &\vdots \\ s(A,m) &= d_{DA} \cdot \tau_A + m \cdot (\tau_D - \tau_A), \quad \text{if } d_{BD} \geq 0 \wedge \tau_D > \tau_A \end{aligned} \tag{5.24}$$

Contrary to other calculations made in previous sections, this time we present a more conservative expression instead of an exact one. In reality, the conservative nature of the previous expression is only applicable to the first time that actor A is blocked due to the lack of tokens in its incoming edge. We could see on the previous examples that only the first waiting period can have a variable duration, depending on the relative duration of the actors. After that first period, the waiting time stabilizes at a fixed value $t_{wait} = \tau_D - \tau_A$.

5.3.3.2 Low priority actor

The insertion of independent actors between the high and low priority actors creates a new layer of complexity, in terms of timing analysis.

One of the objectives in this section is to elaborate a general expression that is able to predict when the low priority actor would be able to start for the first time. This is an important step because after the first execution of this actor, the system assumes a different pace of execution that is more predictable and is mainly dictated by the executions of the low priority actor.

As we have verified so far, actor B can only be set to execution after actor A has exhausted all of its available tokens. So, since actor A retains that much importance, we have to divide this analysis in three subcategories as on the previous section.

5.3.3.2.1 $\tau_A > \tau_D$:

When actor A is slower than actor D, which it also depends for generating additional data tokens to process, we can see that actor A will end up consuming all the tokens in the DA and BD edge (although indirectly), since actor D it is dependent on actor B in the same way. In other words, the flow of tokens is stopped in actor B, because it is constantly being blocked by the continuous activations of actor A.

We do not need to extend this analysis much more since a similar proof of this situation was already made in section 5.3.3.1.1. We already saw that if $\tau_A > \tau_D$, actor A is able to operate

continuously until it exhaust all the combined tokens in the DA and BD edge. So, taking this point into consideration, we can write that

$$s(B,0) = (d_{DA} + d_{BD}) \cdot \tau_A \tag{5.25}$$

After actor B is able to do its first execution, we stay with complex situation in our hands. The single execution of actor B allows actor D to execute one more time. During this execution, actor B is able to do another execution, since actor D runs on a different platform and cannot block actor B in any way. So, as long as $\tau_D > 0$ we can write that

$$\begin{aligned} s(B,1) &= s(B,0) + \tau_B \\ &= (d_{DA} + d_{BD}) \cdot \tau_A + \tau_B \end{aligned} \tag{5.26}$$

Now we have two possible scenarios:

- If $\tau_D < \tau_B$, then, before actor B is able to finish its second iteration, actor D will finish and produce a token into the DA edge, which allows actor A to preempt actor B. Since edges DA and BD are depleted of tokens at this point, after this single execution of actor A, actor B is able to resume and complete its second iteration. This simplifies our study, since it allows us to write a more regular expression

$$\begin{aligned} s(B,2) &= s(B,1) + \tau_B + \tau_A \\ &= (d_{DA} + d_{BD}) \cdot \tau_A + \tau_B + \tau_A + \tau_B \\ &= (d_{DA} + d_{BD} + 1) \cdot \tau_A + 2 \cdot \tau_B \end{aligned}$$

$$\begin{aligned} s(B,3) &= s(B,2) + \tau_B + \tau_A \\ &= (d_{DA} + d_{BD} + 1) \cdot \tau_A + 2 \cdot \tau_B + \tau_A + \tau_B \\ &= (d_{DA} + d_{BD} + 2) \cdot \tau_A + 3 \cdot \tau_B \end{aligned}$$

$$\vdots$$

$$\begin{aligned} s(B,k) &= s(B,k-1) + \tau_B + \tau_A \\ &= (d_{DA} + d_{BD} + k - 1) \cdot \tau_A + k \cdot \tau_B \end{aligned} \tag{5.27}$$

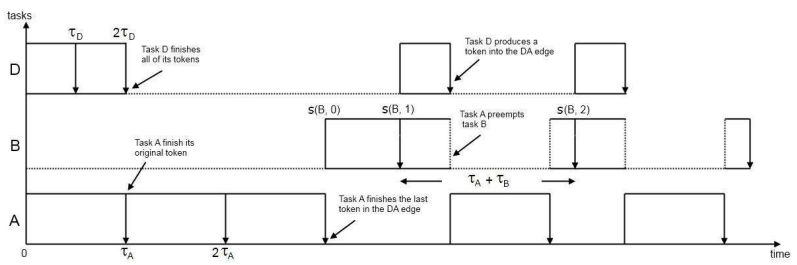

Figure 5.12: Simple example with $\tau_A > \tau_B > \tau_D$, $d_{DA} = 1$, $d_{BD} = 2$ and $d_{CB} = 3$

- If $\tau_D > \tau_B$, than after the second iteration, actor D is still processing the token produced in the first iteration of actor B. The starting time of the next iteration will be

$$\begin{aligned} s(B,2) &= s(B,1) + \tau_B \\ &= (d_{DA} + d_{BD}) \cdot \tau_A + 2 \cdot \tau_B \end{aligned} \tag{5.28}$$

  Eventually actor D will finish executing and when that happens, the token produced by it will be consumed by actor A, which will also preempt actor B from execution. Not only that complicates our analysis but we also need to take into account the fact that actor B has made multiple executions during the last execution of actor D. So, while actor B is being blocked by actor A, actor D has some tokens available to begin executing again, which could lead to future consecutive executions of actor A, depending on the relation between τ_A and τ_D. This specific situation needs to be properly analysed in order to find some rule to predict the further executions of the low priority actor.

5.3.3.2.2 $\tau_A = \tau_D$:

As seen on section 5.3.3.1.2, this situation can be analyse as just the border case of the previous study. In that last section, we have seen than, in an ideal situation, the behaviour of the actors is similar to when $\tau_A > \tau_D$, i.e, actor A only allows actor B to be set to execution after it has consumed all the tokens in the DA and BD edges. Because of that we can simply reapply all the rules stated in the previous section.

5.3.3.2.3 $\tau_A < \tau_D$:

Now we have a high priority actor A that executes at a higher rate than actor D, but since actor A further activations are somewhat dependent on the executions of actor D, this factor introduces a new layer of complexity in the analysis of this situation.

The main question here is: **How long should actor B wait until it can execute for the first time?** So what we want to discover is basically when does actor A stop executing consecutively. Its also important to point out that, for now, we just want to know when does actor B starts for the first time, not when it finishes the first execution. Since in this situation, the stopping of actor A does not necessarily means that the DA and BD edges are out of tokens, actor B can simply be put to execution for an infinitesimal amount of time before actor A pre-empts it again.

The problem here is that now, not only the relative execution times of the actors are important, but also the number of tokens originally in each incoming edge plays a role in this matter. The next figures represent three simple numeric problems that illustrate this situation.

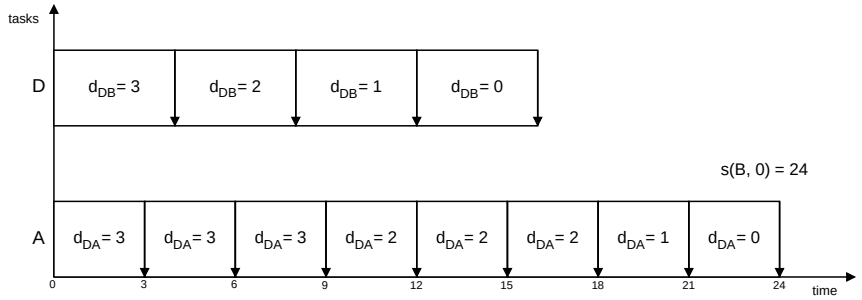

Figure 5.13: Example using $\tau_A = 3, \tau_D = 4, d_{BD} = d_{DA} = 4$

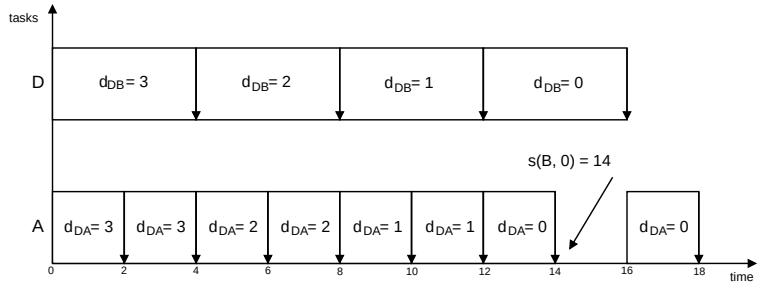

Figure 5.14: Example using $\tau_A = 2, \tau_D = 4, d_{BD} = d_{DA} = 4$

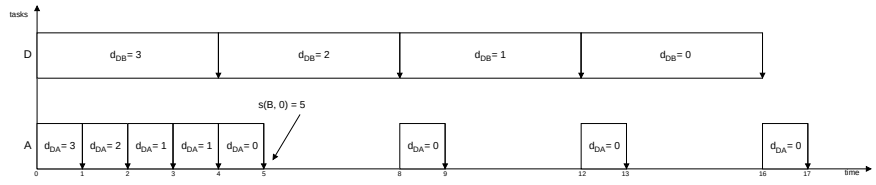

Figure 5.15: Example using $\tau_A = 1, \tau_D = 4, d_{BD} = d_{DA} = 4$

As we can see in the previous figures, actor B can be set to execute after all the tokens in the DA and BD edges are exhausted, like in figure 5.13 or it can be put to execution just after a certain number of tokens processed by actor A. We intend to find out how can we determine this number.

After some careful observations we arrived at the following expression:

$$d_{actA} = \left\lfloor \frac{\tau_A \cdot d_{DA}}{\tau_D} \right\rfloor \tag{5.29}$$

This formula gives us the number of tokens produced into the DA edge, by actor D, while actor A consumes all the initial tokens in the same edge. One of the elapsed times in this situation is the time that actor A takes to process all the tokens that are already in the DA edge at the start of the system. If we divide that number by the time that actor D takes to produce one single token, than we get the number of tokens effectively produced into the DA edge in the meantime. The floor operator is used because this expression must return a whole number and to guarantee that the tokens produced by actor D are only accounted at the end of its execution.

This formula is useful since we can conceive a recursive algorithm that basically reapplies the expression at the result of the last iteration of the same expression. We can see that consecutive applications of the algorithm result in ever decreasing number of additional tokens produced into the DA edge, so it safe to say that we can obtain a final value for the number of extra tokens produced when the algorithm converges.

We can put expression 5.29 into a formalization of the referred algorithm:

$$d_{act}(A, k) = \left\lfloor \frac{\tau_A \cdot (d_{inEdge} + d_{act}(A, k-1))}{\tau_{prodActor}} \right\rfloor \tag{5.30}$$

where

$$d_{act}(A, k) = 0, \forall k < 0 \wedge k \in \mathbb{N}_0 \tag{5.31}$$

The d_{inEdge} term refers to the number of tokens in the incoming edge, which in this particular case is the DA edge. The $\tau_{prodActor}$ term is the execution time of the actor that produces tokens into the incoming edge of the actor in question, which in this case is actor D. The convergence of this algorithm is verified when $d_{act}(A,k) = d_{act}(A,k-1)$.

The convergence value $d_{act}(A,k)$ is the number of additional tokens, after the initial ones, processed by actor A. The total number of tokens processed continuously y actor A is given by $d_{initialTokens} + d_{actConvergent}(A,k) = d_{DA} + d_{act}(A,k)$.

So, the starting time of the first iteration of actor B can now be found by:

$$s(B,0) = (d_{actConvergent}(A,k) + d_{DA}) \times \tau_A \tag{5.32}$$

This algorithm was reached in a holistic fashion, so its better to run an example to clarify it. Lets start by considering one of the examples seen before:

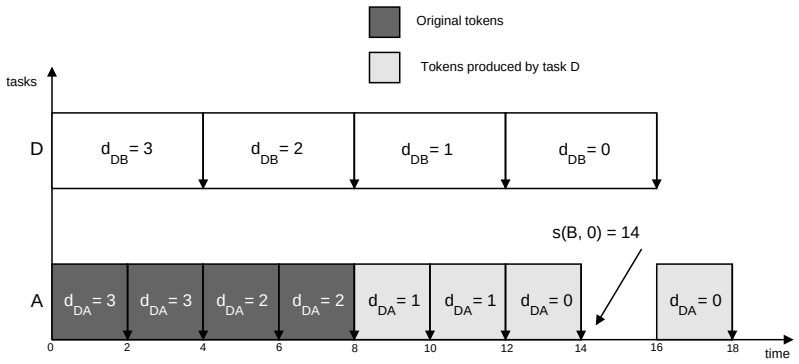

Figure 5.16: Case using $\tau_A = 2, \tau_D = 4$ and $d_{BD} = d_{DA} = 4$

We see that actor A starts with 4 initial tokens in the DA edge and then it gets at least 3 more from the parallel execution of actor D. In this case in particular, actor B should be set to do its first execution at $t = 14$, even though it will loose the processor 2 units of time after it. For now we just want to determine the starting time of the first execution of the low priority actor.

So we start by running the first iteration of the algorithm in 5.30:

$$\begin{aligned}
d_{act}(A,0) &= \left\lfloor \frac{\tau_A \cdot d_{DA}}{\tau_D} \right\rfloor = \left\lfloor \frac{2 \cdot 4}{4} \right\rfloor = \left\lfloor \frac{8}{4} \right\rfloor = 2 \\
d_{act}(A,1) &= \left\lfloor \frac{\tau_A \cdot (d_{DA} + d_{act}(A,0))}{\tau_D} \right\rfloor \\
&= \left\lfloor \frac{2 \cdot (4+2)}{4} \right\rfloor = \left\lfloor \frac{12}{4} \right\rfloor = 3 \\
d_{act}(A,2) &= \left\lfloor \frac{\tau_A \cdot (d_{DA} + d_{act}(A,1))}{\tau_D} \right\rfloor \\
&= \left\lfloor \frac{2 \cdot (4+3)}{4} \right\rfloor = \left\lfloor \frac{14}{4} \right\rfloor = 3
\end{aligned} \tag{5.33}$$

As we can see, convergence was achieved after 3 iterations, returning 3 extra tokens consumed in a continuous fashion by actor A, after the original 4. So the initial starting time for actor B, using expression 5.32, is

$$\begin{aligned} s(B,0) &= (d_{act}(A,k-1)+d_{DA})\times\tau_A \\ &= (4+3)\times 2 \\ &= 14 \end{aligned} \tag{5.34}$$

Which correspond to the value shown in figure 5.16. This was just a short example, chosen by its simplicity. This algorithm was already implemented in software, through a simple simulator written in OCaml.

5.4 Software implementation

In order to be able to check the validity of our theory, we had to modify the existing set of tools, particularly the data flow simulator, to include a fixed priority option. Specifically we had to introduce the following changes:

- **Include an option to activate fixed priority simulation** - In order to preserve stability, its not wise to alter the core code to meet our needs. The most flexible approach is to write our own code inside a conditional branch. This way, the new code is only run if the proper option is activated via the command line, allowing other users to run the original simulator at will.
- **Alter the actor internal structure to include a *priority* and a *processor to map* field** - These will be one of the most used fields in our adaptation. The **priority** field is straight forward while the **processor to map** will be used to allow the simulation of fixed priority systems in which groups of actors can be mapped into different platforms.
- **Alter all the actor creating and manipulating functions to deal with the extra fields mentioned before** - There are several functions that operate on actors that must be adapted to include the new fields of priority and processor mapping.
- **Edit the parser and lexer files to permit the definition of the new fields** - The characteristics of the system to be simulated (name and execution time for all the actors, delays and consumption rates for the edges, among other fields) are inserted into the simulator through a text file. This file is interpreted by a lexer and a parser, so they need to be altered in order to be able to recognize and deal with the extra fields introduced.
- **Alter the *compare* function, responsible to do the ordering of the events** - One of the main changes originated by an inclusion of a fixed priority scheme is the order in which the various events are put into the set. It is through this function that we can make sure that a high priority event is processed before a low priority one.
- **Create an internal structure to deal with the various processors** - In our simulations, sometimes we want to investigate the behaviour of a graph in which there are

actors mapped into different processors. In order to accomplish that, we need to conceive some sort of mechanism that allows us to simulate the available platforms so that we can determine when they are *idle* or *busy*. One approach is to create an internal structure that represents a mappable processor.

- **Alter the stopping condition of the simulator** - The default version of the simulator looks for a periodic behaviour in the graph executions. For that purpose, the simulator records several states of the graph and compares the actual state with all the previous states. This aspect forces the graphs in analysis to be strongly connected. It turns out that, for some of our experiments, the graphs are not strongly connected, which leads the simulator to run indefinitely, looking for a periodic behaviour that will never appear. The stopping condition for the simulator will be change time limit relative to the main simulation clock.

One important aspect to refer is that the fixed priority simulation does not allow preemption. For now, we are just simulating ideal and non pre-emptive systems.

Upon applying these changes, the overall system was similar to the existing one. We now possess two *compare* functions in our code: the regular one and another that is active when the fixed priority option is active. This last one uses the fixed priority value in each event to decide the position that an occurrence of an actor will have in the set of events. After this, the only difference from the original simulator resides in the necessity of a processor available, along with all the other firing rules, for an actor to begin executing. Also, if a processor is attributed to a particular actor, all the other actors that were waiting for that same processor need to have their activation times rescheduled. The flowchart in the next figure illustrates these changes:

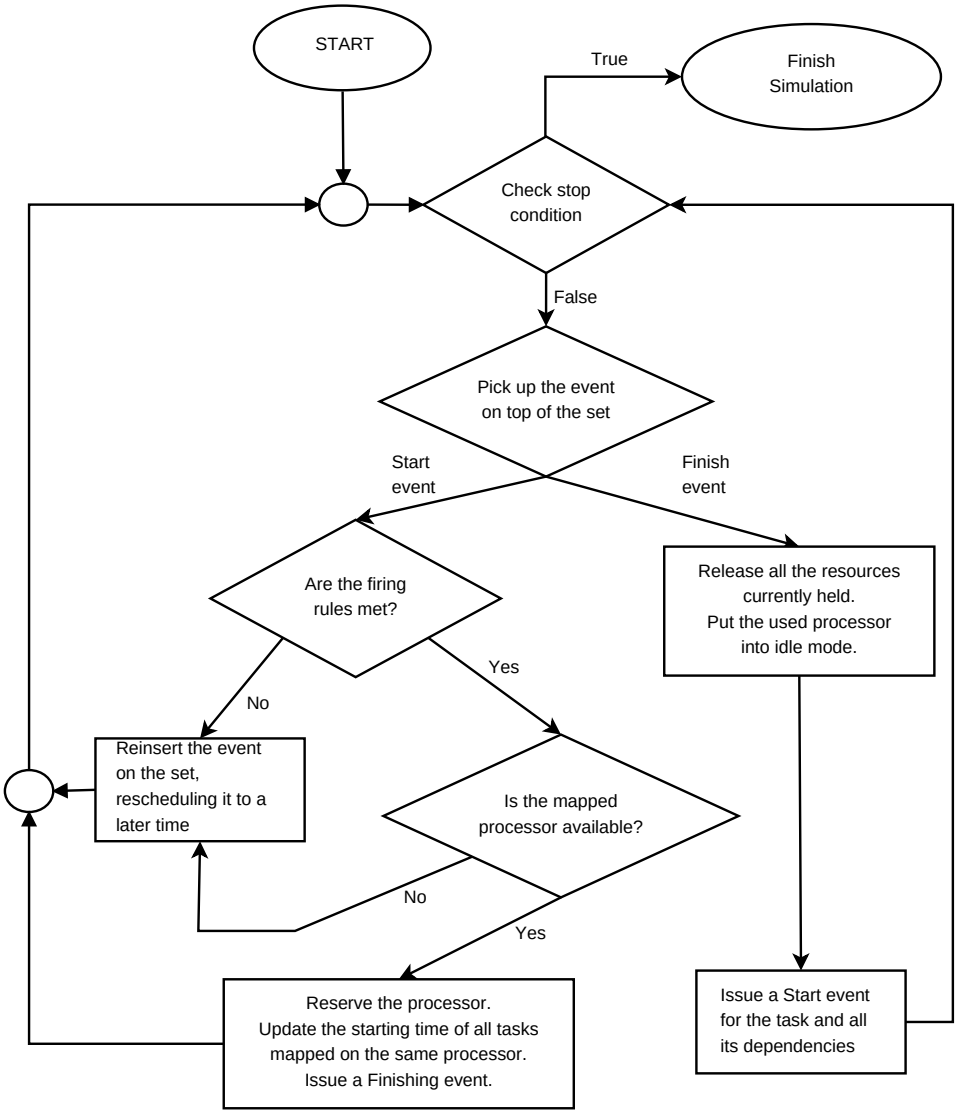

Figure 5.17: Flowchart with the changes made to the simulator to deal with fixed priority schemes

5.5 Conclusion

In this chapter we tackled with the problem of modelling fixed priority graphs. With a simple graph, it is practical to produce a set of equations that model the best and worst-case response times, as also the starting and finishing times of the executions of all the actors. The purpose of our project is modelling fixed priority schedulers in a multi processor environment, so it is only logical that our next analytical step was the study of a fixed priority graph in which some of its actor are mapped into different platforms. Although the example that we started with was not the simplest possible, we were able to understand how complex this analysis was. This increase in complexity is not due to an intrinsic difficulty in reaching some conclusions, but mainly to the fact that any simple analysis will eventually divide itself into several sub cases, and each one of

those originates it own modelling expression.

Chapter 6

Inter-graph fixed priority analysis

This chapter continues the study of the fixed priority problem but from a different perspective. Now we will direct our attention to the interference between multiple jobs with different priority values, mapped on the same processing platform. In this context, we consider a job as a set of task instances. This type of situation is common in most embedded systems where the processors, general purpose or task oriented, are to be used by several independent jobs.

6.1 Problem definition

On a multi-radio baseband system, multiple independent transceivers must share the resources of a multi-processor, while meeting each its own hard real-time requirements. These transceivers execute several tasks in sequence.

Each processor is a system resource and, as such, its utilization should be optimal. In this section we are going to address the resource sharing problem for several jobs with different levels of priority. Our study will focus mostly on the concept of *computational load*. Specifically, we want to characterize the system when it is subject to a *maximum computational load*. The maximum load of the system is always due to the activations of a high priority task and we intend to investigate the influence that such behaviour could produce in the response-time of other lower priority tasks that are also mapped on the same processor. One of the problems that we also want to address is how to generate an upper bound on the *maximum load* that a job can create on another.

In order to establish a worst-case scenario analysis, it is essential that we are able to calculate the *maximum load* a given processor can be subjected due to a particular set of tasks or job. The *maximum load* has an associated concept designated as *load window*. We need also to define this concept, which is essential to expand our study regarding the maximization of the load function.

As before, we also had to transform the software tools available in order to use them in this context.

6.2 Theory

6.2.1 Definition of load of a processor

In the context of this work, it is useful to define a concept that quantifies the amount of work required from a processor by a particular task. In a fixed priority scheme, one of the aspects that it is important characterize is the amount of time that a processor is busy with a task, which also allow us to know the amount of time remaining for processing other tasks.

Definition 6.1. ***Load of a processor*** *We define load of a processor due to a particular task i, with a worst-case execution time τ at a given time interval t as the cumulative time that the processor spent processing that task i. Considering this, the load of task i on processor p, at time t, for a particular start time function s, can be calculated with the following expression:*

$$C(t, \tau, i, s) = \sum_{k:s(i,k)+\tau(i,k)\leq t} \tau(i,k) + \sum_{k:s(i,k)<t<s(i,k)+\tau(i,k)} (t - s(i,k)) \tag{6.1}$$

The previous expression is the general form of the load formula, which was distilled from other more simple configurations. In the next sections we will document the process that led us to expression 6.1. The function requests a time interval t, a task i and a vector of all the start times for that task instances. The summation is performed to all execution times of task i whose starting time, $s(i,k)$, added to the same execution time, $\tau(i)$, is less or equal than the time interval t considered.

6.2.2 Initial considerations and evolution of the concept

In this section we present a detailed description of the steps required to reach expression 6.1. The formal definition of the concept of processor load was established from an early stage. The question that remained was how to find a mathematical expression to calculate this value. A simple approach upon reading definition 6.1 resulted in the following expression:

$$C(t, \tau, i, s) = \sum_{k:s(i,k)<t} \tau(i) \tag{6.2}$$

In order to illustrate the usage of the expression, let us refer to a simple example:

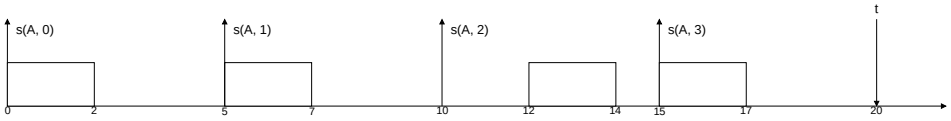

Figure 6.1: Simple example

If we analyse the previous example using expression 6.2,we can conclude that:

$$
\begin{aligned}
C(20,2,s,A) &= \sum_{k:s(A,k)<20} \tau_A \\
&= \sum_{k=0}^{3} 2 \\
&= 8
\end{aligned} \tag{6.3}
$$

In this case we considered that all the executions in which we are interested were finished before the instant t. If we now consider a more complex and possible scenario, we can see that expression 6.2 is not entirely correct:

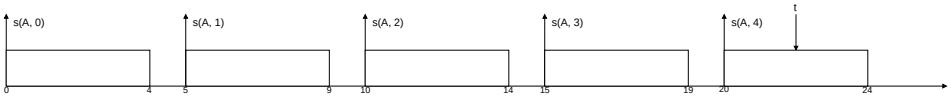

Figure 6.2: Situation where the time interval considered includes a partial execution

Equation 6.2 does not take into account partial executions of a task. We need to correct that.

6.2.2.1 Inclusion of partial executions in the load formula

We are going to consider that all executions taken into account have a constant execution time so that $\tau(i,0) = \tau(i,1) = ... = \tau(i,k) = \tau(i)$. From figure 6.2 we can suggest two approaches for this problem:

- Take into account all the executions until the time limit t, including a possible partial one at the end, and then remove the excess part of this execution that follows after the mark in t. Putting all this into an expression we have that:

$$
C(t,\tau,i,s) = \sum_{k:s(i,k)<t} \tau(i) - \sum_{k:s(i,k)<t<s(i,k)+\tau(i,k)} (s(i,k) + \tau(i) - t) \tag{6.4}
$$

  Here, the summation is made in excess. Once the first operation is complete, the last execution corresponds to the partial one, so we need to remove the part of it that appears after the t mark. For that we subtract the term $s(i,k) + \tau(i) - t$. The application of a summation on this term allow us to simplify the overall expression since it is evaluated only when the choice of t causes a partial execution to be considered.

- Consider all the integer executions on a first stage and then add the partial execution on a second stage of the calculations:

$$
C(t,\tau,i,s) = \sum_{k:s(i,k)+\tau(i)\leq t} \tau(i) + \sum_{k:s(i,k)<t<s(i,k)+\tau(i)} (t - s(i,k)) \tag{6.5}
$$

  The summation takes care of accumulate all the completed executions until the considered time t. The last element of the first summation corresponds the last whole execution before

t, so the next execution will correspond to a partial execution. The summation before this term makes sure that this one is only evaluated when it is necessary.

Both expressions are valid, produce the same outcome and have the same problems, so one can choose any of them to compute the processor load. For the rest of the document we will use expression 6.5.

6.2.2.2 Dealing with non constant execution times

Expressions 6.4 and 6.5 result from a simplistic approach to the problem. In the previous sections we assumed that all executions of the task i had constant duration. In some cases we may want to take into account the possibility of deviations from the standard execution time value. Since we decided to go forward with expression 6.5, we suggest the following adaptation for it:

$$C(t,\tau,i,s) = \sum_{k:s(i,k)+\tau(i,k)\leq t} \tau(i,k) + \sum_{k:s(i,k)<t<s(i,k)+\tau(i,k)} (t - s(i,k)) \tag{6.6}$$

And thus we reached our first expression 6.1. The main difference from expression 6.5 reside on the fact that now all the executions of task i are indexed through k, just as every starting time. Although more precise, this method introduces a new level of complexity, since we need to know the execution time of every single execution.

6.2.3 Establishing time intervals

So far, in all the examples presented, the load calculation is made considering the origins of time as a fixed lower edge. Due to this detail, the time instant t referred could be considered as an absolute value while in reality it should be considered as an interval. To avoid this type of confusion in the future, we are going to replace the t parameter for a more intuitive one, Δt.

As was referred before, all calculations up to here were done considering $t_0 = 0$ as the lower bound for the definition of our time interval Δt. In reality, the time instant at which the high priority job allows the execution of a low priority task is unknown *a priori*. As such, we cannot establish the lower bound of our considered time interval to origin of the referential. A *load window* that maximizes the load function does not necessarily need to be started in $t_0 = 0$.

All the previous presented formulas for the calculation of the load of a processor omitted the t_0 element since it was equal to zero. Taking this into consideration, a more detailed description for the load function is needed. First, let us observe a typical application for this expression in order to comprehend some of its functionalities:

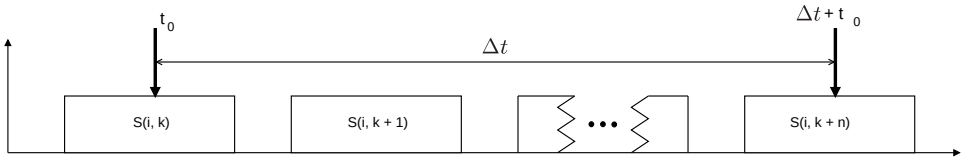

Figure 6.3: Example where a time window is established

As we can see from the previous figure, if the lower bound for out time window, t_0, is not zero, we can face a similar situation as the one described in 6.2.2.1. In order to deal with this situation, we need to adapt expression 6.1 in a similar fashion that was done for when the upper bound can create partial executions:

$$\begin{aligned} C(t_0, \Delta t, \tau, i, s) &= \sum_{k: t_0 \leq s(i,k)+\tau(i,k) \leq \Delta t + t_0} \tau(i,k) \\ &+ \sum_{k: s(i,k) < \Delta t + t_0 < s(i,k)+\tau(i,k)} \Delta t + t_0 - s(i,k) \\ &- \sum_{k: s(i,k) < t_0 < s(i,k)+\tau(i,k)} t_0 - s(i,k) \end{aligned} \tag{6.7}$$

The accounting of the task executions is made by defect in the upper bound of the window and by excess in the lower bound. Hence, the second summation of the formula adds the partial execution left out by the first term while the third term removes the part of the first execution that is added in excess.

Through this expression we can see that the element t_0 is an absolute time quantity while the term t is a time interval. The upper bound of the time window is now found by adding these two terms together.

6.2.4 Getting the maximum load

Now that we got some flexibility through what was defined in the previous section, we can use this to establish what we define as *maximum load*.

Definition 6.2. ***Maximum load****: For a given time window defined by an initial time instant t_0 and a time interval Δt, a load function $\hat{C}$ is maximum if it holds the following:*

$$\hat{C}(t_0, \Delta t, \tau, i, s) \geq C(t'_0, \Delta t, \tau, i, s), \quad \forall \Delta t > 0 \tag{6.8}$$

where t'_0 is any instant of time different from t_0.

Next we will illustrate, with an example, how the redefinition of the time window can allow us to find a greater load function.

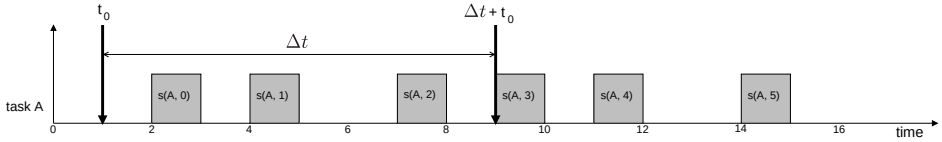

Figure 6.4: Initial definition of the time window

From figure 6.4 we can calculate the load of the processor due to task A for the window considered. Since none of the limits of the time window creates a partial execution to be considered, we will going to omit the calculation of the second and third terms of expression 6.7.

$$\begin{aligned} C(1,8,1,A,s) &= \sum_{k:1\leq s(A,k)+\tau(A,k)\leq 9} \tau(A,k) \\ &= \sum_{k=0}^{2} 1 \\ &= 3 \end{aligned} \tag{6.9}$$

Although, one can see that if we shift the window just one time unit to the right we can increase our load function:

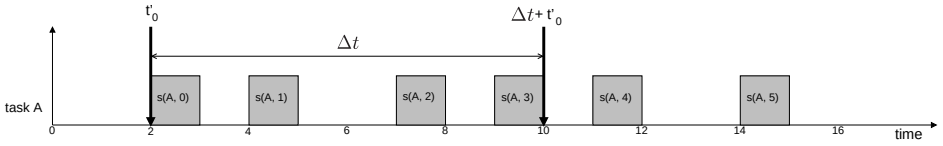

Figure 6.5: Example with the time window right shifted

If we calculate the load function for this new situation, we get that

$$\begin{aligned} C'(2,8,1,A,s) &= \sum_{k:2\leq s(A,k)+\tau(A,k)\leq 10} \tau(A,k) \\ &= \sum_{k=0}^{3} 1 \\ &= 4 \end{aligned} \tag{6.10}$$

We were able to increase the load of the processor for this particular situation by shifting the time window, which allowed us to include another extra execution to the processor load.

From an example like this arises an important question:

How can we make sure that there is no other time window, also only obtainable by shifting the previous one in time, that gives us an even bigger load function?

Establishing a *maximum load window* for the execution of a given job in a processor is an important step towards establishing a worst-case execution time for a low priority task that executes concurrently with the job that generates such a computational load. If we are able to identify an interval of time in which a given processor is subject to the highest computational demand from a job, then we can use this information to compute how long should a low priority task take to complete a full execution in those conditions, i.e, using only the idle times left by the high priority job. If the task is able to execute in these conditions, the time taken to do it is our **worst-case execution time** due to the fact that the load imposed in the processor was maximum.

Ideally, a formal proof is needed to secure the validity of this statement. But for now, all we can present are a series of statements that eventually can lead to such a proof. In the remainder of this section, we are going to introduce all the definitions and conjectures that we verified so far, regarding this subject.

6.2.4.1 Generating the maximum load in an SRDF graph

One important aspect to address at this point is how to subject a given processor to a maximum computational load from an actor, using data flow concepts.

For a SRDF graph, we define the maximum load as the maximum time that the processor, in which an actor i is mapped, is busy processing the executions of that actor, in a load window defined by the initial time t_0 and the amplitude Δt parameters.

We want to devise a process in which, by just altering the initial configuration of a data flow graph, we can create a defined time interval in which our processor is subject to the maximum load possible from that job.

First, lets set up a simple example of the situation that we are addressing.

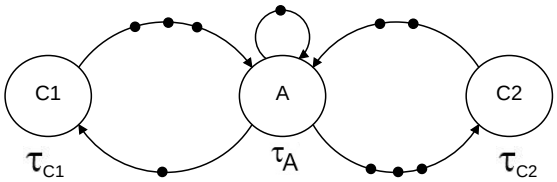

Figure 6.6: Example of a condensed model for a single rate data flow graph

Figure 6.6 represents a condensed model of the type of graphs that we intend to study. Actor A represents the high priority task that will be responsible for generating the load in its mapped processor and the surrounding actors represent all the cycles from which actor A takes part of, condensed into a single actor cycle. The execution time of such an actor is equal to the largest execution time of all the cycles that are condensed in that actor. Similarly, the edges included represent the bundle of incoming and outgoing edges from the actor on focus to the surrounding cycles.

The graph represented is a self-timed scheduled graph, which means that each actor fires as soon as it has its firing rules met. As we have seen before in 2.5.1, once this graph is started, at $t = 0$, it will spend some time in a *transient phase* until it eventually reaches a *periodic phase*. The *transient phase* can be explained as a phase in which the graph is moving its tokens around the edges until it is able to reach the first state of the periodic phase. After this point, the behaviour of the graph can be described as a finite sequence of states that repeat themselves with a fixed period. We realised that, as soon as the graph enters the *periodic phase*, the computational load that a given actor imposes on the processor is constant during that period. Also, the computational load has a direct relationship with the number of tokens present in the incoming edges of the actor selected to impose this load. The greater the number of consecutive firings of an actor, the greater the load that it imposes in its assigned processor. Since the number of consecutive firings of an actor is directly related to the number of tokens in its incoming edges, we are looking for a way to accumulate the maximum tokens possible in the incoming edges of an actor so that it is able to make a maximum number of consecutive firings, generating the maximum load in the processor.

To achieve this goal, we devise the following strategy: we introduce a slow actor in the graph, directly connected to the actor that we want to study. We call this actor the *delay actor*. Along with a large execution time (when compared with the execution time of the rest of the actors in the graph), this *delay actor* possess a self edge with a large number of tokens. Figure 6.7 exemplifies this actor and how it should be inserted in the graph.

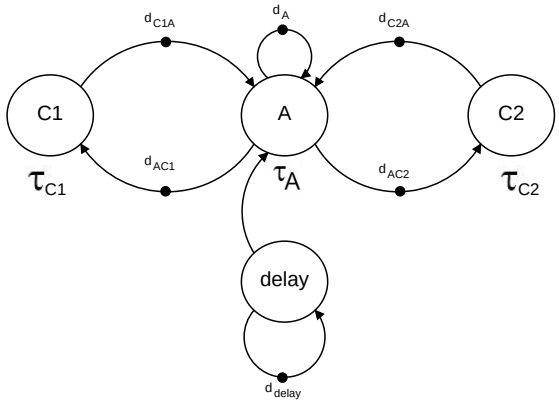

Figure 6.7: Example of an insertion of a *delay actor* into an existing graph

For the graph represented, we have the following:

- $\tau_{delay} >> (\tau_A, \tau_{C1}, \tau_{C2}, ..., \tau_{Cn}), \quad n \in \mathbb{N}.$
- $d_{delay} >> (d_A, d_{C1A}, d_{AC1}, ..., d_{CnA}, d_{ACn}), \quad n \in \mathbb{N}$
- $d_{delayA} = 0$

The purpose of this actor is simple: once the whole system is activated at $t = 0$, the *delay actor* creates an extra dependency on actor A. So, as long as the *delay actor* does not finishes its first execution, which will put d_{delay} tokens in the *delayA* edge, actor A cannot file. In the meantime, all the remaining actors in the graph continue their normal executions until eventually they are stopped due to the fact that they are inserted in cycles that depend on actor A. After a certain amount of time $t \leq \tau_{delay}$, the whole graph is stopped since the surrounding cycles are waiting for actor A to produce some tokens into its outgoing edges. This process is going to accumulate tokens in the incoming edges of actor A. Since the total number of tokens in the graph is finite and constant, we can assume that the number of tokens accumulated in the incoming edges of actor A at this point is maximum.

Once the *delay actor* finishes its first execution, it produces a large number of tokens into the *delayA* edge. This way, the influence of this actor is removed from the remainder of the graph execution: if the number of tokens present in the *delay actor* is enough, after the first firing of this actor, there will be enough tokens in the respective incoming edge of actor A so that it never need to wait for the *delay actor* to input more tokens into this edge.

If the number of tokens present in the incoming edges of actor A is maximum, then once the *delay actor* finishes its first execution, actor A will execute the maximum consecutive number of times, creating the worst-case scenario in terms of processor usage - the **maximum load**.

6.2.4.1.1 Values for d_{delay} and τ_{delay} :

The execution time of the *delay actor* should be large enough to allow all the cycles in the graph to stop due to the lack of tokens before this actor finishes its first execution. Since the cycles surrounding the actor on focus execute in parallel (we are assuming that they are mapped

into different processing platforms for simplicity), a good starting value for τ_{delay} is the product between the execution time of the slower cycle and the total amount of tokens inside that cycle.

$$\tau_{delay} \geq \max(\tau_{C1}, \tau_{C2}, ..., \tau_{Cn}) \cdot \sum_{i \in Cm} d_i, \quad n \in \mathbb{N} \tag{6.11}$$

where Cm identifies the cycle with the larger execution time.

This product gives us an upper bound for how long could any of the cycles in the graph run while one of its actors, actor A, is stopped. By choosing τ_{delay} equal or greater than this value, we make sure that all the cycles are stopped when the *delay actor* finishes its first execution.

The d_{delay} value should be chosen such as actor A does not have to wait for tokens from the *delay actor* after it finishes its first execution. We want to avoid another blocking of actor A due to the *delay actor*, a simple way to achieve this is to make:

$$d_{delay} \geq \left\lceil \frac{\tau_{delay}}{\tau_A} + 1 \right\rceil \tag{6.12}$$

After the *delay actor* as finished its first execution, lets assume that actor A has infinite tokens in all of its incoming edges except the *dealyA* edge. As so, it is able to fire continuously until it consumes all the tokens in this edge. A way to prevent the exhaustion of tokens from this edge between consecutive firings of the *delay actor* is to put in there more tokens than the ones that actor A is able to process during a τ_{delay} interval of time. It is true that a setup like this is unstable since the number of tokens in the *delayA* edge is going to grow to infinite. But since it is nothing else but a construct to use only in a simulation environment and for a limited amount of time, it is viable.

To understand this process correctly, lets run a small example from the graph depicted in figure 6.7:

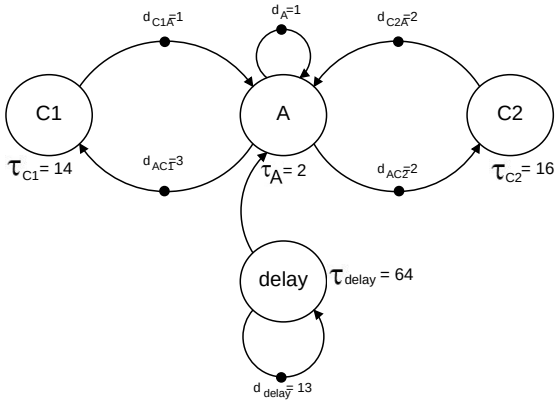

Figure 6.8: Example of usage of the *delay actor*

The characteristics of *delay actor* were obtained using expressions 6.11 and 6.12.

Simulating the previous graph, we obtain the following results:

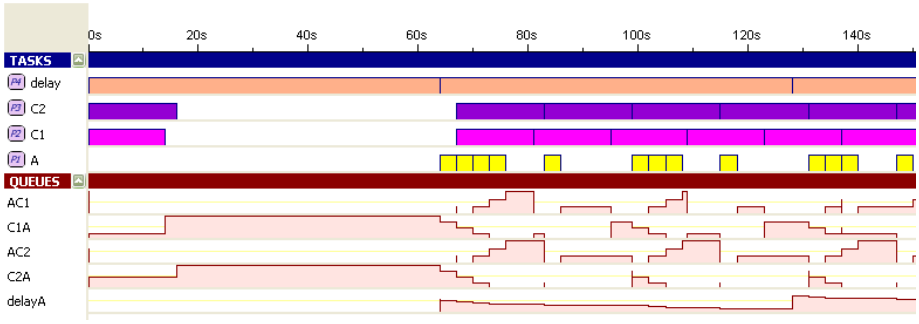

Figure 6.9: Gantt chart of the simulation results with the evolution of the edges

There are some interesting conclusion that we can derive from this results:

- The execution time of the *delay actor* was more than enough to produce the desired effect.
- The number of tokens in the self edge of the *delay actor* was also sufficient to restrict the effect of the *delay actor* to just its first execution.
- This procedure was able to generate the highest possible number of consecutive executions from actor A. Although the figure supplied does not have enough information to verify that, from the results we were able to verify that the maximum load for the processor $P1$ was found when actor A was able to fire for the first time.
- Both the $C1A$ and $C2A$ edges end up with all the tokens in their respective cycles while the opposing edges, $AC1$ and $AC2$ are deprived from all their tokens. All the tokens from the cycles from which actor A belongs were accumulated in the incoming edges of this actor.
- The *delayA* edge is only empty before the end of the first execution of the *delay actor*. We can see that it is replenished with d_{delay} tokens before actor A is able to process the ones from the previous activation. This guarantees that actor A is not getting blocked due to lack of tokens from that edge ever again.

6.2.4.2 Influence of the execution time of the high priority tasks

The high priority tasks are the ones responsible for generating the load on a processor. In this section we are going to infer on the influence of their execution time and the respective load generated. Lets consider the following situation:

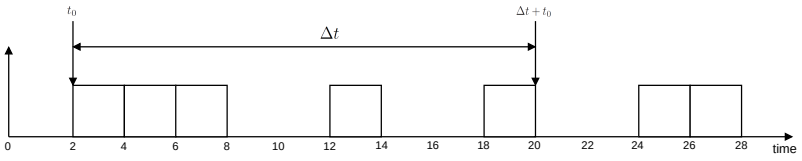

Figure 6.10: Example of a setup of a load window

In figure 6.10, we have a high priority job creating the temporal behaviour shown. For simplicity, we set up all the tasks from that job with the same execution time $\tau(HP, k) = 2$. We

then selected a *load window* by establishing an initial time instant $t_0 = 2$ and a time interval $\Delta t = 16$. The question that we want to answer is, **if we maintain the same load window and vary the execution time of the high priority tasks, what is going to be the influence of this in the load function?**

6.2.4.2.1 Shortening of the execution time :

Let us assume that the execution time indicated for these tasks is its worst-case execution time. So, from this point, the tasks can only execute at the same or a lower execution time. If the execution time of the tasks that compose the high priority job was shortened, the load of the processor would **remain the same**, at best, or it would also diminish. We base this assumption in the monotonicity property: if a task executes sooner than it was suppose to, the subsequent tasks cannot be scheduled later, only sooner. What are the consequences of this event in our load function? If one or several of the tasks that generate the load of the processor finish their execution sooner than their worst-case execution time, this shortening of executions will "pull" part of the executions right outside our *load window* into it.

Before we go any further with this exercise, lets look at a concrete situation. If we observe figure 6.10, we can calculate the load present in the window selected:

$$\begin{aligned} C(t_0 = 2, \Delta t = 16, s, HP) &= \sum_{k:1 \leq s(HP,k)+\tau(HP,k) \leq 9} \tau(HP,k) \\ &= \sum_{k=0}^{4} 2 \\ &= 10 \end{aligned} \tag{6.13}$$

All the tasks represented are executing with their worst-case execution time. Let us now assume that the first task inside the load window finishes sooner than its worst-case execution time:

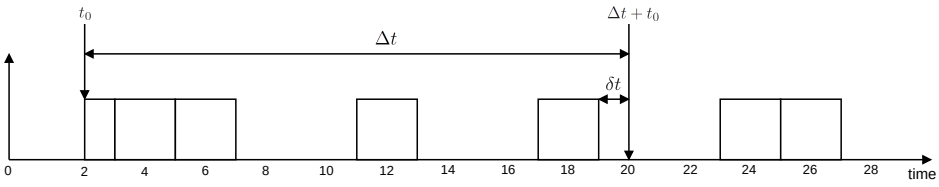

Figure 6.11: The first activation from figure 6.10 finishes δt sooner than its worst-case execution time

If an execution terminates sooner, we can see that two outcomes can happen:

1. The task does not have any dependencies. In this case, this shortening of the execution time does not bear any consequences since all subsequent tasks are still scheduled in their original time instants.

2. The subsequent tasks that fire after the shortened task are dependent on it and as such, they are also scheduled δt before their original time: all the activations inside the *load window* are shifted left by δt, which creates a space of δt time at the right extreme of the *load window*. Now, we must take into account that the diminishing of the execution

time creates also a diminishing by δt of the corresponding load inside the window. The space at the end of the *load window* is going to be filled anyway since the whole graph is getting shifted to the left. Now, if that space is filled with idle time from the processor, the load inside the window gets diminished by δt. In a best-case scenario, the *load window* is terminated just before the beginning of a new execution. This δt shift will make δt from that execution to slip into the load window. But in the end, the amount of load that got inside the load window is equal to the amount of load that was removed by the shortening of one of the executions:

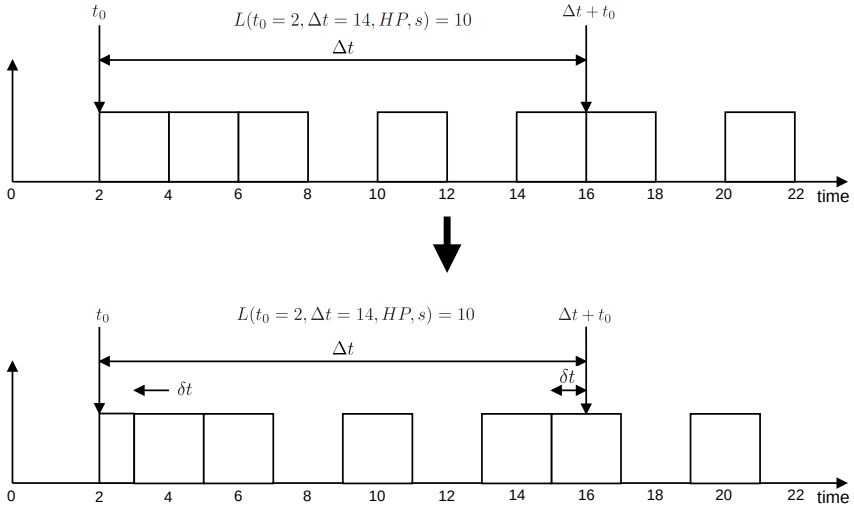

Figure 6.12: Example where the shortening of an execution does not change the load of a processor

So, in conclusion, if a given high priority job has its its tasks executing with a worst-case values for their execution times, the load obtained for the processor this way is maximum and can only be maintained or diminished if one or more tasks take less time to execute.

6.2.4.2.2 Delaying the execution time Let us look at this issue from the opposite point of view: assume now that a certain load function was established through best-case execution times and one of these executions has its finishing time delayed by a δt amount of time. What happens to the load defined by the *load window* in this case?

If a task inside the load window has its execution time elongated, unless it has some cushion space, i.e, some idle processor time after its execution, it will eventually push all the subsequent tasks that execute right after its finish. Let us consider that all the activations inside the *load window* are interdependent, which allow us to establish a worst-case scenario and use the monotonicity argument. The monotonicity of the schedule implies that if, in a stream of interconnected executions, one of them has its finishing time delayed by an amount of δt, the subsequent executions can only be scheduled to start later than their original time and never sooner.

As before, we have two possible outcomes from this case:

1. If the *load window* terminates at the end of an execution, as long as the delaying time $\delta t \leq \tau_{execution}$, the load inside the window remains the same: the extra load gained by delayed the finish time of a task is compensated by the load lost by pushing the last execution partially outside the *load window*.

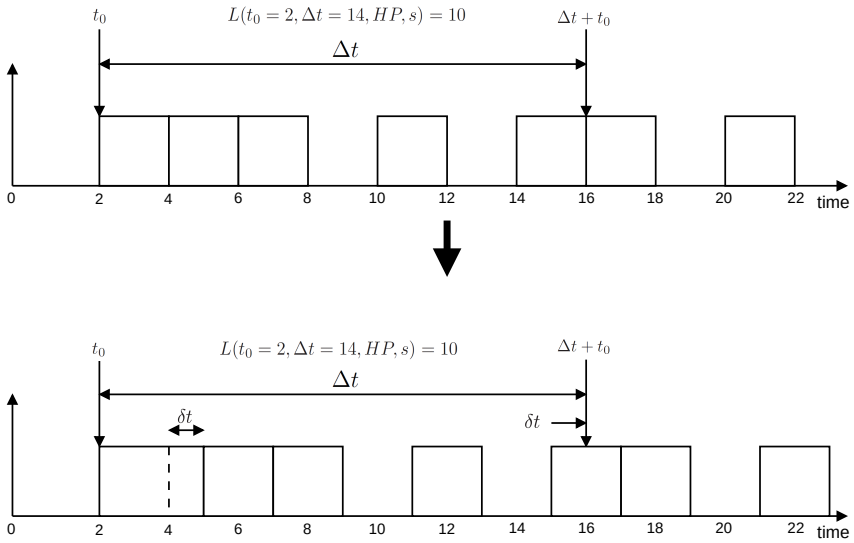

Figure 6.13: Example where the load inside the window remains the same after one of the executions has is finishing time delayed

2. If the *load window* has some idle processor time right before its right delimiter, then a right shift of all of the *load window* executions can push that "empty" space outside of the *load window* and this, increase the total load.

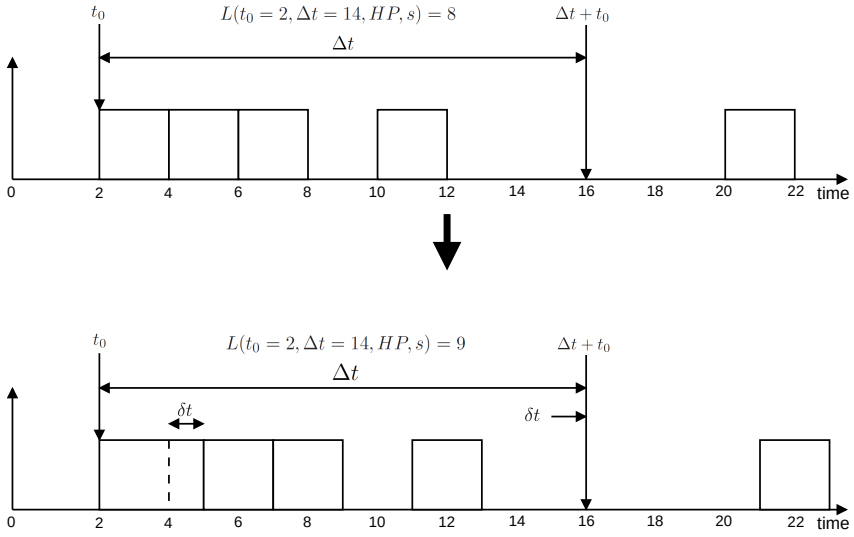

Figure 6.14: Example where the increase of an execution time of a task creates an increase in the load inside of a window

6.2.4.3 Additional considerations

Before we progress in our analysis, we need to clarify a concept first: in all the graphs represented in this context, we saw that the activations responsible for creating the load in the processor were all consecutive activations, i.e, at a given point in time, only one tasks is active

in the processor. In data flow, this effect is achieved by including a self edge with only one token in the actor in study. This guarantees that this actor is able to fire only once at any given time.

In our load considerations, it is important that the actor that generates the load is unable to fire multiple times. If so, all the analysis performed becomes invalid since we cannot predict how many multiple activations can occur at a given point. We investigated the influence that a change in the execution times of the tasks that generate a given computational load have in that load. If we stipulate a certain *load window* and then have one of the activations inside of it to change its execution time, we are going to have temporal shifts from and to the inside of this window that are going to create some changes in the load inside of it. Because of this, there is the possibility that additional load initially outside the *load window* can be brought inside due to a shortening in the execution time of one or more of the tasks whose executions are inside the *load window*. The presence of multiple simultaneous activations can invalidate all the conclusions achieved in the previous sections. To exemplify this behaviour, let us look at the following figure:

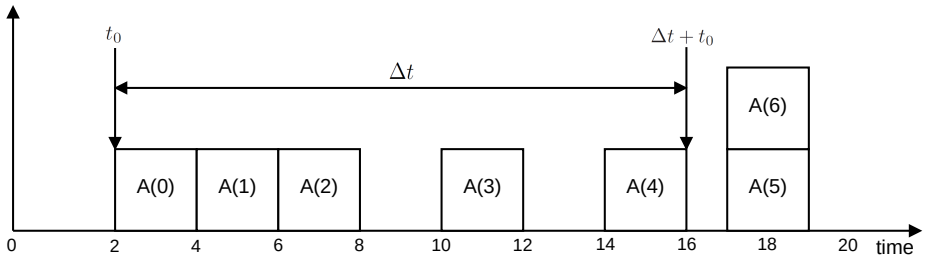

Figure 6.15: Case where a task A that is responsible for generating the load of the processor is allowed to fire multiple times

In this situation, task A is the high priority task whose executions are going to generate the load in the window defined. If one or more of the tasks inside the window have their execution times shortened, some of the executions that sit just outside the right end of the load window can be pulled in. This includes executions $A(5)$ and $A(6)$. These correspond to a multiple execution of task A. If enough of this executions are able to enter the *load window*, the load inside may increase due to an uneven distribution of load through the time.

So, as final remark, in order to execute a correct load analysis of a processor, it is imperative that the job responsible for generating the maximum load does not have tasks from actors that can be fired multiple times simultaneously.

6.3 Software implementation

Since most of work regarding fixed priority software implementation was already done from last chapter, the changes needed to study the concepts introduced in this chapter were minimal.

Most of the changes regarded the visualization tool, namely a processor view, i.e, a timing distribution chart per processor instead of per actor.

The data flow simulator already enabled us to perform simulations with two or more independent graphs simultaneously, so we only needed to create some functions that allowed us to process the data from each graph and perform an interference analysis.

This interference analysis is performed by simulating two independent graphs within the same time frame and alone in a separate platform. During this simulation, data regarding the start

and finishing times for every task is being stored in an appropriate structure and on a different list for each graph.

Finally, these two lists are merged into one single list taking into account the relative priority of each graph and assuming preemption. Task executions from the high priority graph are inserted whole and in their respective time while task executions from the lower priority graph are inserted only where they "fit", timewise, or even divided in case of preemption from a task with higher priority.

In the end, we get a list of activations that represent the timing behaviour of a processor running a high priority job and a lower priority one during the idle times from the high priority task.

6.4 Conclusion

In this chapter we introduced the concept of **load of a processor** so that we establish the necessary bases in order to study the task interference problem. We focused our study on the worst-case scenario in terms of processor usage, i.e, when the processor is subject to the maximum load from a certain task. We also introduced a process to achieve an instant were the load in the processor is maximum. Using this instant as an interference pattern for a low priority task, we are able to compute its worst-case execution time. To support our claims, we relied on the monotonicity concept to establish some ground rules regarding the behaviour of the load function when the tasks responsible for generating this load have variable execution times.

Chapter 7

Results

7.1 Analysis of intra-graph fixed priority data flow graphs

In chapter 5 we introduced various expressions as result from the theoretical deduction taken to model our simple fixed priority systems. In this chapter we intend to confront the models elaborated previously with simulation results and infer about their validity.

7.1.1 Data Flow analysis of a fixed priority graph

7.1.1.1 Worst Case response-time

For the Worst Case response-time of a set of actors mapped into the same processor and scheduled by a fixed priority scheme, as the one depicted in figure 5.1, we define the following expressions in chapter 5:

1. **High priority task**:

$$\hat{r}(A,k) = \tau_A, \quad k \in \mathbb{N}_0 \tag{7.1}$$

2. **Low priority task**:

$$\hat{r}(B,k) = \begin{cases} d_{BA} \cdot \tau_A + \tau_B & \text{if } k = 0 \\ \tau_A + \tau_B & \text{if } k > 0 \end{cases}, \quad k \in \mathbb{N}_0 \tag{7.2}$$

7.1.1.2 Analysis of the start times

As for the start times for the tasks, we reached that:

3. **High priority task**:

$$s(A,k) = \begin{cases} k \cdot \tau_A & \text{if } k < d_{BA} \\ k \cdot \tau_A + (k - d_{BA} + 1)\tau_B & \text{if } k \geq d_{BA} \end{cases} \tag{7.3}$$

4. **Low priority task**:

$$s(B,k) = (d_{BA} + k)\tau_A + \tau_B \cdot k, \quad k \in \mathbb{N}_0 \tag{7.4}$$

To test these expressions, we simulated a graph, structurally identical to the one depicted in figure 5.1, with the following parameters:

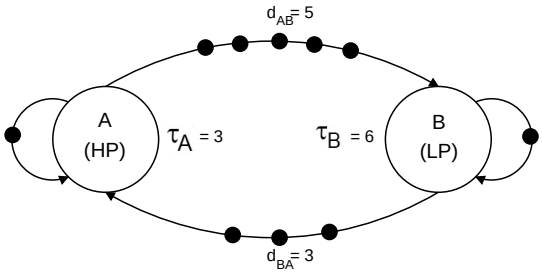

Figure 7.1: Two actor graph

7.1.2 Simulation results

The first two expressions indicated are upper bounds for the response-time of the tasks. In the simulations executed, we just verified if such bounds were respected, but unless we are able to came up with a concrete situation where these bounds may be violated, we assume that they are valid. For this particular situation, we are going to confront the simulation results with the projections obtained through expressions 7.3 and 7.4 and reach a conclusion regarding the validity of these formulas.

So, before we present the simulation results, we start by employing the *start time* equations for both tasks A and B to calculate the starting times of their first four executions. The results obtained are presented in the table below:

k	**s(A, k)** $(\tau_A = 3)$	**s(B, k)** $(\tau_B = 6)$	d_{BA}
0	0	9	
1	3	18	
2	6	27	
3	15	36	3
4	24	45	
5	33	54	

Table 7.1: Start times for the first six firings of each actor

Now for the simulation results. In order to simplify the analysis of the simulation results, we are presenting those in the form of a Gantt chart:

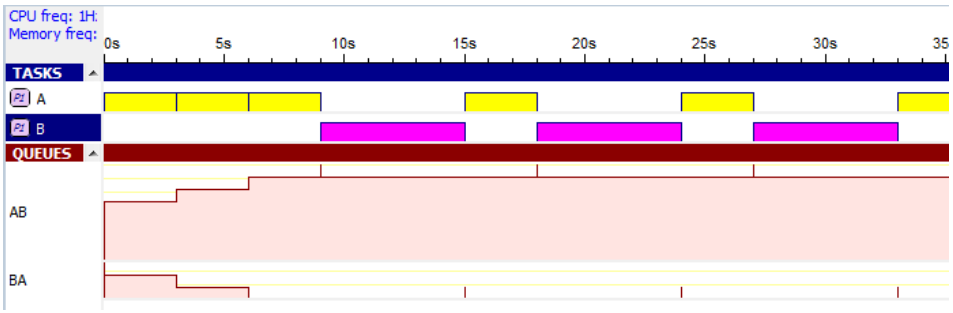

Figure 7.2: Gantt chart of the simulation results

As we can verify in the previous figure, the start times of the various firings of the actors were correctly calculated by the equations 7.3 and 7.4. We can also observe a pattern in the execution of the graph: in the beginning, only the high priority actor A is allowed to fire since it has some tokens in its incoming edge. Only after it finishes the execution of the last one of those, the low priority actor B is able to fire for the first time. These initial executions comprise the *transient phase*. After the first firing of actor B the *periodic phase* is initiated.

Regarding the response-times, we can also verify that the high priority task A always fires as soon as it has at least a token in its incoming edge, the BA edge. By observing the queue evolution of this edge along with the firings of this task, we are able to see that when a token is produced into this edge by the low priority task B, a firing from the task A always follows. On the other turn, task B can only fire when the BA edge is depleted of tokens. In this simulation, the response-time for the high priority task A is never over τ_A while the response-time of the low priority task B reaches is maximum in its first firing. Task B has to wait for three consecutive firings from A, which puts its response-time at $r(B,0) = d_{BA} \cdot \tau_A + \tau_B = 15$, which is coherent with expression 7.4 for $k = 0$.

7.1.3 Multi Processor mapping analysis

7.1.3.1 Worst case response-time

This section is similar to the previous one. We are only extending the study to an arrangement of four actors but now mapped in different processing platforms. The overall schematic of the system in study is identical to the one depicted in figure 5.2. The object of our study are still actors A and B. We are not going to perform the worst case response-time and start time analysis for the remaining actor of the graph. As before, in chapter 5 we defined the following expressions:

1. **High priority task**:

$$\hat{r}(A,k) = \tau_A, \quad k \in \mathbb{N}_0 \tag{7.5}$$

2. **Low priority task**:

$$\hat{r}(B,k) = (d_{DA} + d_{BD} \cdot \tau_A + \tau_B), \quad k \in \mathbb{N}_0 \tag{7.6}$$

7.1.3.2 Analysis of the start times - High priority task

Regarding the start times, we need to proceed with some caution. From this point on forward, the analysis of this type of graph becomes quite complex since it tends to fork into several sub cases due to all the possible relations between the execution times of two interconnected and adjacent actors but mapped in different processors. In the graph used as template for our study (figure 5.2) these actors are the high priority actor A and actor D. These two actors are mapped in different processing platforms, designated as $P1$ and $P2$, and are the higher priority actors in each processor. They also have an interdependence between each other: actor A depends on actor D to produce tokens into its incoming edge. Since both actors run on different platforms, they can be fired simultaneously, which originated different scenarios depending on the relationship between τ_A and τ_D.

7.1.3.2.1 $\tau_A > \tau_D$ In our first case study, we assume that the actor producing tokens (D) has a shorter execution time than the actor consuming its produced tokens (A). The question that was formulated when this case was first discussed was: is it possible to have actor A blocked due to the lack of tokens in the DA edge while actor D has tokens of its own in its incoming edge BD?

After the appropriate analysis, we reached expression 5.19, here repeated for reader's convenience:

$$s(D,n) = (n-1)\cdot\tau_A, \quad n = \tau_A/\delta t \tag{7.7}$$

where $\delta t = \tau_A - \tau_D$, i.e, the time difference between the two actors execution times. In this context, n represents the number of delays δt that compose a single activation of the faster actor A. We proved that as long as $\delta t \geq 0$, actor A is never blocked due to the lack of tokens produced by actor D. In fact, if $\delta t > 0$, the number of tokens in the incoming edge of actor A at any given point increases with time. In order to verify this assumption, we simulated a system identical to the one depicted in figure 5.2 with the following characteristics:

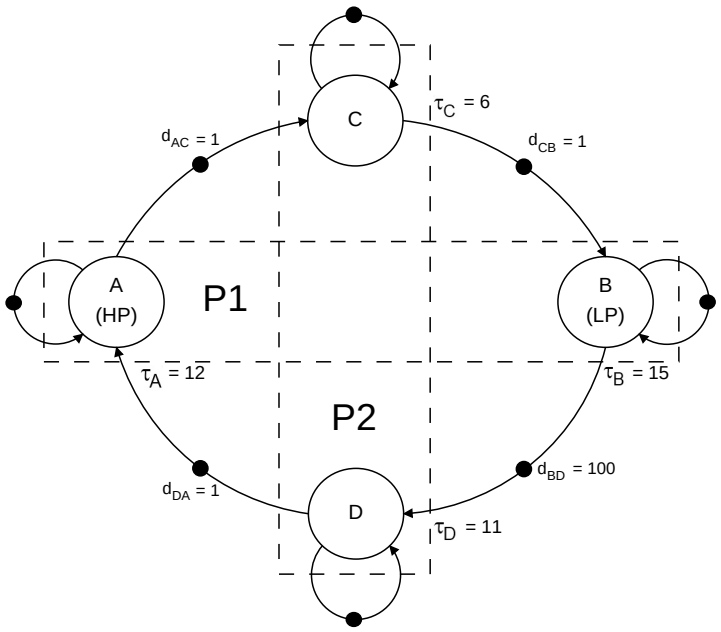

Figure 7.3: Temporal characteristics of the graph in study

The graph depicted in the previous figure was simulated during 1000 time units. The results of the simulation are in the next figure:

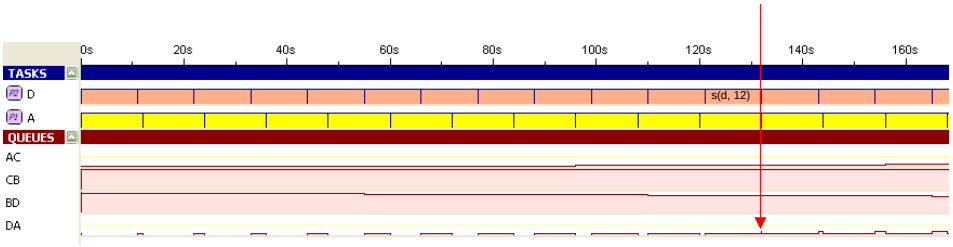

Figure 7.4: Gantt chart relative to the temporal simulation of the graph in figure 7.3

There are several interesting conclusions to retrieve from the previous figure:

- As expected, actor A is able to maintain consecutive executions without being blocked by running out of available tokens in its incoming edge.
- Both actors A and D run uninterrupted as long as the incoming edge of actor D has tokens, as it was expected, since both actors retain the highest priority in their respective processors.
- For this situation we have that $\delta t = \tau_A - \tau_D = 12 - 11 = 1$, hence, $n = \tau_A / \delta t = 12/1 = 12$. This means that after 12 iteration of actor D, this actor is able to produce an "extra" token into the DA edge. This is observable in the figure: after the 12^{th} iteration of actor D, both active actors "synchronize" their start times and one more token appear in the DA edge.
- By observing the evolution of the incoming edge of actor A, we can see that the rate of which tokens are being consumed from it is lower than the rate at which tokens are being produced into it. This leads to the situation that we already mentioned: the number of tokens in the DA edge will grow as long as actor D remains unblocked.

7.1.3.2.2 $\tau_A = \tau_D$ This is perhaps the easiest case to analyse. If both high priority actors have the same execution time and plenty of tokens in its incoming edges, they will fire continuously and synchronously as long as this situation is maintained. In order to verify this assumption, we made just a small change in the graph simulated in the previous section:

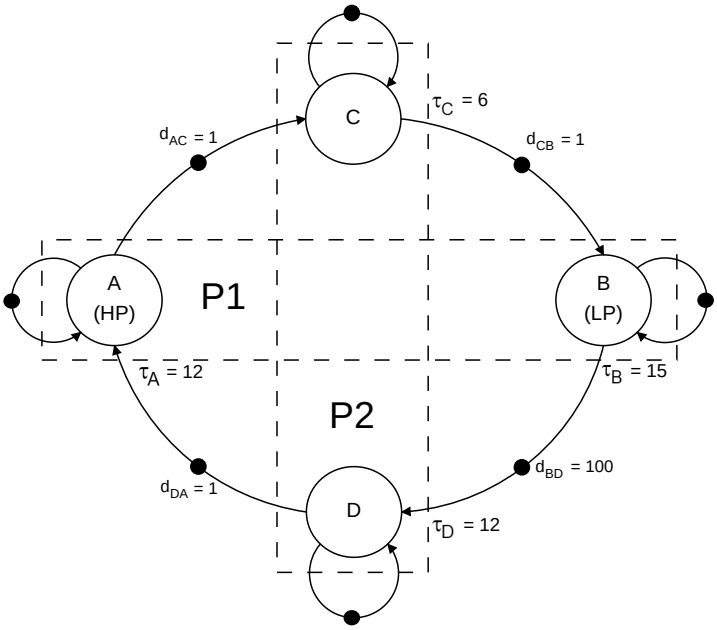

Figure 7.5: Graph to be used in the next simulation with $\tau_A = \tau_D$

The simulation graph has now $\tau_A = \tau_D$. Running a simulation with the same parameters as before, we obtained the following results:

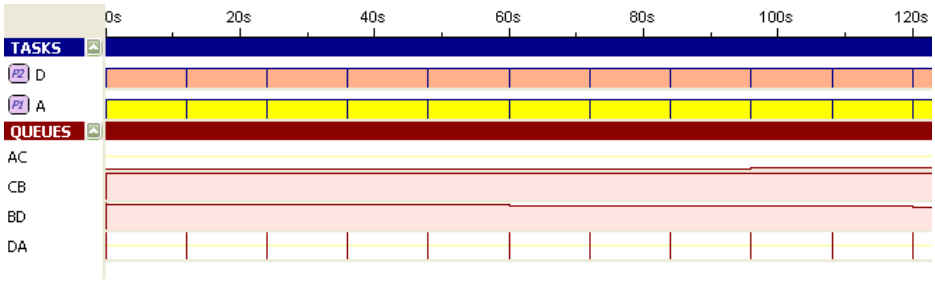

Figure 7.6: Simulation results for the case with $\tau_A = \tau_D$

As we can conclude from the simulation results, the firings of both actors are synchronized. As such, the number of tokens in the incoming edge of actor A is maintained. In reality, this case should be presented just as a special case from the previous section, i.e, we can obtain these same results from the expressions introduced before if we consider $\delta t = 0$.

7.1.3.2.3 $\tau_A < \tau_D$ In this case we have the consuming actor A executing faster than the producing actor D. The question that we want to answer in this section is: how long can the high priority actor A fire continuously before being blocked due to lack of tokens in its incoming edge? This question is important because as long as actor A is able to fire in a continuous fashion, the low priority actor that is mapped in the same processor, actor B, cannot fire. Considering this, we can reformulate our question: how long does the low priority actor need to wait before it can fire for the first time (even if its only a partial execution due to a possible preemption from the high priority actor)?

To answer that, we devise the iterative expression 5.30 to determine this time period:

$$d_{act}(A,k) = \left\lfloor \frac{\tau_A \cdot (d_{in_edge} + d_{act}(A,k-1))}{\tau_{prodActor}} \right\rfloor \tag{7.8}$$

As before, we are going to change the temporal characteristic of the graph to be used in the simulation in order to address this case:

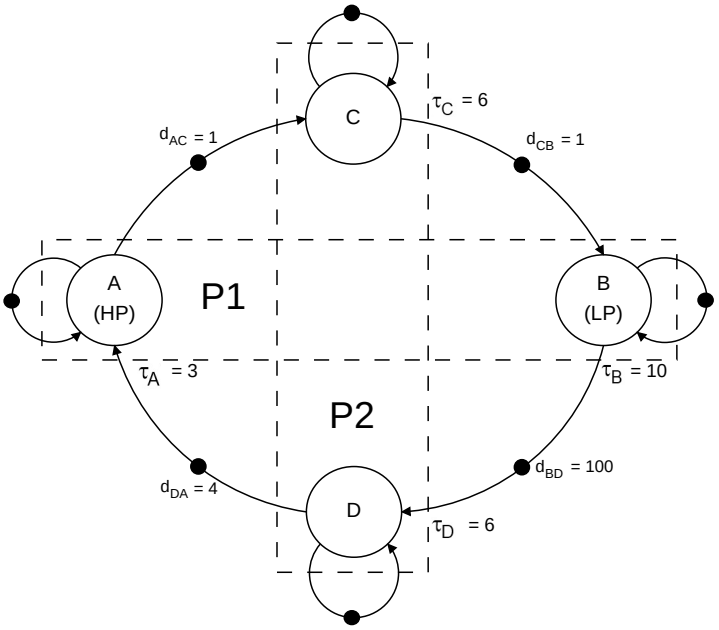

Figure 7.7: Temporal characteristics of the simulation graph

As we can see, in this graph we imposed $\tau_A < \tau_D$, a large number of tokens in the incoming edge of actor D, so that it never blocks due to lack of tokens. Also, it is important to point out the fact that, since our simulator does not support preemption, once the lower priority B is allowed to fire, it will execute until it finishes, actively blocking the high priority task A in the process.

First, lets start by predicting the time at which the first firing of actor B is going to occur. For that, we use expression 7.8 with the values from figure 7.7:

$$d_{act}(A,0) = \left\lfloor \frac{\tau_A \cdot (d_{DA} + d_{act}(A,-1))}{\tau_D} \right\rfloor \tag{7.9}$$

Since $d_{act}(A,k) = 0 \forall k < 0$, we have that:

$$\begin{aligned} d_{act}(A,0) &= \left\lfloor \frac{3 \cdot (4+0)}{6} \right\rfloor \\ &= \left\lfloor \frac{12}{6} \right\rfloor \\ &= 2 \end{aligned} \tag{7.10}$$

Next iteration:

$$\begin{aligned} d_{act}(A,1) &= \left\lfloor \frac{\tau_A \cdot (d_{DA} + d_{act}(A,0))}{\tau_D} \right\rfloor \\ &= \left\lfloor \frac{3 \cdot (4+2)}{6} \right\rfloor \\ &= \left\lfloor \frac{18}{6} \right\rfloor \\ &= 3 \end{aligned} \tag{7.11}$$

Since $d_{act}(A,1) \neq d_{act}(A,0)$, we continue to the next iteration:

$$\begin{aligned} d_{act}(A,2) &= \left\lfloor \frac{\tau_A \cdot (d_{DA} + d_{act}(A,1))}{\tau_D} \right\rfloor \\ &= \left\lfloor \frac{3 \cdot (4+3)}{6} \right\rfloor \\ &= \left\lfloor \frac{21}{6} \right\rfloor \\ &= 3 \end{aligned} \tag{7.12}$$

$d_{act}(A,2) = d_{act}(A,1)$: the algorithm has converged. From this result we know that actor A is able to process 3 more tokens along the 4 initial ones before being blocked by the exhaustion of tokens from its incoming edge. So, in total, actor A is able to perform 7 consecutive fires. Now we can use expression 5.32 to determine the time instant in which actor B fires for the first time:

$$\begin{aligned} s(B,0) &= (d_{actConvergent}(A,k) + d_{DA}) \cdot \tau_A \\ &= (3+4) \cdot 3 \\ &= 21 \end{aligned} \tag{7.13}$$

All its left now is using the simulator to verify this result. The simulation Gantt chart of the graph depicted on figure 7.7 is:

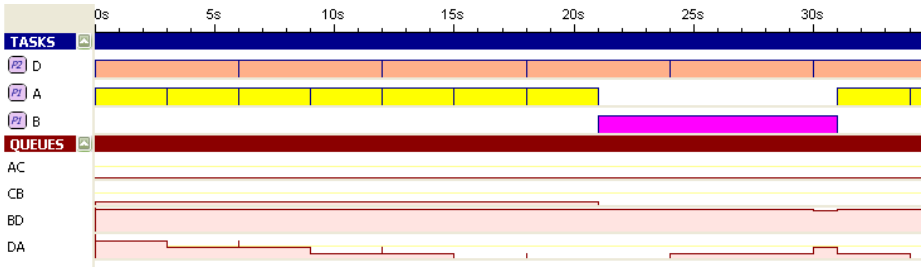

Figure 7.8: Simulation results for the graph in figure 7.7

We can verify from the last figure that our projections were correct: actor D fires continuously due to the large number of tokens in the BD edge, actor A fires seven consecutive times before being blocked due to the lack of tokens in its incoming edge, and finally, actor B fires for the first time right after actor A gets blocked, exactly at $t = 21$, as predicted by our algorithm.

7.2 Load analysis of a Wireless LAN and a TDSCDMA job

7.2.1 Theoretical approach

In this section we intend to study the task interference effects on a low priority job due to a high priority job whose actors are mapped on the same processing platform. Most of our fixed priority jobs run concurrently with other jobs with higher or lower priorities. If all those jobs share the same poll of resources, high priority jobs interfere with the execution of low priority jobs. The main consequence of such interference is a delay on the response-time of the tasks that compose the job being interfered, due to some of these tasks being preempted by high priority ones.

Concretely, we are going to study the effects that a high priority but with a relatively short running time job, such as a WLAN job, creates in a low priority long execution time job, such as TDS-CDMA job.

As before, we are going to perform a worst-case scenario analysis so that we can derive an upper bound for the execution time of the high priority job. As such, we are going to subject the low priority task to a *maximum load* from its concurrent high priority job. For that, we are going to simulate a situation where these two jobs are set to execution at the same time and using the same set of resources. The simulation graph will have a setup that allows it to simulate a situation of maximum load due to the high priority task. As so, the low priority task is going to be subject to the maximum interference possible. From this simulation we obtain the worst-case running time for the tasks that compose the low priority job. We then simulate a solo execution of the low priority job but using the worst-case execution times for the tasks that we obtained in the previous step. Finally, we compute the **Maximum Cycle Mean** of the new graph using the appropriate tool from the Heracles tool set.

If this modified graph has a different MCM than the version with the default execution time values of it, we know that the increase of the execution time of the tasks has originated another critical cycle in the graph. In these two jobs, the pace of the graph is always dictated by a critical cycle formed by the various sources. This is vital for the correct functioning of the system since this means that the sources are outputting data at a lower rate than the same data is consumed, avoiding the exhaustion of buffer space, which could lead to a waste of unconsumed tokens.

In conclusion, as long as the MCM of the low priority job remains unchanged, we can conclude that the increase in the execution time of the tasks is not going to affect the job performance.

7.2.1.1 Simulation graphs

For our experiments, we have chosen two graphs from real life applications: a graph modelling a **Wireless Local Area Network 802.11a receiver** job:

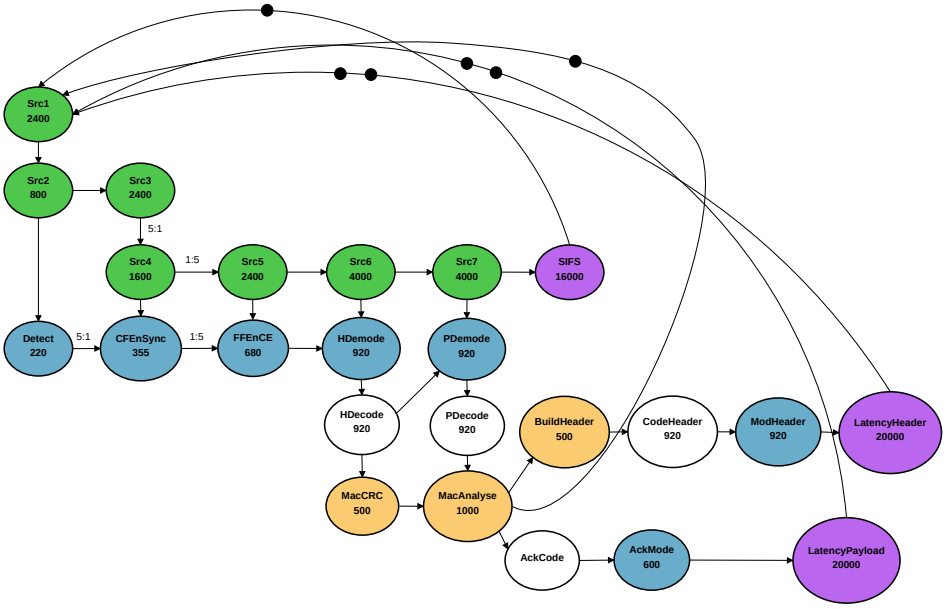

Figure 7.9: A Wireless LAN 802.11a receiver job

and a **Time Division Synchronous Code Division Multiple Access (TDSCDMA)** job:

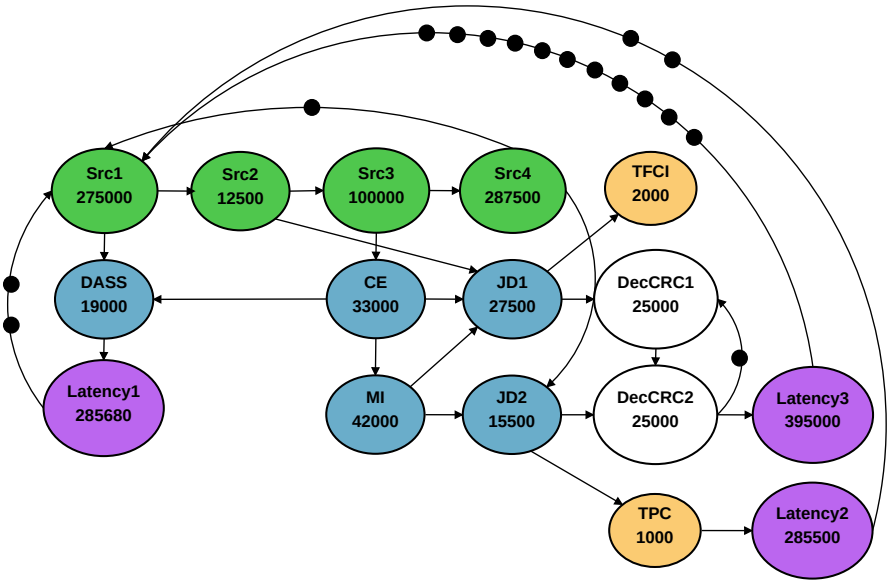

Figure 7.10: A TDSCDMA job

The execution times, in CPU cycles, of each task that compose the jobs are indicated inside the representation of the actor. The number of initial tokens in each edge is represented explicitly, i.e, each token is represented by an individual circle over the edge, instead of our normal representation in which we condensate all the tokens into one circle and indicate the actual value outside. A color scheme is used to identify the processing platforms in which the actors are

mapped: The blue actors are mapped in the **EVP** processor, the orange actors in the **ARM** and the white actors are processed by the **Software Codec**. The green actors represent **sources** of data that operate independently and the purple actors represent **latencies**.

As we can see by observing the execution times of the various actors, the WLAN job is significantly shorter than the TDSCDMA job, although the latter has a low number of actor and dependencies. In our simulations, we are going to run both jobs simultaneously and independently on each other, but both affected by a *delay* actor imposing the same delay to both graphs. This *delay* actor is going to be directly connected to the first actor of the EVP cycle from both graphs: the **DETECT** actor in the WLAN job and the **CE** actor in the TDSCDMA job.

7.2.1.2 Implementation of the interference between jobs

The simulator tool allows us to run to independent graphs simultaneously but it does not permit that actors from two independent graphs to be mapped into the same processing platform. In order to simulate the interference between the tasks we had to develop a separate tool for that.

The previous simulation is still necessary because it will provide us with a list of executions, in ascending order of start times, for the two graphs. These lists are then used by a set of function to calculate the effects of the interference from the high priority job into the low priority one.

This tool merges both lists into a single one using the following criteria:

- The start time and duration of the high priority execution remains unchanged.
- The low priority executions are fit into the idle times of the processor. For that, most of the executions have their start times delayed and are preempted several times to fit into the available time.

This procedure is repeated for a single execution of every actor from a set mapped into a given processor and for every processor used in that job. By doing this, we subject every actor to the maximum load attainable in that processor, thus obtaining the **worst-case execution time** for every actor.

Finally, with the set of worst-case execution times for a certain job in our possession, we run another solo simulation of the low priority job, the TDSCDMA job, but using these worst-case values instead of the default ones and infer about the influence of this increase in actor execution times and the pace of the graph.

7.2.2 Simulation results

7.2.2.1 Context switching time = 0 cycles

For our first experiment we decided to simply simulate both the WLAN and the TDSCDMA jobs, considering an ideal context switching time of zero, and then simply merge the respective actor executions lists. The worst-case execution times for all the actor mapped on the EVP, ARM and Software Codec platforms are indicated in the next table:

The "normal" column contains the default values for the execution time of the actors from the TDSCDMA job mapped in the processor indicated, i.e, the execution times that those actors would have if the job was mapped alone in the processing platform (refer to figure 7.10). The "worst case" column shows the execution times that these same actors displayed when subjected

Processor	Tasks	Execution Time		Increase
		normal	worst-case	
EVP	CE	33000	45070	+36,58%
	DASS	19000	28630	+50,7%
	MI	42000	54290	+29,3%
	JD1	27500	39570	+43,9%
	JD2	15500	24210	+56,2%
ARM	TFCI	2000	4000	+100,0%
	TPC	1000	1500	+50,0%
SWC	DECODECRC1	25000	30200	+20,8%
	DECODECRC2	25000	30200	+20,8%

Table 7.2: Simulation results without considering context switching time

to the maximum interference from the WLAN high priority job that was mapped in the same processing platform.

As we can observe by the increase column, the increment in the execution times measured is quite substantial in most cases.

Figure 7.11 displays the results obtained in the form of a Gantt chart:

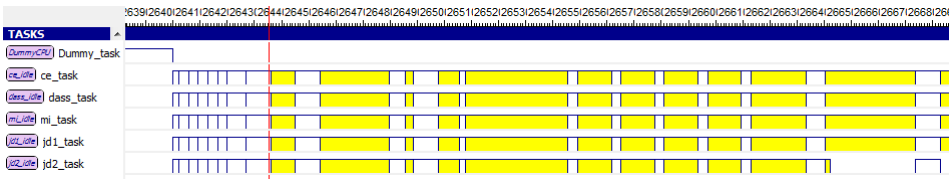

Figure 7.11: Gantt chart for the interference simulation on the EVP processor tasks

The high priority executions are represented by the white bars while the low priority executions use yellow in its representation.

Next, we are going to analyse if this rise in the execution times has any influence in the pace of the graph, i.e, we are going to verify if, due to this increase, there was a change in the critical cycle of the low priority job.

For the TDSCDMA job, the execution pace of the graph is dictated by the critical cycle formed by all the sources:

$$C_{critical} = [src_1, src_2, src_3, src_4] \tag{7.14}$$

This feature is important since it implies that no other cycle in the graph executes at a higher pace than the sources and so, it guarantees that all the tokens originated from the sources are consumed at a higher rate than they are produced, which also guarantees that the output edges of the sources never get completely full. This could lead to potential loss of data since new tokens originated from the sources had to be discarded due to the lack of space in the outgoing edges.

So, to make sure that our job remains consistent, we have to verify if the critical cycle remains the source cycle. A simple process to ascertain this is to compute the **Maximum Cycle Mean** of the graph before and after we insert the execution time changes. Fortunately, the Heracles tool set has a functionality to compute the MCM of a Single Rate Data Flow graph, which simplifies our analysis since this graph is somewhat complex.

The Heracles tool uses the Howard algorithm to compute the MCM of an SRDF graph. A more detailed explanation of this particular algorithm can be found in [10]. The MCM for the TDSCDMA job in its default configuration is:

$$\mu_{TDSCDMA} = 675000 \quad \text{cycles} \tag{7.15}$$

Running a new simulation of the TDSCDMA job but using the values of the worst-case column as execution times, we obtained the following MCM for the altered graph:

$$\mu_{interference} = 675000 \quad \text{cycles} \tag{7.16}$$

The MCM for both graph is the same which means that the critical cycle has not changed with the increase in the execution times: the sources cycle still dictates the pace of execution of the graph.

7.2.2.2 Context switching

Next we intend to push our simulations further into more realistic scenarios. The next logical step is to take into consideration context switching. Whenever a given processor is busy processing an execution and this execution is preempted by another one with higher priority that became ready in the meantime, there are a series of steps that need to be executed before putting the high priority pre-empting task into execution. Namely, the system must take the appropriate measures in order to allow the preempted task to resume execution later, from the point where it was interrupted. This procedure is known as context switching and consists in storing the current state (registry contents, cache, variables etc.) of the interrupted task so that it can be restored later on. This is a costly operation, with an execution time that vary from processor to processor.

So far, in our simulations we neglected this operation, assuming that we were using an ideal processor capable of switching and resuming preempted tasks in a time that it is negligible in comparison with the task's execution times. To approach our model to a more realistic one, we are going to include this context switching operation in our simulation. But first is important to define exactly in which situations we employ such an operation and its main characteristics.

The description above refers to the cases where a running task is interrupted by another one. But whenever a processor finishes a task and starts a different one, there is also a change in context, which also implies that some maintenance procedures need to be done before running the next execution. However, since the execution model is stateless, this type of context change is significantly shorter then the one referred before, and, for simplicity, we are going to assume that the execution time of the tasks considered has this context switching time implicit.

Taking this into account, we are only going to consider the existence of a context switch delay whenever a running execution is preempted by another one. This context switch is internally represented as a typical task activation, with a predefined and constant execution time, which is defined in the beginning of the simulation.

We are also assuming that the context switch is started by the interrupting task and, as such, there is as an extra time that this task has to deal with. So, whenever an interrupting task issues a context switching operation, the resulting delay is propagated through the remaining executions. The next figure exemplifies this procedure:

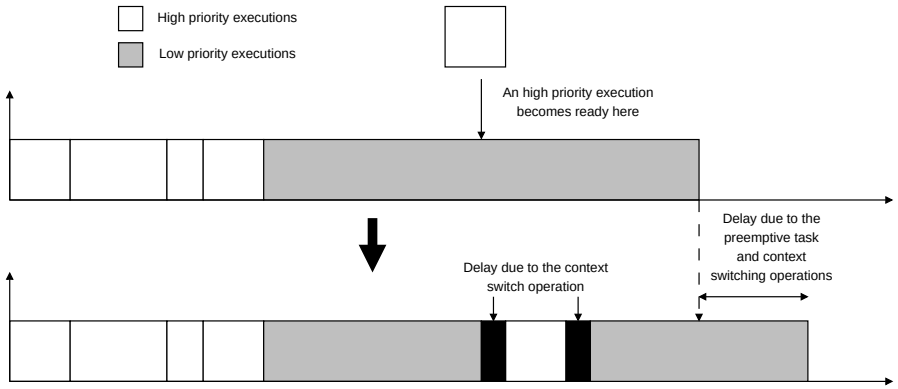

Figure 7.12: Example of the functioning of the context switch operation

As we can observe from this figure, the insertion of a context switch operation (context store) before the preemption imposes a delay in it and the other context switch operation (context restore) after this execution imposes a second delay on the finishing time of the interrupted executions as in all the executions from this point on forward.

7.2.2.3 Simulation using a context switching = 200 cycles

Since we verified in the previous section that the increase in the execution times of the actors, due to task interference, does not destabilize the low priority job, we are now ready to take a step further in our analysis depth by including context switching delays whenever a low priority execution is preempted by a high priority one.

For now, we start with a relatively small value for the duration of the context switch operation. We altered the simulator code so that we could specify this duration as a command line parameter for an easy implementation of this feature.

In our first simulation considering context switching, we used a context switch delay of 200 cycles. The simulation results are presented in the following table:

Processor	Tasks	Execution Time		Increase	Context switch increment
		normal	worst-case		
EVP	CE	33000	49870	51,1%	10,7%
	DASS	19000	31310	64,8%	9,4%
	MI	42000	59490	41,6%	9,6%
	JD1	27500	44370	61,3%	12,1%
	JD2	15500	26730	72,5%	10,4%
ARM	TFCI	2000	4400	120,0%	10,0%
	TPC	1000	1500	50,0%	0,0%
SWC	DECODECRC1	25000	31800	27,2%	5,3%
	DECODECRC2	25000	31800	27,2%	5,3%

Table 7.3: Simulation results when considering a context switching time of 200 cycles

We can see a considerable increase in the worst-case execution times of most tasks. The TPC task in the ARM processor is not affected by the insertion of context switch executions,

something that we could expect of such a short task.

The following figure shows the Gantt chart of the simulation results for this case:

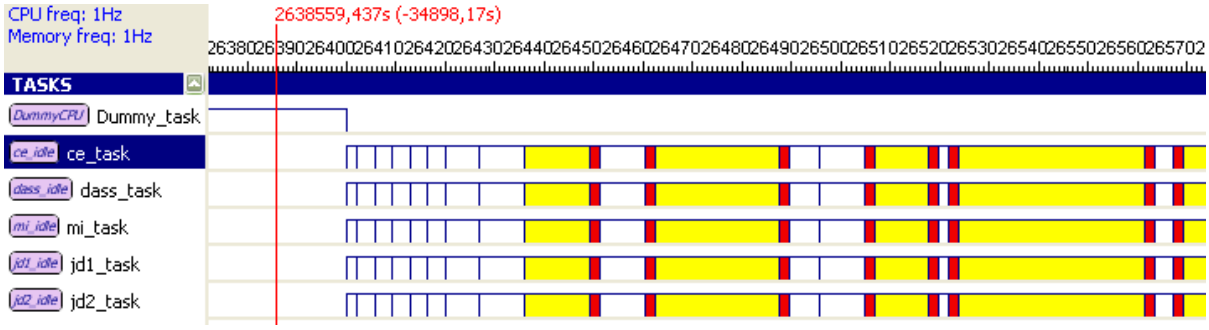

Figure 7.13: Simulation results for the EVP tasks when a context switching time of 200 cycles is considered

The context switch time is marked in red while the high priority executions are still represented in while while the low priority execution uses yellow.

As before, we used the worst-case execution times indicated on table 7.3 to run a solo simulation of the TDSCDMA job with a MCM computation for the respective graph. The MCM tool returned that

$$\mu_{ctxsw=200} = \mu_{TDSCDMA} = 675000 \quad \text{cycles} \tag{7.17}$$

As before, the extra delay due to context switching was not enough to disrupt the pace of execution of the graph.

Since 200 clock cycles is considered a small value for context switching, we are going to extend our study by increasing this value and repeating the previous process.

7.2.2.4 Simulations using a context switching of 500 and 1000 cycles

After running two more simulations using a context switch delay of 500 and 1000 cycles, we obtained the following results:

Processor	Tasks	Execution Time		Increase	Context switch increment
		normal	worst-case		
EVP	CE	33000	57070	72,9%	26,6%
	DASS	19000	36710	93,2%	28,2%
	MI	42000	67290	60,2%	23,9%
	JD1	27500	51570	87,5%	30,3%
	JD2	15500	31530	103,4%	30,2%
ARM	TFCI	2000	5000	150,0%	25,0%
	TPC	1000	1500	50,0%	0,0%
SWC	DECODECRC1	25000	34200	36,8%	13,2%
	DECODECRC2	25000	34200	36,8%	13,2%

Table 7.4: Simulation results when considering a context switching time of 500 cycles

Figure 7.14 shows the Gantt chart of the simulation results for the ARM processor executions with the 500 cycles context switch:

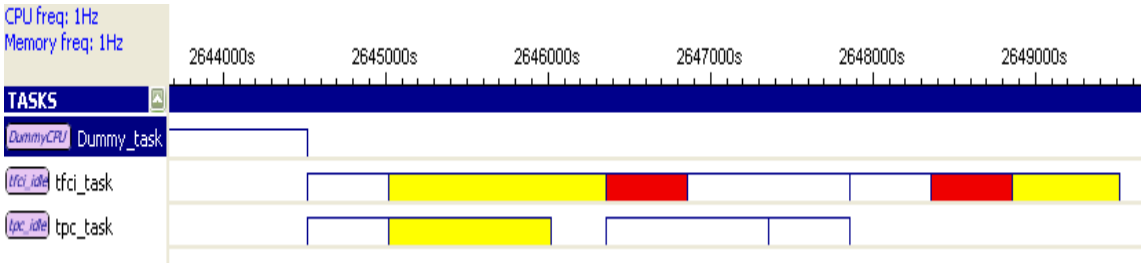

Figure 7.14: Simulation results for the ARM tasks with a context switching time of 500 cycles

Processor	Tasks	Execution Time		Increase	Context switch increment
		normal	worst-case		
EVP	CE	33000	63070	91,1%	39,9%
	DASS	19000	39710	109,0%	38,7%
	MI	42000	74290	76,9%	36,8%
	JD1	27500	57570	109,3%	45,5%
	JD2	15500	33530	116,3%	38,5%
ARM	TFCI	2000	6000	200,0%	50,0%
	TPC	1000	1500	50,0%	0,0%
SWC	DECODECRC1	25000	38200	52,8%	26,5%
	DECODECRC2	25000	38200	52,8%	26,5%

Table 7.5: Simulation results when considering a context switching time of 1000 cycles

The Gantt chart of simulation results for the SWC while considering a context switch delay of 1000 cycles is presented in the figure 7.15:

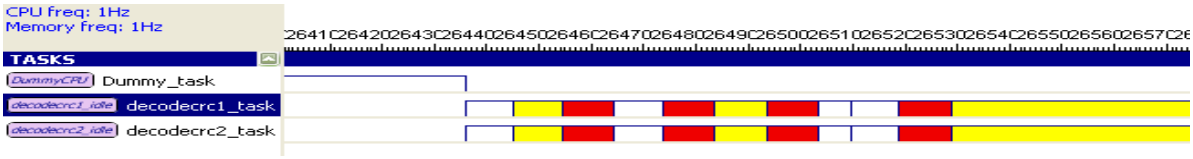

Figure 7.15: Simulation results for the SWC tasks with a context switching time of 1000 cycles

The MCM computation tool returned the following results for these two cases:

$$\mu_{ctxsw=500} = \mu_{ctxsw=1000} = \mu_{TDSCDMA} = 675000 \quad \text{cycles} \tag{7.18}$$

As with all the previous situations, the delay increase was not sufficient to alter the rate of execution of the TDSCDMA graph.

7.3 Conclusion

The results obtained in this chapter were satisfactory. For the models introduced in chapter 5, we were able to verify the validity of the expressions. Even if the graphs analysed in this

chapter were not very complex, the models for the worst-case response-times and start times were consistent with the simulation results.

As for the results acquired from the task interference study, we can use them to support the pertinence of the fixed priority schedulers as efficient schedulers, when compared with TDM schedulers, for instance. We were able to verify that, in a worst-case scenario, if two concurrent jobs were scheduled using fixed priority, they were able to meet their deadlines, even considering context switching delays relatively large. Unfortunately our analysis is limited to only two graphs operating concurrently. Also, we only analyse two concrete examples of these jobs: a high priority and relatively short one and another one with a low priority and longer execution time, which limits our scope of applications.

Chapter 8

Conclusion and future work

Fixed priority scheduling is a type of scheduler that has a simple implementation and has a set of characteristics that make it an attractive solution, specially to systems that include one or more critical tasks.

If we think about all the embedded systems that are present in our actual lives, we can easily pinpoint some that perform certain tasks that are either vital to the system or to the user, sometimes even both. We have cars whose breaking, traction and a myriad of security related systems are controlled by a system that employs some kind of scheduler. Medicine nowadays rely in a collection of apparatus that provide monitoring and support to human lives. Power plants, factories and even laboratories use electronic embedded systems to control processes that can potentially harm workers and structures if they become unstable. Variables such as temperature, pressure, humidity, etc, must be closely monitored and, in case of deviation from their secure values, a control process must be activated as soon as possible to correct that deviation and avoid a possible disaster.

All these situations exemplify systems have one thing in common: one or several critical tasks. These tasks must have their deadlines met under the possibility of severe penalties if not. As such, they benefit greatly from a fixed priority scheduler, specially if the system processor allows preemption. With a non preemptive processor, the advantages of this scheduler towards other scheduling policies are not that clear. In this case, the advantage of using a fixed priority scheduler over any other scheduler are dependent on the execution times and rates of execution of the remaining tasks in the system. If for instance, the system possesses a task with a long execution time, the worst case response-time for any critical tasks might be too long, since there is always the possibility of the critical tasks becoming ready to execute just after the processor started the execution of the long task.

Given the usefulness of this type of schedulers, it is important to perform a thorough analysis in order to fully comprehend them and establish solid mathematical models.

One of the objectives with this project was precisely the elaboration of these models using data flow concepts. Although we were able to perform a concise analysis of some simpler cases, the complexity attained while implementing this type of scheduler into data flow graphs made the creation of a functional model quite difficult.

The second part of our project retained the fixed priority scheduler as a focal point but migrated to another aspect of it: the interference between tasks scheduled into the same processing platform but with different fixed priority values. This is an interesting subject to study, specially

considering that are plenty of computational systems around us that implement this type of schedule. The focal point of our study was to determine the worst case response-time of a low priority task mapped into a processor along with a high priority task, and thus, suffering from interference from the latter, in the form of blocking through preemption. This analysis required us to define some base concepts, such as *computational load* and *load window* of a processor, which revealed to be quite important and with some promising research ahead. With these concepts well defined, we are now able to determine the worst case response-time of what we designed as *secondary tasks*, i.e, tasks with lower values of priority that suffer from interference from another task with higher priority that share the same platform. If we are able to establish an upper bound for the response-time of a given task, we are also able to infer about the ability, advantages and disadvantages of using a fixed priority scheduler in detriment of other schedulers. In the simulations run in chapter 7, we determined that, for the concrete applications simulated, the fixed priority scheduler was more efficient, in term of processor utilization and overall execution time of the applications graphs, than the TDM scheduler.

Although we were able to achieve some interesting results with this project, there is still room for improvements in this subject. As suggestions for future work we would like to refer the following topics:

- **Expand the study of fixed priority graphs to more complex systems**: In this project we dealt with somewhat simple models, namely a basic two actor graph mapped into the same platform and a four actor, two processors system. It should be interesting to check if some of the models derived to the systems mentioned can be applied to more complex graphs or if there is need to define new models to accommodate the increase in complexity.

- **Include preemption into the simulator tool**: The current version of the simulator does not contemplate the use of preemption in the fixed priority simulations. Although we were able to include it through some post processing, namely through a set of functions that took lists of already simulated executions and rearrange them, including preemptive effects in the process, the native code of the simulator does not have the capabilities to implement preemptive action between the simulated tasks. This capability should be inserted in a way that it could be activated or not through an option passable via command line.

- **Include a scale factor or floating point calculation capabilities in the simulator**: As it is right now, the data flow simulator does all of its processing using only integer values. As such, the graphs to simulate must have all of their characteristics defined through integer values, namely, the execution time of their actors. This fact introduces a small granularity factor in the simulations, specially if we work with actors with short execution times. One possible solution to this case is simply augment the execution time of all the actors by a scale factor. We are still working with integer values but at least we have a greater granularity in the execution times. But this is not a practical solution, specially for complex graphs with plenty of actors. As a solution, we can create the possibility of activate and define a scale factor through the command line, which would then be used to adapt the execution time of the actors in the graph. The execution time of these elements could even be defined with floating point quantities, provided that after the application of the scale factor, the value gets converted into an integer. Another possible alternative, although with a considerable complexity of implementation, is the introduction of floating point operations

in the simulator. The simulator is a complex tool that depends on several external modules to function. Any changes in this sense to the simulator files should be also applied to all of its dependent modules. And since OCaml, the programming language in which the simulator was written, is a strongly typed language, this task promises to be difficult.

- **Expand the task interference study to the coexistence of three or more tasks in the same processor**: The task interference study presented in chapter 6 deals with only two tasks mapped in the same processor at a time: a low priority one that tries to execute subject to the interference of a high priority task that is able to preempt it. We were able to get some interesting results from this study, so it is logic to expand this idea to more tasks, with different priorities obviously, mapped into the same platform. We were able to generalize the function that computes the interference between tasks to deal with more than two concurrent tasks at a time. The interference is computed in a cumulative fashion, i.e, first is calculated the interference between the high priority task and the task with the next higher priority. From this operation we obtain a list of executions relative to this interference. We then assume this resulting list as the executions list of the high priority task and repeat this interference computation with the next task in line and so on until all the tasks executions are merged into one final execution list. Unfortunately we were not able to analyse this situation long enough to obtain significant results.

- **Conceive formal proofs for the conjectures presented in the task interference study**: We needed to establish some ground concepts before starting inferring properties about the systems approached in the task interference study, namely, about the computation load, maximum load and load window concepts. In order to support the results attained in this chapter, we need to provide some solid formal proofs of these properties.

- **Introduce all the context switching operations in the task interference simulations**: In chapter 7, in order to approximate our simulations to more realistic scenarios, we introduced the context switching operation that takes place whenever a running task is preempted by another task with higher priority. In this chapter we only considered the existence of context switching in the event of a preemption. But we know that whenever a processor finishes executing a certain task and starts executing another, there is always the need to establish a proper context. Since the previous task is terminated, there is no need to store all the information needed for it to be resumed in the future, and this type of context switch operation is significantly shorter than the previous one. But if we pretend to make our study more realistic possible, this smaller context switching delay should also be taken into account.

- **Determine the limit for the context switch duration in the task interference study**: This is an interesting aspect to investigate, mostly to find the limits of our fixed priority scheduler than anything else. In chapter 7 we experimented with increasingly higher values of context switch delays to see at which point we were able to compromise the rate of execution of the graphs analysed. All the context switch delays tested was insufficient to destabilize the graphs simulated. It would be interesting to check how far can we go, in terms of context switching delay applied, until the deadlines of one of the graphs simulated are violated. The problem is, the way the context switching is applied, the executions of both high and low priority graphs are affected. The effect of this delay in the low priority

graph is easily verified but as for the high priority graph, some additional computations need to be performed before we can conclude if this graph can meet its deadlines while dealing with such delays.

All the suggested topics provide some interesting research suggestions.

The data flow approach for the fixed priority scheduler revealed to be a fruitful area of research. Although this is a popular scheduler in actual systems and there is significant research published from a classical real-time theory point of view, the investigation from more modern perspectives is still very limited. With this project we hoped to take some significant steps towards the formulation of complete data flow model of fixed priority schedulers.

Bibliography

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, September 1993.

[2] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: A historical perspective. *Real-Time Systems*, 8:173–198, 1995.

[3] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers*, 52:933–942, July 2003.

[4] Jean-Yves Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Transactions on Information Theory*, 44, May 1998.

[5] Joseph Buck. *Scheduling dynamic data flow graphs with bounded memory using the token flow model.* PhD thesis, University of California, Berkeley, September 1994.

[6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.

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

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* McGraw-Hill, 2001.

[9] Raymond Cunninghame-Greene. *Minimax algebra in Lecture Notes in Economics and Mathematical Systems.* Springer-Verlag, 1979.

[10] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactionson Design Automation of Electronic Systems*, 9(4):385–418, October 2004.

[11] Lui Sha et al. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

[12] Marco Bekooij et al. Dataflow analysis for real-time embedded multiprocessor system design. *Dynamic and Robust Streaming in and between Connected Consumer Electronic Devices*, 3:81–108, 2005.

[13] A.H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency minimization for synchronous data flow graphs. *10th Euromicro Conference on Digital Systems Design Architectures, Methods and Tools*, pages 189–196, August 2007.

[14] Michael Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh lcf: a mechanized logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[15] Arne Hamann, Rafik Henia, Razvan Racu, Marek Jersak, Kai Richter, and Rolf Ernst. Symta/s - symbolic timing analysis for systems. 2004.

[16] Rafik Henia, Arne Hamann, Razvan Racu, Marek Jersak, Kai Richter, and Rolf Ernst. System level performance analysis - the symta/s approach. *IEE Proceedings online*, 20045088, 2004.

[17] Jason Hickey. Introduction to objective caml. Submitted to publication by Cambridge University Press, 2008.

[18] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. *Proceedings IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[19] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[20] Edward A. Lee. A denotational semantics for dataflow with firing. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkley, January 1997.

[21] Edward A. Lee and David G. Messerchmitt. Synchronous data flow. *Proceedings of the IEEE*, 1987.

[22] Xavier Leroy. *The OCaml System - Documentation and User's Manual*, 2011.

[23] Joseph Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[24] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20:46–61, January 1973.

[25] Jane W. S Liu. *Real-Time Systems*. Prentice Hall, 2000.

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

[27] Orlando Moreira, Marco Bekooij, J. Mol, and J. van Meerbergen. Multiprocessor resource allocation for hard real-time streaming with a dynamic job mix. *IEEE Proceedings in Real-Time and Embedded Technology and Application Symposium (RTAS)*, March 2005.

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

[29] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. *Proceedings of Design, Automation and Test in Europe Conference*, March 2002.

[30] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. *Proceeding 39th Design Automation Conference*, June 2002.

[31] Martijn Rutten. http://sourceforge.net/projects/timedoctor/, 2011.

[32] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2009.

[33] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. *International Symposium on Circuits and Systems*, 2000.

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

[35] Ken Tindell and Hans Hansson. *Real Time Systems by Fixed Priority Scheduling*. Department of Computer Systems, Uppsala University, 1997.