Priority budget scheduling using dataflow

Tomlow, E.A.W.

Award date:
2013

TU/e Technische Universiteit Eindhoven University of Technology

ST ERICSSON

Priority budget scheduling using dataflow

By

E.A.W. Tomlow

Master Thesis
Eindhoven University of Technology
Department of Mathematics and Computer Science
Chair of Systems Architecture and Networking

Student:
E.A.W. Tomlow (0661784)
e.a.w.tomlow@student.tue.nl

Supervisor:
Dr. ir. P.J.L. Cuijpers
Assistant Professor
p.j.l.cuijpers@tue.nl

Tutor:
Dr. ir. O. Moreira
Principal DSP Engineer
ST-Ericsson B.V.
orlando.moreira@stericsson.com

Abstract

For cost and efficiency reasons multiple applications of an embedded system must often share the same hardware resources such as processors and memory. Since resources are shared between applications, decreasing the resource usage of one application frees up resources for other applications. We therefore want to minimize the amount of resources we assign to each application.

A considerable number of embedded applications need to satisfy hard real-time requirements. In order to guarantee the timing requirements of an application, we must determine the worst-case timing effect of resource arbitration on the execution of an application's tasks. One way to do this is by arbitrating access to a shared resource using a runtime budget scheduler. Budget schedulers guarantee each application a minimum amount of processing time (called budget) in a certain time period, thereby allowing us to bound the execution behaviour of an application independent of other applications.

There are various budget scheduler variants. One of the most notable differences between these variants is whether or not a budget scheduler support application's with different priority levels. Non-priority based budget schedulers tend to have problems dealing with applications with tight timing requirements. This thesis therefore focuses on resource sharing using online priority budget schedulers (PBS), which allows one high priority application preferred access to the processor resources at the cost of increased worst-case behaviour of the other applications.

We develop an improved modelling technique that captures the worst-case temporal behaviour of application in resource-shared environment where processor usage is shared by PBS arbitration, and use this improved worst-case modeling technique together with a scheduling strategy to derive application schedules and PBS parameters that allow us to minimize the required resource reservation of applications running under PBS arbitration.

We demonstrate that the proposed modelling technique and scheduling strategy can actually improve the net reservation requirements of actual real-time application in the software-defined radio domain.

Contents

Chapter 1

Introduction

Computers are more prevalent in our daily lives than most people realise. Whereas most people might think of a computer as the traditional desktop, laptop or tablet they are familiar with, there are computers embedded in all kind of devices. Examples of such devices range from telephones and televisions to cars and washing machines. These special purpose computers are referred to as embedded systems.

An embedded system is the combination of the hardware of the device and the embedded applications that run on it. For cost and efficiency reasons multiple applications must often share the same hardware resources such as processors and memory. Since a processor cannot be used concurrently by multiple applications, access to the processor is arbitrated by a runtime scheduler. This scheduler controls which application is allowed to run on the processor at any given time, and therefore has influence on the timing behaviour of an application. A considerable number of embedded applications are time critical, meaning the correctness and relevance of the results of an embedded application not only depend on the value of the result returned by the application, but also on when the result is returned. When designing an embedded system it is therefore imperative that one can guarantee that for a certain combination of running applications and shared resources, all application are always able to meet their timing requirements.

There are various techniques that can be used to verify wether an application can always satisfy its timing requirements in a resource shared environment. Based on these techniques we can determine the minimum amount of resources an application needs such that its timing requirements can be guaranteed. Due to problems of non-priority based runtime schedulers to deal with applications with tight timing requirements, this thesis will focus on resource sharing using online priority budget schedulers (PBS). We will develop an improved modelling technique that captures the worst-case temporal behaviour of application in resource-shared environment where processor usage is shared by PBS arbitration. We will use this improved worst-case modeling technique together with a scheduling strategy to derive application schedules and PBS parameters that allow us to minimize the required resource reservation of applications running under PBS

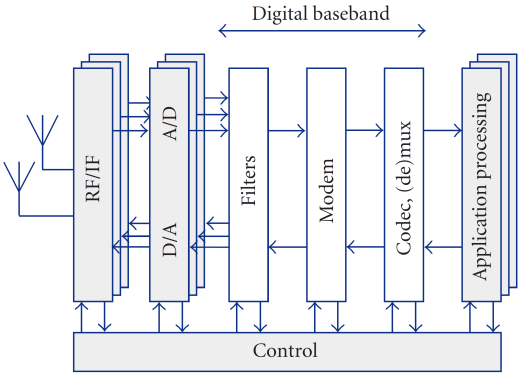


Figure 1.1: The stages of a radio transceiver (taken from [1]).

arbitration.

1.1 Context

The motivation behind the work discussed in this thesis is to decrease the required processor usage needed for baseband processing of a multi-radio modem. Baseband processing is performed when sending and receiving data over an analog channel. When sending data the bytes that have to be sent over the analog channel are encoded and modulated such that they are ready to be converted to an analog signal. When receiving data baseband processing filters, decodes and demodulates the data received from an analog-to-digital converter into meaningful data for higher level applications. Figure 1.1 shows the simplified architecture of a multi-radio modem, where the parts in white are part of baseband processing.

When communicating between two parties using communication standards such as Wi-Fi or TDS-CDMA, both standards require that at certain times transmitted data is acknowledged by the receiving party. In case the sender does not receive an acknowledgement within a certain time frame, it assumes the sent data has been lost due to signal interference or other reasons and retransmits the data. Regulations concerning efficient bandwidth usage of the radio spectrum mandate that once a radio correctly receives data, the receival of the data must be acknowledge in time, thereby avoiding unnecessary retransmissions. This mandate imposes timing requirements on the application implementing the baseband transceiver, since both the incoming data and outgoing data acknowledgement must be processed fast enough such that no unnecessary retransmissions occur. The time in which such an acknowledgement must be send can differ between radio standards, meaning each transceiver application can have different timing requirements.

Real-time applications are applications in which the correctness and relevance of the

results of an embedded application not only depend on the value of the result returned by the application, but also on the moment in time when the result is returned. Real-time applications are often divided into three categories: soft real-time, firm real-time and hard real-time, in which the occurrence of an untimely result is respectively undesirable, unacceptable or catastrophic. An example of a soft-real time application is a video player, in which dropping a few frames now and then is not very noticable and can therefore be tolerated. Regulation of the radio spectrum makes a radio transceivers a firm real-time application, meaning we do not accept a multi-radio modem that cannot guarantee that all its transceiver applications meet their timing requirements. Hard real-time applications are at times defined as application in which failure to meet timing requirements would endanger human lives. However, since from a system viewpoint the difference between an unacceptable and catastrophic event is not clearly defined, some prefer to divide real-time application in either soft real-time or hard real-time applications. We will discern two types of timing requirements a real-time application can have; a minimum throughput requirement and a maximum latency requirement. Throughput is a measure of the number of events per time unit, for instance the number of displayed frames per second of a video. Latency describes a distance between two events, for example the time between detection from a car crash and the firing of a car's airbag.

A radio transceiver is a steaming application. We define a streaming application as an application that operates over a long - possible infinite - sequence of input data. Data arrives from an external source (for instance another application) and must be processed before the resulting data is output. A streaming application on itself does not need to have timing requirements. However, in practice the memory available to the streaming application is limited, and thus the application must be able to 'keep up' with the data that arrives for processing. The fact that a steaming application must processes its data fast enough to keep up with the arrival of new data can be expressed as a minimum throughput requirement. Examples of other streaming applications include audio and video processing.

A steaming application such as a radio transceiver can be thought of to consist out of one or more tasks. Each task represents a number of single-threaded operations that must be performed on a (type of) processor in the embedded system. A steaming application is data centric, meaning tasks can receive data from other tasks or an external source, processes the data, and output the result to another task. The relation between the input and output of data between the tasks of an application can be though of to define a directed graph, in which the nodes represents tasks and edges represent tasks passing data between each other. This graph is called the *task graph* of an application.

Since the execution of an application consists of a number of task executions, we can derive the worst-case timing behaviour of the whole application by examining the worst-case behaviour of all the application's tasks. However, before we can determine the worst-case task behaviour we must first bound the effects of resource sharing on a task execution. In this thesis we assume that the only shared resources in a system are pro-

cessors, and that each processor has a local runtime scheduler that determines which application is allowed to execute on the processor at any given time. Whenever an application is given access to a processor by the processor's scheduler, the application can execute its tasks on that processor until the scheduler revokes its access. A *budget scheduler* guarantees that each application gets a minimum amount of processing time (called a budget) in a periodically repeating time interval (called a replenishment period). If we know an application's task graph and each task's worst-case computation time, then we can transform a task graph into a *temporal analysis graph*, in which each task is replaced by a task's *response model*. A response model of a task is a graph component that represents the worst-case temporal behaviour of the task under the effects of resource arbitration by a processor's runtime scheduler. Since the temporal analysis graph contains a response model for each task, it compositionally represents the worst-case behaviour of all the application's tasks, and can thus be used to derive the worst-case temporal behaviour of the entire application.

Dataflow modelling is extensively used for modelling and analysis of embedded applications and their timing properties. The task graph of a data centric application such as a steaming application can be naturally expressed as a dataflow model. When transforming the dataflow task graph to a temporal analysis graph its tasks are replaced by a dataflow response model that captures a task's worst-case response time under resource arbitration. The obtained temporal analysis dataflow model can be used to verify if the applanation's latency an throughput requirements hold using techniques such as *Maximum Cycle Mean* analysis.

The creation of an accurate response model of a task is an important step in verifying an application's timing requirements. A response model should model the worst-case time between the moment a task receives data to process, and the moment the task can output the processed data. This time between the input and output of data in a task is also called the *response time* of the task. An accurate worst-case response model of a task must take into account the resource arbitration of the processor on which the task must run, and therefore the worst-case behaviour of the runtime scheduler arbitrating access to the processor.

1.2 Problem statement

For cost and efficiency reasons multiple applications of an embedded system must often share the same hardware resources such as processors and memory. Since resources are shared between applications, decreasing the resource usage of one application frees up resources for other applications. We therefore want to minimize the amount of resources we assign to each application, while respecting the application's timing requirements. To determine the minimum resource requirements of an application we need a technique that can accurately capture the worst-case behaviour of an application's tasks as a dataflow response model, such that we can ensure that all applications are guaranteed to meet

their timing requirements.

The most prevalent response models that capture the effect of resource arbitration by budget schedulers are those that capture the effects of TDM arbitration [2, 3, 4, 5]. A TDM scheduler is a budget scheduler that divides a fixed time interval (replenishment period) into multiple - possibly differently sized - slices. Each application is assigned a single slice during which it can execute its tasks on the processor. Slices are served in a fixed order and a slice is consumed even if the application does not have a task ready to execute. Advantages of a TDM scheduler are that they are easy to implement and have negligible runtime overhead and predictable number of context switches. There have recently been considerable advances in the accuracy of worst-case task response time modelling under TDM arbitration using dataflow constructs [4, 5]. However, an inherent problem of TDM schedulers is that when we have applications with tight timing requirements then either we must accept that the application gets a considerably larger slice in which it will only spend a fraction of the time executing its tasks, or we must decrease the replenishment period of the TDM scheduler such that the time spend on context switching becomes considerably large.

A WLAN transceiver is an example of an application that has tight timing requirements. Since we want to be able to run a WLAN transceiver but do not want to suffer from the resource over-reservation or context switching overhead associated with using a TDM scheduler, we opt to choose another type of budget scheduler for resource arbitration, namely Priority Budget Scheduling (PBS). There exist some literature on dataflow response models for PBS scheduling [6, 7, 8], but all proposed models have limitations: Staschulat's model [7] requires that every task assigned to the same processor has the same worst-case computation time, and does not allow this worst-case computation time to be smaller than the application's budget. Cai [8] proposes two different response models based on wether the application's budget is either smaller, or larger or equal to a task's response time, which means we cannot use static ordering of tasks that execute on the same processor, and therefore requires increased resource reservation. Steine's model [6] is based on the the latency-rate model for TDM arbitration as proposed by Wiggers [3] and can therefore be excessively pessimistic with respect to the capturing the actual task's response, such as which was shown to be the case for TDM arbitration by Butala in [5]. Finally, all of these PBS arbitration models assume there is always a minimum wait time for the high priority application before it can execute on a processor, even if it has sufficient remaining budget. This wait time can be as large as a the largest low priority task budget in Steine's model. Although this assumption is made such that one can bound the maximum number of context switching, we belief it is possible to find another method to bound these context switches such that we can eliminate this added pessimism from our proposed PBS response time model.

Once we have an appropriate response model that accurately captures the effects of PBS arbitration we can verify if an application's timing requirements can be met for a given amount of resource reservation. However, this does not tell us how to obtain sufficiently small resource reservation: There is no clearly defined methodology for choosing appli-

cation schedules and PBS scheduler parameters such as replenishment period and application budget given a set of applications that a system must run. Currently the choice of scheduler parameters is done my an educated guess of a domain expert, but we would like to propose a methodology that given a set of applications and information about their timing requirements the methodology can come up with schedulers and scheduler parameters that should yield acceptably low resource reservation requirements.

1.3 Project goals

We want to show that by using priority budget schedulers instead of non-priority based budget schedulers we can reduce the total required processor reservation needed for running certain applications, while still being able to guarantee that every application's timing requirements can be met. We discern the following goals of this project:

- Define a new dataflow PBS arbitration response time model that:
  - Provides accurate estimation of the worst-case behaviour of an application's tasks under PBS scheduling.
  - Does not make unnecessary assumptions about the characterizes of PBS arbitration, such as imposing limits on application budget of worst-case task computation times.
- Define a heuristic that given a set of applications and their timing requirements can come up with application schedules and scheduler parameters for each processor's priority budget scheduler such that the timing requirements of both high priority and low priority applications can be fulfilled using an acceptable amount of resource reservation requirements.

1.4 Approach

We can make use of the recent improvements on the state-of-the-art in TDM response models as introduced by Lele and Butala in [4, 5] to come up with an improved PBS response model. By making a small change to the the latency actor of the proposed TDM response model we can make the model accurately represent the worst-case timing behaviour of task executions under PBS arbitration.

The scheduling strategy heuristic is based on the assumption that tasks with tight timing requirements are more sensitive to the choice of scheduling parameters that other applications. The application with the tightest timing requirements is given high priority, and the heuristic will initially set the processor replenishment periods of the runtime priority budget schedulers to be equal to the required production period of the high

priority application. This allows the high priority application to have the lowest reservation theoretical possible, significantly reducing the minimum required reservation of the application when compared to a budget scheduler without different priority levels. The heuristic will then try to improve the required reservation of the low priority applications by reducing the replenishment period of the PBS arbitrated processors by certain integer factors. Reducing the replenishment period by an integer factor allows the budget of the high priority application to be reduced by this same factor, thereby keeping the reservation for the high priority application to the lowest value possible, while potentially decreasing the minimum required reservation of the low priority applications. The combination of replenishment periods that allows the low priority application to have the minimum amount of processor reservation is found using a binary search.

1.5 Contributions

This thesis makes the following contributions:

- We define a generally applicable dataflow response model for capturing the worst-case temporal effects of PBS arbitration. The introduced response model does not make any restrictive assumptions and is most likely less conservative than the current state-of-the-art PBS response models.
- We formulate equations that can give the exact finish times of task executions under PBS arbitration.
- We define an approach for bounding the worst-case number of context-switches under PBS arbitration. This bound is derived based on analysis of the task graph and schedule of the high priority application.
- We propose a PBS scheduling strategy that given a set of applications and their timing requirements can find application schedules, scheduler parameters and the minimum required reservation for all applications.
- We conduct experiments with the proposed PBS response model and scheduling strategy, verifying that using PBS arbitration can indeed improve the minimum required resource reservation for actual applications in the software-defined-radio domain.

1.6 Thesis organisation

Chapter 2 and chapter 3 will provide a basic introduction to dataflow and how dataflow can be used to verify timing requirements of jobs. Chapter 4 will more elaborately define the problems with the current dataflow response models for TDM and PBS. Chapter 5 will introduce the proposed new response model that accurately captures the resource

arbitration effects of a PBS scheduler. Chapter 6 will introduce the scheduling heuristic that can be used to obtain scheduler parameters and application schedules given a set applications and their timing requirements. Chapter 7 will discuss the various experiments performed to come up with a scheduling strategy and to show the potential improvement of our proposed PBS response model over the state-of-the-art TDM response model. Chapter 8 will give a summary of the highlight of this thesis, and also discus its limitations and potential further work.

Chapter 2

Dataflow

A dataflow model expresses the flow of information through some modeled construct. In a dataflow model information can travel between pairs of entities, where each of these entities can consume and/or produce information when certain conditions hold. Due to the fact that information always flows between a pair of entities, a natural interpretation of a dataflow model is as a directed multigraph (V, E, src, snk), in which the set of vertices V of the graph represent data consuming and/or producing entities, and the set of edges E define how data can flow between these entities. The functions $src : E \to V$ and $snk : E \to V$ define respectively the start and end vertex of an edge. The flavour of dataflow under consideration defines with which additional information the vertices and edges are annotated, and how this information should be interpreted. A dataflow model annotated with sufficient information defines an execution model.

This chapter will introduce the flavours of dataflow that are relevant for this thesis. We will introduce some basic properties of the dataflow variants under consideration, and see how we can convert these dataflow variants to the single-rate variant we will be using in most of this thesis.

2.1 Synchronous dataflow

A synchronous dataflow model [9] is a tuple $(V, E, src, snk, \tau, d, prod, cons)$, which is essentially a multigraph annotated with additional information. Each vertex in V is called an actor and represents a time-consuming entity, and the edges in E are called arcs and represent First-In-First-Out (FIFO) queues for transferring data between actors. The functions $src : E \to V$ and $snk : E \to V$ define which actors are connected by an arc, where for arc $e \in E$ we call actor $src(e)$ the source of the arc and actor $snk(e)$ the sink. Data is abstracted as chunks, where each chunk of data is called a token. If there are n data chunks in the queue of an arc we say that there are n tokens on that arc. Each actor in a dataflow model can fire, during which it transfers data through the

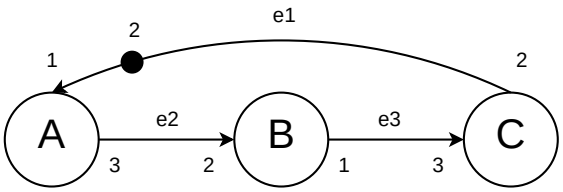


Figure 2.1: A synchronous dataflow graph.

model by removing data from the queues of those arcs in which the actor is a sink and appending data to the queues of those arcs for which the actor is a source. We call the arcs for which an actor v is the source the output arcs of v and the arcs in which v is a sink the input arcs of v. Each time an actor v fires it will start by consuming tokens from all its input arcs. The number of tokens that will be consumed from each input arc is defined by the function $cons : E \to \mathbb{N}_{\geq 1}$. After consuming tokens from its input arcs, actor v will wait a fixed time until it will produce a number of tokens on each of its output arcs as defined by $prod : E \to \mathbb{N}_{\geq 1}$. When an actor fires we consider the actor to be executing. The time between token consumption and production of a firing of actor $v \in V$ is therefore called the execution time of v, and the execution time is a constant defined by the function $\tau : V \to \mathbb{R}_{\geq 0}$. Each actor can only fire if its firing rule is satisfied. The firing rule for a multi-rate dataflow actor is that all input arcs of the actor should contain at least the number of tokens that the firing of the actor will consume from that arc. Since actors require tokens to fire, a dataflow model can contain an initial placement of tokens on one or more arcs. By convention the number of initial tokens on an arc e is called the delay of e, which is given by the function $d : E \to \mathbb{N}_{\geq 0}$.

Figure 2.1 depicts a synchronous dataflow model. It has three actors named A, B and C which are represented by circles that contain their name. An arc is represented by an arrow that points from the source actor to the sink actor. Each arrow has a number attached to both its ends indicating the number of tokens produced on and consumed from this arc by respectively the arc's source and sink. If an arc has a non-zero delay it contains a small black dot with next to it the delay of the arc. When relevant for discussion arcs will be labeled with a name, where the arc's name is placed near the center of the arrow representing the arc. Actor execution times are usually not depicted, but when they are an actor's execution time is placed under its corresponding actor. Note that in figure 2.1 arc e_1 is the only arc with a non-zero delays in the depicted model, and its delay is two. This implies that only the firing rule of actor A is satisfied, and that a firing of A would consume one of the initial tokens from arc e_1 and produce two tokens on arc e_2.

The production and consumption rates of arcs in a synchronous dataflow model define how the firings of an arc's source and sink actor relate to each other. However, allowing a synchronous model to have arbitrary rates complicates the expression of certain desirable properties of a dataflow model. We therefore discern a subset of synchronous dataflow models in which every model has restricted production and consumption rates. We call

this subset homogenous synchronous dataflow.

2.1.1 Homogenous synchronous dataflow

A homogenous synchronous dataflow model (from here on out just referred to as homogenous dataflow model) is a synchronous dataflow model $(V, E, src, snk, \tau, d, prod, cons)$ for which for every arc $e \in E$ it holds that $prod(e) = cons(e) = 1$. The production and consumption rates of a homogenous dataflow model ensure that each actor firing will consume exactly one token from all its input arcs and produce exactly one token on all its output arcs. Expressing certain properties of a dataflow model is often less complex when the model is homogenous. Examples of such properties include the relation between *start times* of actor firings and checking whether or not a dataflow model is *deadlock-free*, which are both defined in section 2.3.

What a homogenous dataflow model gains with respect to ease of analysis when compared with an arbitrary synchronous dataflow model, it tends to lack in the ease in which certain concepts can be expressed. For instance, a synchronous dataflow model can trivially express that some actor produces more data than one firing of another actor can consume, just by setting the rates of the relevant arcs connecting these actors such that they have an appropriate ratio. Expressing the same property in a homogenous model is more complicated, since the rates are fixed and we are therefore forced to use more complex modelling tricks to express the desired concept. Conveniently enough, section 2.5 shows that we can convert any dataflow model of the in this thesis relevant dataflow flavours to a model in which all arcs have a production and consumption rate of one. We can therefore first construct a dataflow model that expresses the desired relation between actor firings without worrying about the rates of the whether the model is homogenous or not, and can later derive from this model a homogenous model when needed.

2.2 Cyclo-static dataflow

The production and consumption rates and execution time of any actor of a synchronous dataflow model is always constant across any of its firings. Cyclo-static dataflow [10] relaxes the requirement for constant rates and execution times by allowing them to be periodically recurring: Each arc has two sequences of respectively the consumption rates of the arc's sink and the production rates of the arc's source. Furthermore, each actor has a sequence of execution times. Now for any such sequence of arbitrary length p that belonging to actor v, the k-th firing of v will use the $(k \bmod p)$-th entry of the sequence as the actor's relevant production/consumption rate or execution time.

More formally; A cyclo-static dataflow model is a tuple $(V, E, src, snk, \tau, d, prod, cons, p_\tau, p_{prod}, p_{cons})$ where V, E, src, snk and d are defined as in a synchronous dataflow model. The functions $p_\tau : V \to \mathbb{N}_{\geq 0}$ and $p_{prod}, p_{cons} : E \to \mathbb{N}_{\geq 0}$ denote the length

of respectively the execution time and production and consumption sequences of an actor or arc. We define $\tau : (V \times \mathbb{N}_{\geq 0}) \rightarrow \mathbb{R}_{\geq 0}$ such that $\tau(v, k)$ gives the execution time of actor $v \in V$ during its k-th firing, and for any firing $k \in \mathbb{N}_{\geq 0}$ it holds that $\tau(v, k) = \tau(v, k \bmod p_\tau(v))$. Similarly we define $cons, prod : (E \times \mathbb{N}_{\geq 0}) \rightarrow \mathbb{N}_{\geq 0}$ such that $cons(e, k)$ and $prod(e, k)$ give respectively the consumption and production rates of an edge $e \in E$ during firing $k \in \mathbb{N}_{\geq 0}$ of its source and sink actor, such that $cons(e, k) = cons(e, k \bmod p_{cons}(e))$ and $prod(e, k) = prod(e, k \bmod p_{cons}(e))$ hold. The firing rule of a cyclo-static dataflow actor is the same as for a synchronous dataflow actor; an actor's firing rule is satisfied if all its input arcs contain at least the amount of tokens the firing of the actor would consume.

Definition 2.1. *The* firing behaviour $b_f : (V \times \mathbb{N}_{\geq 0}) \rightarrow (\mathcal{P}(E \times \mathbb{N}_{\geq 0}) \times \mathcal{P}(E \times \mathbb{N}_{\geq 0}) \times \mathbb{R}_{\geq 0})$ *of a firing $k \in \mathbb{N}_{\geq 0}$ of actor $i \in V$ is a tuple (cn, pr, tm) where $cn = \{(e, cons(e, k)) \mid e \in E \wedge cons(e) = i\}$, $pr = \{(e, prod(e, k)) \mid e \in E \wedge prod(e) = i\}$ and $tm = \tau(i, k)$.*

If two firings of the same actor consume/produce the same amount of tokens from/to each arc and both firings have the same execution time, then the two firings have the same firing behaviour. The firing behaviour of an actor in a cyclo-static dataflow graph will start to repeat after a finite number of firings.

Definition 2.2. *The* firing behaviour period $p_f : V \rightarrow \mathbb{N}_{\geq 1}$ *of an actor is the minimum number of actor firings necessary such that for any firing $k \in \mathbb{N}_{\geq 0}$ and actor $i \in V$ it holds that $b_f(i, k) = b_f(i, k + p_f(i))$. The firing behaviour period of actor $i \in V$ is given by*

$$p_f(i) = lcm(\{p_\tau(v)\} \cup \{p_{prod}(e) | e \in E \wedge src(e) = i\} \cup \{p_{cons}(e) | e \in E \wedge snk(e) = i\})$$

where the function lcm calculates the least common multiplier of the numbers in the given set.

Now that we have defined both synchronous and cyclo-static dataflow we can make the following observation.

Lemma 2.1. *Every synchronous dataflow model $(V, E, src, snk, \tau^m, d, prod^m, cons^m)$ can be converted into a cyclo-static dataflow model $(V, E, src, snk, \tau^c, d, prod^c, cons^c, p_\tau, p_{prod}, p_{cons})$ by defining the fucntion $\tau^c, prod^c, cons^c$ such that for any actor $v \in V$, arc $e \in E$ and actor firing $k \in \mathbb{N}_{\geq 0}$ it holds that $\tau^c(v, k) = \tau^m(v)$, $cons^c(e, k) = cons^m(e)$, $prod^c(e, k) = prod^m(e)$ and $p_\tau(v) = p_{prod}(e) = p_{cons}(e) = 1$.*

Since every homogenous single-rate dataflow model is a synchronous dataflow model, and every synchronous dataflow model can be converted to a cyclo-static dataflow model using lemma 2.1, cyclo-static dataflow is a generalization of both synchronous and homogenous dataflow. Since we want to avoid mixing the different notations of synchronous dataflow and cyclo-static dataflow when specifying properties or algorithms, we will exploit the fact that cyclo-static dataflow is a generalization of the for this thesis relevant dataflow flavours by specifying from this point onwards any dataflow model only as a cyclo-static dataflow model. However, it is still convenient to be able to express that a

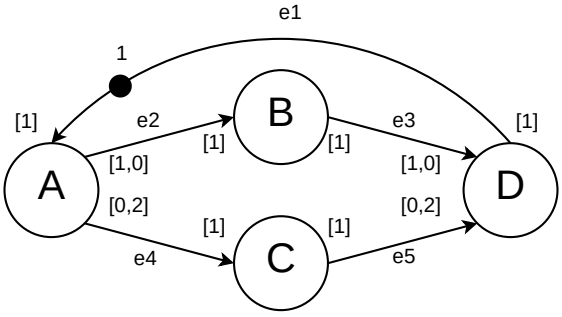


Figure 2.2: A cyclo-static dataflow graph.

certain cyclo-static dataflow has the same properties as a homogenous or synchronous dataflow model, especially since we've seen that a homogenous dataflow model allows for easier expression of certain desirable properties. We therefore introduce the notion of a cyclo-static dataflow graph being *single-rate* or *multi-rate*, which expresses that a cyclo-static model has the same properties as respectively a homogenous or synchronous dataflow model.

Definition 2.3. *A cyclo-static dataflow model is* multi-rate *if* $p_\tau(v) = p_{prod}(e) = p_{cons}(e) = 1$ *for any actor* $v \in V$ *and arc* $e \in E$.

Definition 2.4. *A cyclo-static dataflow model is* single-rate *if it is multi-rate and* $prod(e,k) = cons(e,m) = 1$ *for any arc* $e \in E$ *and firing* $k, m \in \mathbb{N}_{\geq 0}$.

If a cyclo-static dataflow model is single-rate or multi-rate then there exists respectively a homogenous or synchronous dataflow model that can be converted into this cyclo-static dataflow model using lemma 2.1. Such a cyclo-static dataflow model will have the same (for this thesis relevant) properties as the original homogenous or synchronous dataflow model. The fact that we call a cyclo-static dataflow model single-rate if it has the same properties as a homogenous dataflow model, or multi-rate if it has the same properties of a synchronous dataflow model is no coincidence; single-rate and multi-rate are in some dataflow literature used as synonyms of respectively homogenous and synchronous dataflow. However, in this thesis single-rate and multi-rate refer only to a property of a cyclo-static dataflow model.

Figure 2.2 depicts a cyclo-static dataflow model. Note that each arc has a sequence of production and consumption rates, in which the rate of the first actor firing is the left-most number in the sequence. Observe that actor A will produce one token on arc e_2 during its first and third firing, while it will produce two tokens on arc e_4 during its second and fourth firing.

2.3 Firings and behaviour of dataflow models

Once we have a dataflow model we are interested in knowing when the actors of that dataflow model will fire. We can talk about an actor firing in terms of its start, execution and finish time. We denote the start time of firing k of actor i by $s(i,k)$ and its execution time by $\tau(i,k)$, where by convention $k = 0$ denotes the first actor firing. We can then define a firing's finish time as $f(i,k) = s(i,k) + \tau(i,k)$.

An actor's firing rule enforces that an actor can only fire if there are enough tokens on each of its input arcs. Every token in a dataflow model is either produced by an earlier actor firing or is an initial token. This means that after a large enough number of firings of actor j, an arc e for which $src(e) = i$ and $snk(e) = j$ will constrain the earliest start time of the next firing of j such that it cannot start before a certain firing of i has finished. This is due to the fact that j cannot fire if arc e does not contain enough tokens, and since actor i is the only actor that produces tokens on the arc, a firing of j will at some point need one or more tokens produced by a firing of i. If our dataflow model is single-rate, we can express this relation between an arc's source and sink actor as follows.

Lemma 2.2. *In a single-rate dataflow model an arc e with $src(e) = i$ and $snk(e) = j$ constrains the earliest start time of a firing k of actor j in terms of the finish time of a firing of i such that*

$$s(j,k) \geq f(i, k - d(e)) \tag{2.1}$$

Although arcs constrain the earliest time an actor firing can occur, they do not define when the actor should fire, nor enforce that an actor fires at all. When the start times of all firings of the actors in a dataflow model are fixed to occur at a precise moment in time, we say that the start times of the actor firings define a schedule.

Definition 2.5. *A* schedule *of a dataflow model is a function $sched : (V \times \mathbb{N}_{\geq 0}) \rightarrow \mathbb{R}_{\geq 0}$ such that $sched(i,k)$ defines the start time of firing $k \in \mathbb{N}_{\geq 0}$ of actor $i \in V$.*

Definition 2.6. *A schedule is* admissible *for a certain dataflow model if for every firing of every actor in the dataflow model the by the schedule defined start time does not violate an actor's firing rule.*

When we fire the actors of a dataflow model in accordance with a schedule, we say we are executing the model under that schedule. There are a potentially infinite number of admissible schedules for a given dataflow model, but in this thesis we will focus on one important type of schedule.

Definition 2.7. *A* self-timed schedule *is a schedule in which every actor fires as soon as its firing rule is satisfied.*

Note that when executing a dataflow model under a self-timed schedule, any actor firing will occur at the earliest time possible. We will see in chapter 3 that when we execute

a dataflow model under a self-timed schedule, the firings of the dataflow model have certain interesting properties.

Lemma 2.3. *When executing a single-rate dataflow model under its self-timed schedule the start time of a firing $k \in \mathbb{N}_{\geq 0}$ of actor $i \in V$ is defined by*

$$s(i,k) = \max_{e \in E \wedge snk(e)=i} \begin{cases} s(src(e), k - d(e)) + \tau(src(e), k - d(e)) & k \geq d(e) \\ 0 & k < d(e) \end{cases} \tag{2.2}$$

When we execute a dataflow model - regardless of the schedule - we can either fire all actors an infinite number of times, or there exists at least one actor for which after a finite number of firings its firing rule will never be satisfied again during the model's execution.

Definition 2.8. *A dataflow model is* deadlocked *if during execution of the model under an admissible schedule at least one of its actors cannot fire an infinite number of times.*

Since we want to be able to execute a dataflow model for an infinite amount of time, we are only interested in models that will not deadlock. Whether or not an arbitrary dataflow model can deadlock depends on which actors are connected by arcs and the number of delays and rates of these arcs. A deadlock occurs when there is a cyclic dependency between the firings of two or more actors such that each firing requires tokens produced during the firing of the other to fire itself. Determining if a single-rate model can deadlock is fairly easy.

Lemma 2.4. *A single-rate dataflow model is deadlock-free if and only if all cycles in the dataflow graph contain an arc with one or more delays. [11].*

Checking if an arbitrary dataflow graph can deadlock is less straightforward, but as we will see in section 2.5 we can always convert such a model such that it becomes single-rate, and then check if the obtained single-rate model has at least one delay in each cycle.

Even if a graph is deadlock-free, and each actor can thus theoretically fire an infinite amount of times, it may not be feasible to execute a given dataflow model indefinitely. When executing a deadlock-free dataflow model under some schedule for an infinite amount of time, we will fire each actor infinite may times. However, there exist cyclo-static dataflow models that can accumulate an unbounded number of tokens on some of their arcs. Since an arc containing an unbounded number of tokens will need an unbounded large FIFO queue to store the data represented by these tokens, we would need infinite memory if we wanted to fire the model's actors indefinitely. The assumption that we have infinite memory is unreasonable in almost all cases, hence we are only interested in dataflow models which we can execute indefinitely in finite memory. How we can check if a dataflow model fulfills this requirement will be explained in section 2.4.

2.4 Repetition vectors and iterations

By observing that the firing rule of an actor is solely dependant upon the number of tokens on its input arcs - and thus distinguishing between different tokens on the same arc is not meaningful - we can define the state of a cyclo-static dataflow model in the following manner.

Definition 2.9. *The state of cyclo-static dataflow model is defined by:*

- *The number of tokens on each arc.*
- *The number of performed actor firings of each actor modulo the actor's firing behaviour period.*
- *A sequence containing the remaining firing time for each active actor firing.*
- *A sequence containing per arc the number of tokens that will be produced on that arc when one of the active actor firings finishes.*

When we fire each actor in a dataflow model a certain number of times we may observe that the model has the same state before and after these firings.

Definition 2.10. *A* repetition vector *of a dataflow model is a vector $\vec{r}$ such that if every actor i is fired a number of times equal to the i-th entry of $\vec{r}$ (denoted by $\vec{r}(i)$), the state of the dataflow model before and after performing these firings is the same.*

Definition 2.11. *An* iteration *of a dataflow model is a set of actor firings such that each actor fires a number of times equal to its repetition vector entry.*

We call the smallest non-trivial repetition vector *the* repetition vector, since if it exists it is unique. Whether or not the repetition vector exists is an important property of a dataflow model; if it exists, we know there is a schedule such that we can perform an infinite number of actor firings using only finite memory, since after firing each actor the finite number of times specified in the repetition vector, the number of tokens on each arc is back to its initial value.

Definition 2.12. *A dataflow model is* consistent *if and only if it has a non-trivial repetition vector.*

Since we want to the able to fire the actors of a dataflow model indefinitely using finite memory, we are only interested in dataflow models that are consistent and thus have a non-trivial repetition vector. From our definition of state we know we have a repetition vector if we know how many times we need to fire all actors such that the number of tokens on each arc after these firing remains unchanged, while ensuring we fire each actor a number of times equal to an integer multiple of its firing behaviour period. We can express this requirement as a linear constraint.

Lemma 2.5. *To find a repetition vector of a cyclo-static dataflow model we must find*

a vector $\vec{q}$ *such that for each arc* $e \in E$ *with* $src(e) = i$ *and* $snk(e) = j$ *it holds that*

$$\vec{q}(i) \cdot \sum_{k=0}^{p_f(i)-1} prod(e,k) - \vec{q}(j) \cdot \sum_{k=0}^{p_f(j)-1} cons(e,k) = 0 \tag{2.3}$$

If such a vector exists, a repetition vector of the dataflow model is the vector $\vec{r}$ *such that* $\vec{r}(i) = \vec{q}(i) \cdot p_f(i)$.

The equations obtained in this fashion are called the balance equations of the dataflow model. Another way to express these balance equations and calculate a model's repetition vector is by using a topology matrix. The topology matrix of a dataflow model is a $|E| \times |V|$ matrix that describes the relation between token production and consumption along the arcs of the model. Each row of a topology matrix represents an arc of the dataflow model and each column an actor. The entry for each arc and actor combination in a topology matrix is equal to the token production minus the token consumption of the actor on that arc across a certain number of its firings.

A problem that we face when we want to create a topology matrix for a cyclo-static dataflow graph is that a topology matrix cannot express that the production and consumption rates of an actor can vary between firings. We therefore create a *lumped topology matrix*, in which each matrix entry for an actor $i \in V$ and arc $e \in E$ corresponds to the total number of tokens actor i consumes and produces on arc e during $p_f(i)$ consecutive firings.

Definition 2.13. *The* lumped topology matrix *of a cyclo-static dataflow model is a* $|E| \times |V|$ *matrix* M *such that:*

$$M_{e,i} = \begin{cases} \sum_{k=0}^{p_f(src(e))-1} prod(e,k) - \sum_{k=0}^{p_f(snk(e))-1} cons(e,k) & \text{if } src(e) = snk(e) = i \\ \sum_{k=0}^{p_f(src(e))-1} prod(e,k) & \text{if } src(e) = i \\ -\sum_{k=0}^{p_f(snk(e))-1} cons(e,k) & \text{if } snk(e) = i \\ 0 & \text{otherwise} \end{cases}$$

Lemma 2.6. *If there exists a non-trivial solution such that* $M \cdot \vec{q} = \vec{0}$ *for a lumped topology matrix* M*, then the repetition vector of a cyclo-static dataflow model whose lumped topology matrix is* M *is equal to* $\vec{r}(i) = \vec{q}(i) \cdot p_f(i)$*, for each actor* $i \in V$.

As an example we will calculate the repetition vector of the cyclo-static dataflow graph displayed in figure 2.2. Note that the firing behaviour periods of the actors in the depicted model are $p_f(A) = p_f(D) = 2$ and $p_f(B) = p_f(C) = 1$. Using definition 2.13 we obtain the lumped topology matrix M.

$$M = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} -2 & 0 & 0 & 2 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 2 & 0 & -1 & 0 \\ 0 & 0 & 1 & -2 \end{pmatrix} \end{array}$$

A non-trivial solution for $\vec{q}$ such that $M \cdot \vec{q} = \vec{0}$ is the vector $\vec{q} = \begin{bmatrix}1 & 1 & 2 & 1\end{bmatrix}^T$. Multiplying each entry in $\vec{q}$ with its actor's firing behaviour period yields the vector $\vec{r} = \begin{bmatrix}2 & 1 & 2 & 2\end{bmatrix}^T$. Vector $\vec{r}$ is thus a repetition vector of the dataflow model depicted in figure 2.2, meaning the model performs an iteration each time we fire actor B once and actors A,B and C twice. Note that any scalar $n \in \mathbb{N}$ multiplied with $\vec{r}$ yields a repetition vector, but because a non-trivial repetition vector cannot fire actor B less than once, we know vector $\begin{bmatrix}2 & 1 & 2 & 2\end{bmatrix}^T$ is *the* repetition vector of the cyclo-static dataflow model shown in figure 2.2.

2.5 Conversion to single-rate

In earlier sections of this chapter we indicated that there are certain properties that single-rate models have which arbitrary cyclo-static models lack. An example of such a property is lemma 2.4, which allows us to easily verify that a single-rate dataflow model is deadlock-free by checking if every cycle has at least one token. In section 3.3 we will introduce the most significant reason why single-rate models tend to be preferable; they can be analysed using *maximum cycle mean* analysis, thereby yielding information about *throughput* and *latency* of the entities abstracted as actors. Since we will be using maximum cycle mean analysis on all of our dataflow models, a single-rate model is for our purpose the most desirable form of a dataflow model.

It is always possible to convert an arbitrary consistent cyclo-static dataflow model to a single-rate dataflow model. Before we worry about how we can perform this conversion, we first define the notion of equivalence that we want this conversion to preserve.

Definition 2.14. *A dataflow model G with actors V_G and execution time function τ_G that is executing under schedule S_G is* execution equivalent *to a dataflow model H with actors V_H and execution time function τ_H that is executing under schedule S_H, if and only if there exists a bijective function map $: (V_G \times \mathbb{N}_{\geq 0}) \to (V_H \times \mathbb{N}_{\geq 0})$ such that for all $i \in V_G$ and $k \in \mathbb{N}_{\geq 0}$ it holds that*

- *$S_G(i,k) = S_H(map(i,k))$ and $S_G(i,k) + \tau_G(i,k) = S_H(map(i,k)) + \tau_H(map(i,k))$*
- *$map(i,k) = (j,m)$ such that actor j is an abstraction of the same entity as actor i*

We thus want to convert an arbitrary consistent cyclo-static dataflow model to a single-rate model such that when executing both models under their respective self-timed schedules the models are execution equivalent. The main idea behind the conversion is that each actor in the unconverted model will be represented by one or more actors in the single-rate model, such that the k-th firing of such an actor i_m represent the $(k \cdot \vec{r}(i) + m)$-th firing of the original actor i, where $\vec{r}$ is the repetition vector.

Lemma 2.7. *We can transform an arbitrary consistent cyclo-static dataflow model G with repetition vector $\vec{r}$ into an execution equivalent single-rate dataflow model H in the following manner*

1. *For every actor i in model G we add $\vec{r}(i)$ actors to model H, where we denote these actors by i_m with $0 \leq m < \vec{r}(i)$ and actor i_m has execution time $\tau(i, m)$. All of these actors represent the same entity as the original actor i. We call each actor i_m a copy of i.*

2. *For every arc e in model G with $src(e) = i$ and $snk(e) = j$ we have to add arcs between the copies of i and j to model H. Any actor copy i_m with $0 \leq m < \vec{r}(i)$ in model H should have $prod(e, m)$ arcs connecting i_m to actor copies of j, and in reverse any actor copy j_n with $0 \leq n < \vec{r}(j)$ must have $cons(e, n)$ arcs connecting it to copies of i. We assume each actor copy i_m and j_n has a number of ports equal to respectively $prod(e, m)$ and $cons(e, n)$, where each of these ports represent a begin or end point of a single arc. We number the ports such that an actor copy i_m has port numbers in the interval $[\sum_{k=0}^{m-1} prod(e, k), \sum_{k=0}^{m} prod(e, k))$. The ports for actor copies of j are numbered similarly, but based on the consumption rate of edge e. Note that the actor copies of i and the actor copies of j have the same total number of ports, which we designate with N. We can now define a bijective function $connect : [0 \ldots N] \times [0 \ldots N]$ between the port numbers of the actor copies of i and the actor copies of j. We define this function such that*

$$connect(p) = (p + d(e)) \bmod N$$

If $connect(p) = q$ we introduce an arc e_{pq} in H such that it has as source the actor copy of i with the port numbered p and as sink the actor copy of j with the port number q. The delay of the arc e_{pq} is $\lceil \max(0, d(e) - q)/N \rceil$.

When we convert a dataflow model to single-rate, the created single-rate model tends to have more actors and arcs than the original model. An increase in the number of actors and arcs in a dataflow model is generally undesirable, since it will increase the storage space requirements and analysis time for the model. When converting a model to single-rate the increase of both actor and arcs can be exponential, which can cause problems for sufficiently large input models.

Figure 2.3 depicts a consistent cyclo-static dataflow model and its execution equivalent single-rate model. Note that we did not depict the rates of the arcs in the single-rate model to increase readability, but since the model is single-rate all of its arcs have a production and consumption rate of one.

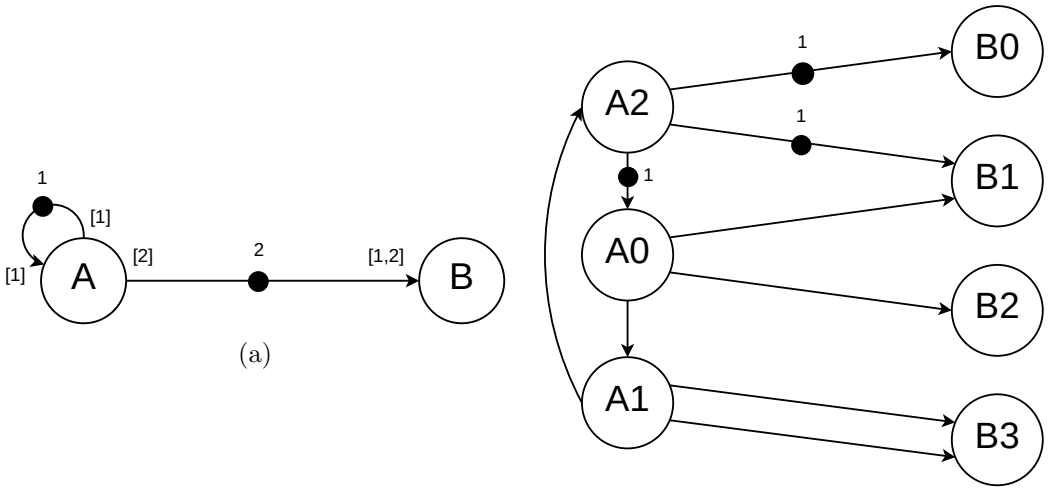


Figure 2.3: A cyclo-static dataflow graph (a) and its execution equivalent single-rate graph (b).

Chapter 3

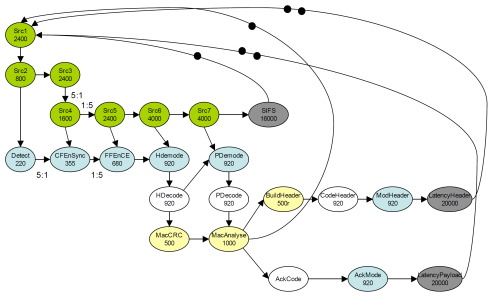
Application modelling and analysis using dataflow

A real-time application has certain timing requirements that must be satisfied for correct operation of the application. Since there may be multiple real-time applications running at the same time, and resources such as processors may have to be shared by multiple applications, meeting these timing requirement is far from guaranteed. We can abstract an application as a dataflow model which can then in turn be transformed such that it includes the worst-case timing effects of resource arbitration. By analysing the resulting dataflow model we can verify whether or not a application's timing requirements can be satisfied.

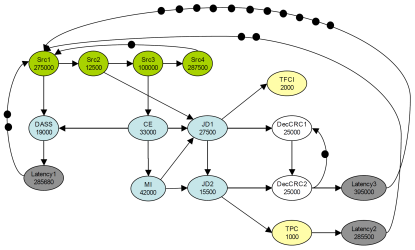
We will start this chapter by sketching how an application and its tasks can be converted to a dataflow model. Afterwards we will define how we can modify this dataflow model such that its execution includes the effects of resource arbitration, and how the modified dataflow model can be analysed such that we can verify its timing requirements. We will end this chapter by giving an overview of the relevant existing techniques for capturing the worst-case temporal effects of resource arbitration in a dataflow model.

3.1 Capturing application behaviour in dataflow

A application can be thought of to consists of one or more tasks, where each task represent a finite number of single-threaded operations that must be executed on a processor. We assume that the in this thesis relevant applications are data centric, meaning tasks receive data from other task or an external source, processes the data, and output the result to another task. Since a task cannot execute if it has no input data, an application defines a precedence relation on its tasks. This precedence relation can be though of as a directed graph, in which the nodes represents tasks and edges represent tasks passing data to each other.



(a) Dataflow model of a WLAN transceiver.



(b) Dataflow model of a TDS-CDMA transceiver.

Figure 3.1: A WLAN (figure 3.1a) and TDS-CDMA (figure 3.1b) transceiver job expressed as a dataflow model. Both dataflow models are taken from [1].

We can try to abstract an application as a dataflow model by representing each of its tasks as an actor and connecting actors by arcs as specified by the precedence relation between the application's tasks. The rates of the arcs must then be set such that they match the relative amount of data consumed and produced during one execution of each task, and actor execution times must be set to the worst-case computation time of the task. However, converting an arbitrary application to a dataflow model solely based on the precedence relation of its tasks is not trivial, since cyclo-static dataflow has limited expressivity. If the behaviour of the application is not periodically repeating, but for example dependant upon its data values, we cannot accurately express such behaviour in a cyclo-static dataflow model. Furthermore, every dataflow model should be consistent. It is therefore often necessary to create a dataflow model that approximates the application's actual behaviour. We therefore assume that a dataflow representation of an applications behaviour is created by a domain expert and provided as input to any further analysis. This dataflow model is called the task graph of the application.

Definition 3.1. *A* task graph *of an application is a consistent dataflow model conservatively approximating the execution behaviour of an application. The actors in a task graph correspond to the tasks of the application, and the execution time of each actor matches the worst-case computation time of the corresponding task.*

An example of a real-time application is a WLAN transceiver, which has a task that decodes received data and a task that calculates a CRC code of the decoded data. Figure 3.1a and figure 3.1b display the task graphs of respectively a WLAN and TDS-CDMA transceiver. The production and consumption rates of all arcs without explicit rates is 1. An arc with rate other than 1 is indicated by the notation $x : y$, which means the arc has a production rate of x and consumption rate of y.

3.2 Including effects of resource arbitration

Since the execution of an application consists out of one or more task executions, we can derive the worst-case timing behaviour of the whole application by composing the worst-case temporal behaviour of all the application's tasks. The task graph dataflow model abstracts the temporal behaviour of an application, but does not take into account timing effects due to resource arbitration. Before we can determine an application's actual worst-case task behaviour we must bound the effects of resource arbitration on a task execution.

In this thesis we assume that the only shared resources in a system are processors, and that each processor has an runtime budget scheduler that determines which application is allowed to execute on the processor at a given time. Whenever an application is given access to a processor by the processor's runtime scheduler, the application can execute its tasks on that processor until the scheduler revokes its access. The time it takes for a task execution to complete from the moment the task becomes enabled to the moment it finishes can be divided into two parts; arbitration time and processing time. The arbitration time of a task is the time it takes until the scheduler grants execution resources to the task once it becomes eligible for execution. The processing time of a task is the time between the moment a task can start executing and the moment it finishes its execution.

If we have an application's task graph dataflow model, we can transform the task graph into a *temporal analysis model*, in which each task is replaced by task's *response model*.

Definition 3.2. *A* response model *of a task is a dataflow construct that captures the worst-case timing behaviour of a task under the effects of resource arbitration by a processor's scheduler.*

Definition 3.3. *A* temporal analysis model *is a dataflow model in which each task in a task graph is replaced by its response model.*

Figure 3.2 illustrates the replacement of actors in a task graph by their response models. Each actor in the task graph is replaced by a *latency-rate* response model (which we will introduce in section 3.4) that captures the worst-case response time of the replaced actor. The obtained temporal analysis graph can can be used to verify an application's timing requirements.

Since all tasks in a temporal analysis model are replaced by their response models, it compositionally represents the worst-case behaviour of all the application's tasks, such that we can derive the worst-case temporal behaviour of the entire application. The creation of an accurate response model of a task is an important part in verifying an applications temporal requirements. A response model should model the worst-case time between the moment a task receives data to process and the task can output the processed data. This time between the input and output of data in a task is also

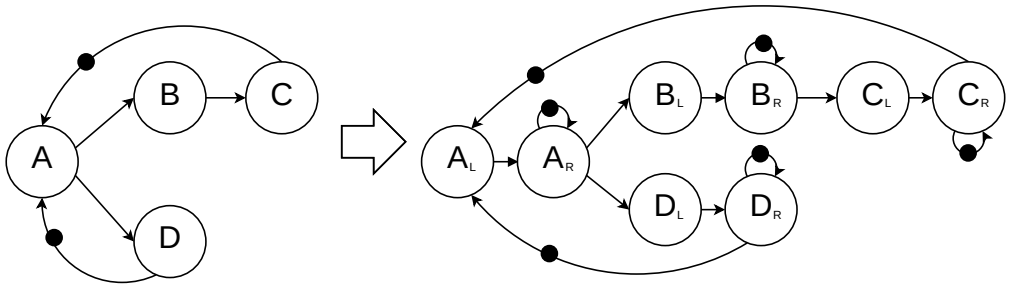


Figure 3.2: Replacing the actors in a task graph with their latency-rate response models.

called the response time of the task. An accurate worst-case response model takes into account the various factors that can affect the execution of a task, such as the resource arbitration mechanism of the processor on which the task runs and settings such as application budget, processor replenishment periods and application priorities.

Since we are interested in the worst-case temporal behaviour of the application, a task's response model is allowed to overestimate the response time of a task; if the composition of the overestimated worst-case task temporal behaviour still fulfills the application's timing requirements, then the in reality 'better' worst-case temporal behaviour will also satisfy the requirements. We call a response model *pessimistic* if its modeled worst-case behaviour is worse than the actual worst-case behaviour of a task. An overly pessimistic response model can hide the fact that the actual application's timing requirements might also be satisfied with a reduced processor budget or larger replenishment period for some of the processors, thereby leading us to overestimate the minimum amount of resources needed to run the application. Underestimation of a task's response time is not allowed, since this could lead to an underestimation of the worst-case temporal behaviour of the whole application.

3.3 Temporal analysis of a dataflow model

Once we have created a temporal analysis model we have a dataflow model that conservatively captures the worst-case temporal behaviour of an application. We discern two types of timing requirements in this thesis; maximum latency requirements and minimum throughput requirements. We can analyse a temporal analysis model using dataflow analysis techniques such that we obtain its guaranteed minimum throughput. A maximum latency requirement can be validated by first converting the maximum latency requirement into a minimum throughput constraint as described in [12]. The guaranteed throughput of a dataflow model can be computed using either static analysis or simulation based analysis. We will only describe the static analysis approach, and refer for information about the simulation approach to [13].

Static analysis of a dataflow model can be performed by calculating the Maximum Cycle

Mean (MCM) of the single-rate version of the dataflow model of interest.

Definition 3.4. *The* maximum cycle mean *of single-rate dataflow model G is defined as*

$$\mu(G) = \max_{c \in C(G)} \frac{\sum_{i \in N(c)} \tau(i)}{\sum_{e \in E(c)} d(e)}$$

where $C(G)$ is the set of simple cycles in dataflow model G and $N(c)$ and $E(c)$ are respectively the sets of actors and arcs traversed by cycle c.

The inverse of the maximum cycle mean of a dataflow model is equivalent to the model's guaranteed minimum throughput. The maximum allowed MCM of an application such that is does not violate its minimum throughput requirements is called the maximum production period of the application. The production period of the application is the time it takes for the application to complete a full iteration.

Note that both static and simulation based analysis techniques require that the dataflow model is single-rate. As we have already mentioned in chapter 2 this conversion to single-rate can lead to an exponential increase in the number of actors and edges of the model.

3.4 Existing resource arbitration models

We will give a briefly overview of the most important dataflow response models to which we will refer in this thesis. Some of the models have been superseded by less pessimistic models, but are included for completeness and the fact that they are smaller and simpler than current state-of-the-art models.

3.4.1 Response models for TDM arbitration

Although the focus of this thesis lies on PBS arbitration models, we will start with introducing some of models that capture the effects of TDM arbitration, since this is the resource arbitration method that is most extensively researched. In the formula expressing the worst-case response times of TDM arbitration we will use the following notation: P represents the replenishment period of a processor, S represents the slice size (budget) of the task's application and $\tau(i)$ represents the worst-case computation time of the i-th activation of the task.

Single-Actor model

The singe-actor response model [2] replaces each task's actor by a single actor with a self-edge (a self-edge is an edge with delay one whose source and sink is the same actor).

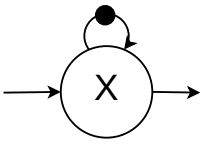


Figure 3.3: Singe actor model.

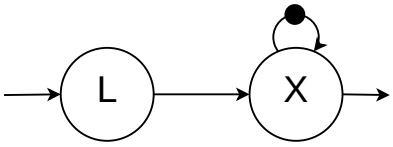


Figure 3.4: Latency-rate model.

Figure 3.3 shows the single-actor response mode. The execution time of actor X is $\lfloor \frac{\tau(i)}{S} \rfloor \cdot P + (\tau(i) \mod S)$. This model represent one of the simplest response models imaginable, but it can be excessively pessimistic.

Latency-Rate model

The latency-rate response model [3] improved on the single-actor response model by discerning a separate latency and rate phase. Figure 3.4 depicts a latency-rate response model. The execution time of actor L is $P - S$ and of actor X is $P \cdot \frac{\tau(i)}{S}$. Although this response model is less pessimistic during burst arrival of tasks, can still be exceedingly pessimistic.

Latency-Cyclic-Rate model

The latency-cyclic-rate response model [4] exploits the fact a continuous sequence of consecutive task executions can be shown to have a cyclic pattern over a fixed number of executions. Figure 3.5 depicts an example latency-cyclic-rate response model with $q = 4$ X actors, but the actual number of X actor in the model depends on the value q which is determined by the least-common-multiple between the task's computation time and its application's budget. The execution time of the actors of the latency-cyclic rate model are $P - S$ for actor L, $\lfloor \frac{k \cdot \tau}{S} \rfloor \cdot P + \lceil k \cdot \tau \mod S \rceil - \lfloor \frac{(k-1) \cdot \tau}{S} \rfloor \cdot P + \lceil (k-1) \cdot \tau \mod S \rceil$ for actors X_k with $k < q$ and actor X_q has execution time $\lfloor \frac{k \cdot \tau}{S} \rfloor \cdot P + \lceil k \cdot \tau \mod S \rceil - \lfloor \frac{(k-1) \cdot \tau}{S} \rfloor \cdot P + \lceil (k-1) \cdot \tau \mod S \rceil - (P - S)$. The latency-cyclic-rate model captures the worst-case response time of a task under TDM arbitration quite accurately, but its problem is that there is currently no known way in which this response model can be used together with the static ordering of tasks that execute on the same processor.

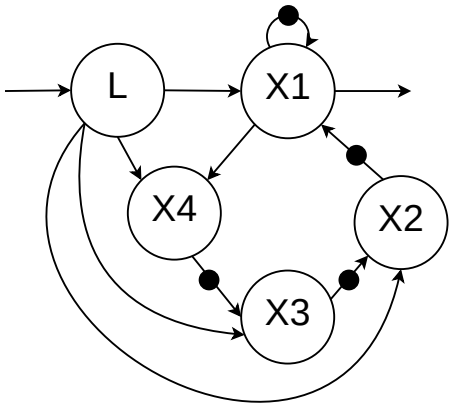


Figure 3.5: Instance of a latency-cyclic-rate model.

Multi-Rate model

The multi-rate response model (proposed in [4] and improved in [5]) is currently the most accurate TDM arbitration model together with the latency-cyclic-rate model. Similar to the latency-cyclic-rate model, the multi-rate response model takes into account previous task executions to determine the response time of the current task execution. Whereas all previously mentioned response models where single-rate, this model derives its name from the fact that it is a multi-rate model (or even a cyclo-static model if it is used in combination with static ordering). The model uses tokens to represent available budget, whereby it divides a task in smaller slices of size z, consuming for each such an slice execution one budget token. The value z is equal to the greatest-common-divisor of the task computation time(s) and the application's budget S. Figure 3.6 depicts a multi-rate response model. Actor L has execution time $P - S$, actor X has execution time z and actor W has execution time $P - z$. Actors S and C both have an execution time of zero, since they are only used to split a single incoming token into multiple parts and afterwards merge these multiple parts back into a single token. The number of initial budget tokens n is equal to S/z. A problem with the multi-rate model is that although it is both accurate and widely applicable, the fact that the response model is multi-rate can lead to an excessively large single-rate temporal analysis model, which means it can take a significant amount of time to verify an application's timing requirements.

3.4.2 Response models for PBS arbitration

This section will introduce the current state-of-the-art response models for PBS arbitration. In the formula expressing the worst-case response times of PBS arbitration we will use the following notation: P represents the replenishment period of a processor, B_H and B_L respectively represent the budget of a high priority and low priority application and $\tau(i)$ represents the worst-case computation time of the i-th activation of the task.

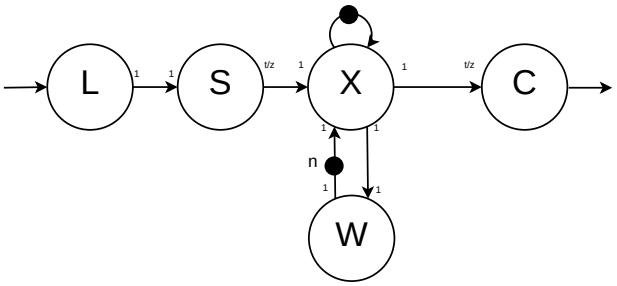


Figure 3.6: Multi-rate model.

Steine's Model

The response model as proposed by Steine [6] is an adaption of the latency-rate model for use under PBS arbitration. Steine's model assumes that a high priority task cannot pre-empt a low priority task, but has to wait until the low priority task finishes its slice before it can start to execute. The rate actor of the modified latency-rate model is left unchanged for both the high and low priority variants, but the execution time of its latency actor depends on the priority of the task's application. For a high priority application it has an execution time equal to the largest low priority task, whereas for a low priority application the rate actor's execution time is equal to $P - B_L + B_H$. A disadvantage of Steine's model is that it is uses the same rate actor execution time as the latency-rate model, and therefore suffer from the same excessive pessimism as the latency-rate response model.

Staschulat's Model

Staschulat proposes a response model for a PBS arbitrated memory arbiter in [7]. Staschulat's model defined a slot length S and required that the replenishment period is a multiple of this slot length. Furthermore, both the budget of the high and low priority task are required to be a multiple of S, and the worst-case computation time of all tasks must be equal to S. Figure 3.7 depicts Stachulat's proposed response model. The execution times of actor L is B and the execution time of actor W is $P - B$, regardless whether the model is for a task of a high of low priority application. The execution time of actor X is S for a high priority application and S plus twice the budget of all other low priority applications for a low priority application. A disadvantage os staschulat's model is that it has very restrictive assumptions about acceptable budget and worst-case task computation times.

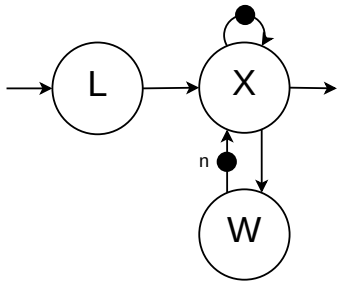


Figure 3.7: Staschulat's model.

Cai's Model

Cai proposes a PBS response model in [8], but only for the tasks of a high priority application. For a low priority application he proposes that the response model of Steine is used. Cai defines a slot size of size S, which is the smallest granularity in which a scheduling decision can be taken. The execution time of a task must be a multiple q of this slot size and the budget of the applications should be a multiple p of this slot size. Cai proposes two different response models based on wether the application's budget is either smaller or larger/equal to a task's response time. The structure of the proposed model is the same as Stachulat's response model in figure 3.7, but the execution time of its actors differ. In the response model for tasks with a execution time larger than the application's budget, the execution time for its actors are: Actor L has an execution time of S, actor X has an execution time of $p \cdot S + \lceil \frac{q \cdot S - p \cdot S}{p \cdot S} \rceil$ and actor W has an execution time of $P - p \cdot S$. The number of tokens indicated by n is one, since after one task execution we have to wait for a budget replenishment to occur before we can execute another task. For the response model suitable to tasks that have an execution time smaller than the application's budget the execution times of the model's actors are: S for actor L, $q \cdot S$ for actor X and $P - q \cdot S$ for actor W. The number of initial budget tokens is $\lfloor \frac{p}{q} \lceil$. A disadvantage of Cai's response model is that because it uses different response models based on the computation time of a task, either every task that executes on specific processor must be smaller than the budget of the application or every task must be larger/equal than the budget of the application.

Chapter 4

Problem Statement

We would like to guarantee an application's timing requirements can be met in a resource shared environment while allotting the application only a minimum amount of processor resources. Since an application cannot make progress unless it can execute its tasks on a processor, the manner in which processor sharing is arbitrated has a critical impact on the timing behaviour of an application and therefore its minimum resource requirements. We can therefore define the following steps we should take when we want to minimize an application's resource requirements:

- Determine a suitable processor arbitration method.
- Have a response model that accurately represent worst-case arbitration effect, such that the timing requirements of an application can be verified.
- Have a scheduling strategy that given a arbitration method and response model can obtain application schedules and scheduler parameters that minimize the net resource requirements.

We examine each of these points in more detail in the following sections.

4.1 Inflexibility of TDM with low latency requirements

There exists a considerable amount of research concerning the representation of the worst-case effects of TDM arbitration using dataflow response models [2, 3, 4, 5]. A TDM scheduler is a budget scheduler that guarantees each application an amount of processor time per replenishment period equal to its budget. The replenishment period of a TDM scheduler is denoted by P, and an application's budget - called its slice - is denoted by S. A TDM scheduler gives each application access to the processor at a fixed interval in each replenishment period, gradually consuming its slice regardless of whether the application executes tasks or not. Advantages of TDM schedulers are that

they are easy to implement, have negligible runtime overhead and cause a predictable number of context switches.

The worst-case response time under TDM arbitration of a task with a computation time of τ is equal to

$$(P - S) + \lceil \frac{\tau}{S} - 1 \rceil \cdot (P - S) + \tau \tag{4.1}$$

which consists out of a worst-case arbitration time of $P - S$ and a processing time of $\lceil \frac{\tau}{S} - 1 \rceil \cdot (P - S) + \tau$.

There have recently been considerable advances in the accuracy of worst-case task response time modelling of TDM arbitration [5]. However, TDM arbitration has an inherent problem with applications with tight latency requirements due to its worst-case $P - S$ arbitration time. For an application with tight latency requirements a wait time of $P - S$ can be considerable larger than its task's computation time, such that the task gets an unacceptably large task response time. In such a case we must either decrease the replenishment period P or increase the application's allotted budget S to ensure the application can meet its timing requirements.

Increasing the slice size of an application is one approach to decrease the worst-case response time of a task. However, applications with low latency requirements tend to consists out of tasks with relatively small computation times. Increasing such an application's budget just such that we can reduce its worst-case response times means that a considerable amount of its assigned budget will be spend waiting until a task becomes enabled, thereby wasting processor time that could have been used by another application.

By decreasing the replenishment period P we let the TDM scheduler switch between active applications more frequently, thereby improving the task's response time. However, each time the active application changes, the processor must make a context switch. Since the context switch time of a processor is a constant, each decrease in P means that the percentage of time the processor spends context switching increases. Because a processor cannot execute tasks during context switching, an increase in the amount of context switching wastes processor time that could have been used productively by another application. Another potential problem with decreasing the replenishment period P is that although applications may be served more often, it does not mean that over-reservation of resources does not occur: Depending on the pattern in which an application's tasks are enabled, a small replenishment period increases the likelihood that there exists a sequence of consecutive periods in which the application has no task to execute, meaning the application wastes its full allotted budget of processing time during such periods.

4.2 Problems with existing PBS response models

Since different applications can have different timing requirements, it makes sense to give applications with tighter timing requirements more flexible access to processor resources than applications with more relaxed requirements. A priority budget scheduler guarantees each application a minimum amount of processing time in a time interval, but allows the high priority application more flexibility about when it can spend its budget at the cost of an increased response time for its low priority applications. By giving the application with tight timing requirements a higher priority under PBS arbitration we are able to avoid the problems encountered by TDM arbitration.

There exist a number of response models that capture worst-case timing behaviour of PBS arbitration in literature [6, 7, 8]. However, all proposed models have a number of limitations.

4.2.1 Pessimism of Steine's model

The worst-case response time model proposed by Steine [6] uses the same rate component as the latency-rate model [3], whose response times have been shown to be overly pessimistic [4, 5]. Composing task response models that are overly pessimistic gives a high over-estimation of the temporal behavior of the entire application, which in turn would lead us to believe the application needs a higher resource reservation that its actual minimum resource requirements.

Furthermore, Steine's response model assumes the high priority job cannot actually preempt running low priority applications; if the high priority application want to execute it must wait until the current slice is completed. Its modeled worst-case arbitration time for a task of the high priority application is therefore equal to the largest budget of the low priority applications. We feel that this worst-case arbitration time is too large for a high priority application, since the whole point of using a PBS scheduler is exploiting the low arbitration time of its high priority task.

4.2.2 Limited usability of Staschulat's and Cai's Models

Staschulat's response model [7] defines a slot length and requires that all worst-case computation times are equal to this length. This means that when we use the proposed response model to capture the the worst-case timing behaviour of tasks assigned to one specific processor, these tasks must all have the same worst-case computation time.

Cai [8] proposes two different response models based on wether the application's budget is either smaller or larger/equal to a task's worst-case computation time. Using different response models based on the computation time of a task means that when we use such a response model, either every task that executes on specific processor must be smaller

than the budget of the application or every task must be larger/equal than the budget of the application.

We could theoretically satisfy the requirements posed by both Staschulat's and Cai's on the worst-case task computation times by either increasing the task's computation times such that all tasks that execute on the processor either have the same computation time (for Stachulat) or increasing all computation times such that they are larger than the application's budget (for Cai). However, rounding will almost always introduce a significant amount of pessimism in the modeled response times.

4.3 Lack of PBS scheduling strategy

We would like to assign to each application only the minimum amount of resources needed such that it can meet its timing requirements, since any time we reduce the shared resources needed by one application we free resources that can be used to run other applications. The choice of an application's schedules and scheduler parameters such as a replenishment period can have a significant impact on the minimum required resource reservation of an application. It is therefore important that a good choice is made when choosing application schedules and scheduler parameters.

There are a few basic goodliness that can help make appropriate scheduling decisions, but even these guidelines are not always correct. For instance, a smaller period allows each application to spend its budget more often, and may therefore allow us to eliminate some of an application's over-reservation it would normally need such that its tasks can have a sufficiently small worst-case response time. However, for some combinations of replenishment period, applications budget and task computation times, a smaller replenishment period may actually increase a task's response time, therefore potentially requiring more processor resources instead of less. Figure 4.1 depicts the worst-case response time of a task under TDM arbitration with computation time of 10 in two scenarios. In scenario A the task is executed on a processor with replenishment period 10 and has a slice size of 5, yielding a worst-case task response time of 20. In scenario B the task executes on a processor with replenishment period 8 and has a slice size of 4, such that it has a worst-case response time of 22. For a task with computation time 10 the larger period yields the better response time, but if the task's computation time would be 11 the scenario with the smaller period would again be better.

This example illustrates that the actual effect of a scheduling decision on an application's timing behaviour tends to depend on many factors. It is therefore in general not desirable to require optimal application schedules and processor settings, since searching for optimal solutions would not be feasible timewise. We would therefore like to define a scheduling heuristic that given a set of applications and information about their timing requirements, can come up with application schedules and replenishment periods that yield acceptably low minimum resource reservation requirements.

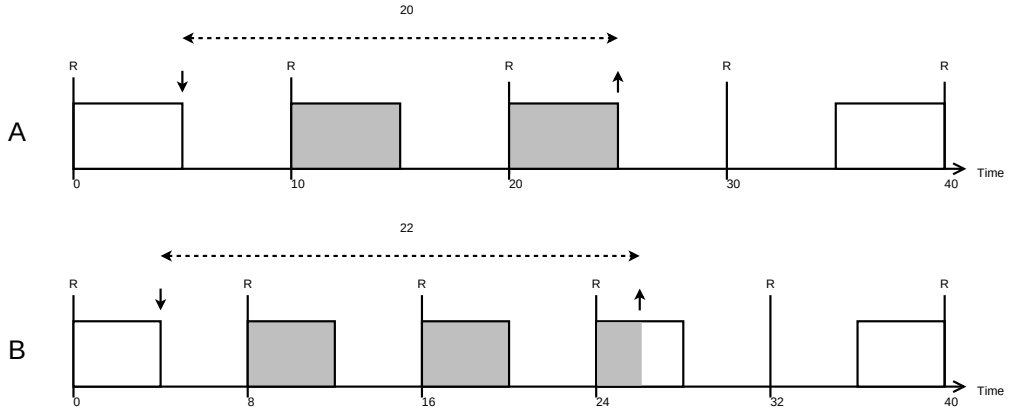


Figure 4.1: Effects of replenishment period and slice size on a task's response time.

Chapter 5

Characterizing PBS arbitration

A runtime priority budget scheduler (PBS) guarantees each application a minimum amount of processing time per time interval, but allows a high priority application more flexibility about when it can spend its budget at the cost of an increased task response time for low priority applications. Using priority based resource arbitration is attractive if the improved timing behaviour of a high priority application can offset the worsened timing behaviour of the low priority applications, allowing a net increase of free resources when compared with non-priority based budget schedulers.

This chapter will define our assumed variant of PBS arbitration, present formulas that give the exact response times of task executions of high and low priority applications, and introduce a dataflow response models that captures its worst-case temporal behaviour.

5.1 Definition of PBS arbitration

There are multiple variants of runtime PBS arbitration imaginable. Each variant can have different rules about aspects such as the supported number of priority levels or when a high priority application can pre-empt a lower priority application. We therefore start this chapter by defining the characteristics of the priority budget scheduler whose worst-case behaviour we want to represent.

We will consider a runtime priority budget scheduler with two priority levels; a high priority and a low priority. Exactly one application can have a high priority and multiple applications can have low priority. The scheduler defines a fixed size replenishment period, after which the full budget of both high and low priority applications is restored. Low priority applications consume their budget in a manner similar to TDM arbitration; the order in which low priority applications are given access to the processor is fixed, and budget is consumed regardless of whether the active low priority application

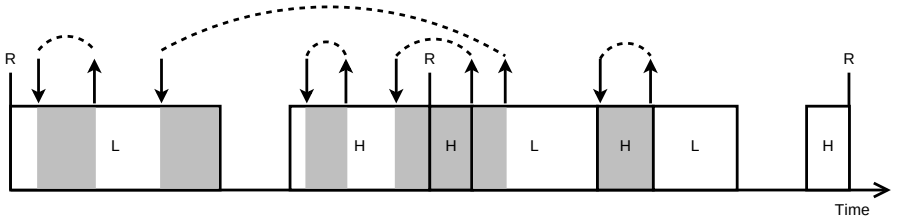


Figure 5.1: An example of task execution behaviour under PBS arbitration.

has a task to execute or not. A high priority application can preempt a running low priority application at any time as long as is has non-zero remaining budget. Once the high priority application has no more tasks to execute it can relinquishes control of the processor, allowing the previously preempted low priority application to continue. If the high priority application has more remaining budget than it can spend in the time left until the next budget replenishment occurs, then this remaining budget is gradually idled away over time, such that it has zero remaining budget the moment the next budget replenishment occurs.

Figure 5.1 illustrates some of the PBS arbitration behaviour that might occur. It depicts a time span of two budget replenishment periods in which a high and low priority application execute their tasks. The application that has access to the processor (and therefore consumes budget) is indicated by a white rectangle labeled either L for the low priority application or H for the high priority application. The enabling and finish times of tasks are depicted by respectively an incoming and outgoing arrow. The gray areas indicate time internals in which the processor is executing a task. The response time of a task is the time between its enabling and completion and is represented by a dotted arc. Observe that in the first period the high priority application does not have an enabled task until late in the period, such that it already started to idle away part of its budget. The second task execution of the high priority task cannot be completed in the first period, causing the high priority application to immediately claim the processor after budget replenishment in the second period, thereby additionally delaying the completion of the low priority application's task.

5.2 An exact PBS response time formula

Based on our definition of a PBS arbitration we can give the exact finish time of a task executions of both high and low priority applications. Subtracting an obtained task finish time from its activation time yields the response time of a task execution. We will make use of the following notation to describe the finish time formulas:

- HP: The high priority application.
- LP_k: Low priority application k, where the k-th low priority application is the k-th

low priority application that can spend budget during after a budget replenishment.

- $C(i)$: Worst-case computation time of task execution i of the high priority application.
- $C_k(i)$: Computation time of task execution i of the low priority application LP_k.
- B : Maximum budget of the high priority application.
- S_k : Maximum budget (also called slice size) of low priority application LP_k.
- P : Length of replenishment period. P is the sum of the budget of the high priority application plus the slice sizes of all low priority applications.
- ϕ : Offset that specifies the moment in time the first budget replenishment occurs. We assume $0 \leq \phi < P$.
- i : Sequence number of a task executions, where $i = 1$ designates the first task execution.
- $a_H(i)$ and $a_L(i)$: Activation/enabling time of the i-th task execution of respectively a high and low priority application.
- $f_H(i)$ and $f_L(i)$: Finish time of i-th task execution of respectively a high and low priority application.

5.2.1 Task finish times of a high priority application

If we want to know the finish time of a task execution i we need to know how much budget is available at the moment execution i can start. If we do not know the current remaining budget we cannot make an exact prediction of the finish time $f_H(i)$ of execution i, since depending on the remaining budget and the task's computation time we might have to wait for one or multiple budget replenishments.

We define the start time of a task execution i of the high priority application as the earliest time after the initial budget replenishment in which the task is enabled and all previous task executions are finished. We thus define the start time of execution i as:

$$s_H(i) = \max(a_H(i), f_H(i-1)) \tag{5.1}$$

Since the budget replenishment period is always of size P and we assume to know the offset ϕ for the first budget replenishment, we can calculate for any given time t the exact moment the last budget replenishment happened before the time t as:

$$r(t) = \lfloor \frac{t-\phi}{P} \rfloor \cdot P + \phi \tag{5.2}$$

To avoid complicating equations needlessly we will at times use an overloaded version of function r to which we pass a task execution i instead of a time t. We thus define $r(i) = r(s_H(i))$ such that $r(i)$ represents the start of the replenishment period in which task execution i can potentially start executing for the first time.

Let us assume the finish time of a task execution $i-1$ can be calculated. To determine how much budget is available for task execution i, we need to look at all the previous executions that consumed budget in the period that contains start time $s_H(i)$. This period p is thus the first replenishment interval in which task execution i could execute if there is enough remaining budget. Due to our definition of $s_H(i)$ we know that if task execution i can start in period p, all previous execution must have finished before or in period p. Since the budget is fully replenished at the start of each period, it is sufficient to look at all executions that finished later than time $r(i)$. Using the start time of task execution i and the replenishment time of the period in which i can start, we can calculate how much budget previous iterations spend since the last budget replenishment by adding up the time intervals during which these task executions occurred since the last replenishment:

$$u_H(i) = \sum_{j=1}^{i-1} \max(0, f_H(i-j) - \max(s_H(i-j), r(i))) \tag{5.3}$$

Once all low priority applications have completed their slices the budget of the high priority application will linearly decrease with time, even if no task executions are performed. Because we know how much budget the high priority application has spend on its previous task executions, we know the moment all low priority applications have finished their slices. We can thus define the exact time after which budget will be idled away as the time interval $[r(i) + P - B + u_H(i), s_H(i))$. If this interval is not empty, we know that the budget wasted by idling is equal to the length of this interval. We can thus express the final budget that a task execution i can use during its first period as:

$$b_H(i) = B - u_H(i) - \max(0, s_H(i) - (r(i) + P - B + u_H(i))) \tag{5.4}$$

We now know the exact moment when task execution i can start and for how long it is guaranteed to be able to execute once started. We can make two observations that help us determine the eventual finish time of the task execution:

- If the remaining budget in the first period in which execution i can run is sufficient to finish the task execution, we know the iteration can finish after performing $C(i)$ work.
- If the remaining budget is not sufficient to perform all work of the task execution, we will need to wait for one or more budget replenishments before completing the task execution.

Translating these observations to a formula gives us the following definition of the finish time of task execution i:

$$f_H(i) = \begin{cases} 0 & \text{if } i = 0 \\ s_H(i) + C(i) & \text{if } C(i) \leq b_H(i) \\ r(i) + P + \lceil \frac{C(i) - b_H(i)}{B} - 1 \rceil \cdot (P - B) + (C(i) - b_H(i)) & \text{if } C(i) > b_H(i) \end{cases}$$

5.2.2 Task finish times of a low priority application

When calculating the finish time of a task execution of a low priority application, we assume that the activation and finish times of all high priority task executions are known. Whereas there is but a single high priority application that can have multiple task executions, there can be multiple low priority applications, each executing its own tasks. Each low priority application LP_k has an assigned slice of size S_k, during which it can perform its task executions. Each of the low priority applications are served in a fixed order, meaning that if (LP_1, LP_2, LP_3) is the order in which low priority applications are served their slice, then LP_2 cannot execute before LP_1 has executed in the same period, and A_3 requires both other applications to execute first in the same period. A low priority application consumes its slice whenever its turn comes up, even if it has no tasks it wants to run.

Low priority applications can be preempted immediately when a high priority application has a non-zero budget and an enabled task. Because we define the replenishment period as $P = B + \sum_{k \in LP} S_k$, all low priority applications are guaranteed to have the opportunity to consume their whole slice within P time. Interference of the high priority applications can however still introduce jitter in the start and end times of the slices with respect to previous periods, which in turn has an effect on the finish times of task executions of low priority applications.

Assume we know the finish time of all earlier executions of a low priority application LP_k. If we want to determine the exact finish time of a task execution i, we must know exactly how long the high priority application will run during its execution. Every task execution of a low priority application has an activation time $a_L(i)$. Since a task execution cannot start before previous executions are finished we can define the start time $s_L(i)$ as the earliest time task execution i is enabled and all previous executions are completed. We thus define the potential start time of task execution i as:

$$s_L(i) = \max(a_L(i), f_L(i-1)) \tag{5.5}$$

Tasks of an application LP_k can only be executed when its slice is being consumed. Before a specific low priority application can consume its slice of size S_k, all preceding

slices of other low priority applications must have been completed. The minimum time it takes until the slice of application LP_k can execute is the sum of the slice sizes of all preceding applications. However, if the high priority task preempts at any time during the execution of any of the preceding slices, the wait time until LP_k can execute its tasks will increase. We are therefore interested in knowing for some time t how long the high priority application executes in the interval $[r(t), t)$. We can define a formula for calculating this time by making the following observations about the behaviour of high priority task exsections:

- If a high priority execution starts after moment t or finishes before time $r(t)$ it does not execute in the time interval we're interested in.
- A task execution i of a high priority application that finishes before moment t has executed the whole time from either the start of the period or the start of the task's execution (whichever is larger), until the moment that it finishes.
- When task execution i start within the same period as time t but finishes later then time t, the time it can execute from its start time until moment t is dependant on whether or not the task execution runs out of budget before time t. We know task execution i had only $b_H(i)$ budget when it started executing.
- When task execution i starts in a previous period and will finish after time t, the task execution either executes the whole time up to time t, or runs out of budget before time t. Since the maximum budget is B we know it can at most execute for B time.

We can translate these observations into a formula that calculates the time $h(t)$ representing the time the high priority application has been executing in the interval $[r(t), t)$ as:

$$h(t) = \sum_{i=1}^{\infty} \begin{cases} 0 & \text{if } f_H(i) < r(t) \text{ or } t \leq s_H(i) \\ \max(0, f_H(i) - \max(s_H(i), r(t))) & \text{if } r(t) \leq f_H(i) < t \text{ and } s_H(i) < t \\ \max(0, \min(b_H(i), t - s_H(i))) & \text{if } r(t) \leq t \leq f_H(i) \text{ and } r(t) \leq s_H(i) < t \\ \max(0, \min(B(i), t - r(i))) & \text{if } r(t) \leq t \leq f_H(i) \text{ and } s_H(i) < r(t) \leq t \end{cases}$$

If execution of a low priority application's slice is delayed due to interference of the high priority application then the time at which the end of the low priority slice is reached will also be delayed. Since the slice is thus active until a later moment in time, additional preemptions by the high priority application can occur, delaying the end of the slice even further. We can calculate the time until the slice of application LP_k can start by finding the maximum of a recursive equation that takes into account this interference of the high priority application. For time t in which we want to start the work that without preemptions would take w time, the time that all work is actually finished when we take the preemptions by the high priority application into account is $d(t, w)$. Note that care should be taken to ensure that t and w are chosen in such a manner that the actual finish time $d(t, w)$ always lies in the same replenishment period as t. We define

the function $d(t, w)$ as:

$$\begin{aligned} d_0(t, w) &= t + w \\ d_n(t, w) &= t + w + h(d_{n-1}(t, w)) - h(t) \end{aligned} \tag{5.6}$$

The maximum is reached when $d_n(t, w) = d_{n+1}(t, w)$. This equation always has a finite maximum since the maximum interference of the high priority application within a period is limited by its maximum budget B. We can thus calculate the time until the slice of application LP_k can start in any given period by setting t as the replenishment time that starts the period whose behaviour we're interested in and setting w to the sum of the slices of all preceding tasks.

Once a slice of an application is started, it will be consumed regardless whether there are tasks executing or not. If task execution i wants to execute in its first period it not only has to be enabled and any previous executions have to be finished, but the application's slice should also not have ended. We can make the following observations:

- If the start time of the task execution lies before the start time of its slice, we know the previous task execution completed before the current period, since none of the task execution of the application can have executed yet within this period. Task execution i can thus execute for the full slice.
- If the start time of the task execution lies somewhere between the start and end of its slice, task execution i can execute for a time that is equal to the length of the interval between $s_L(i)$ and the end of the slice, minus the time the high priority application will executes during this interval.
- If the start time of the task execution lies after the end of the slice - either due to arriving too late or the previous task execution needing the whole slice - task execution i cannot execute at all in its first replenishment period.

We can thus calculate the remaining slice budget in the first period of task execution i as:

$$b_L(i) = \begin{cases} S_k & \text{if } s_L(i) < d(r(i), \sum_{j=1}^{k-1} S_j) \\ (d(r(i), \sum_{j=1}^{k} S_j) - s_L(i)) - & \text{if } d(r(i), \sum_{j=1}^{k-1} S_j) \leq s_L(i) \\ (h(d(r(i), \sum_{j=1}^{k} S_j)) - h(s_L(i))) & \text{and } s_L(i) < d(r(i), \sum_{j=1}^{k} S_j) \\ 0 & \text{if } d(r(i), \sum_{j=1}^{k} S_j) \leq s_L(i) \end{cases} \tag{5.7}$$

Due to the fact that every low priority application is guaranteed to get an opportunity to execute its allotted budget every period, we can derive the exact finish time of task execution i once we know how long the task can execute in its first period and what the

amount of interference is by the high priority application in the execution's last period. We can define the following possible scenarios as:

- If there is not enough remaining slice budget $b_L(i)$ at the start time of the task execution such that it can be completed in its first period, we know that we need $0 \leq q$ additional full periods plus one final period to finish all work for task execution i. The amount of work we will have to perform in the last period is equal to $C_k(i) - b_L(i) - q \cdot S_k$. Note that we do not care about the exact time task execution i can start executing in the periods that are not its last period, since knowing how much work it can perform in these periods is enough.
- If task execution i has enough remaining budget such that it can finish within the first period in which it can execute, we do need to know the actual moment in time task execution i can start executing. This time cannot be before task execution i is enabled and the previous executions are completed. This condition is met from time $s(i)$ onwards. However, task execution i can also not start executing until its slice is active. We can discern two sub-cases when looking at the relative slice position of a low priority application:
  - The start time $s_L(i)$ lies before the beginning of its slice. In this case task execution i can start immediately once the slice is started, since this implies that all preceding task execution have finish in earlier periods.
  - The start time $s_L(i)$ lies on or after the beginning of the slice. Since $s_L(i) = \max(a_L(i), f_H(i-1))$ we can start task execution i at time $s_L(i)$. Note that we're never in this case if $s_L(i)$ lies after the end of the slice, since in that case the remaining slice budget $b_L(i)$ would not be enough to complete the task execution.

We can translate each of these scenarios into a start time and an amount of work that has to be completed. We can then use the recursive formula $d(t, w)$ to calculate at which time all work is completed when taking into account preemptions of the high priority application. The resulting time is the finish time of task execution i:

$$f_L(i) = \begin{cases} 0 & \text{if } i = 0 \\ d(r(i) + P + q \cdot P, \sum_{j=1}^{k-1} S_j + \\ C_k(i) - q \cdot C_k(i) - b_L(i)) & \text{if } C_k(i) > b_L(i) \\ d(r(i), \sum_{j=1}^{k-1} S_j + C_k(i)) & \text{if } C_k(i) \leq b_L(i) \text{ and} \\ & s(i) < d(r(i), \sum_{j=1}^{k-1} S_j) \\ d(s(i), C_k(i)) & \text{if } C_k(i) \leq b_L(i) \text{ and} \\ & d(r(i), \sum_{j=1}^{k-1} S_j) \leq s(i) \end{cases}$$

Where $q = \lceil \frac{C(i) - b_L(i)}{S_k} - 1 \rceil$.

5.3 A worst-case PBS response model

This section will introduce the dataflow response model that we will use to capture the worst-case timing behaviour of tasks executions under PBS arbitration.

5.3.1 Approach

The multi-rate response model (proposed in [4] and improved in [5]) accurately captures the worst-case temporal effects of TDM arbitration. Although it is much less pessimistic, the multi-rate response model resembles the latency-rate response model in that it can essentially be divided into two parts; a latency part and a rate part. By looking at both the latency and rate parts of the multi-rate response model we can come up with an approach to modify the multi-rate model such that it can be used to capture the worst-case effects of PBS arbitration.

The latency part of a response model expresses the worst-case arbitration time of a runtime scheduler, meaning the worst-case time a task may have to wait until its application can spend its budget. Under TDM arbitration with a replenishment period P and a task of slice size S this arbitration time amounts for both the latency-rate and multi-rate response model to $P-S$ time. For PBS arbitration the worst-case arbitration time of a high priority application's task is 0, since we assume we can always pre-empt the running low priority application. Wether the application actually has remaining budget to spend is a concern for the rate part of the model. A task of a low priority application suffers its worst-case arbitration time when its enabling occurs just after the application consumes the last of its budget, and the high priority task still has its full budget available. In such a scenario the high priority application is able to spend its full budget twice before allowing the low priority application to execute. The worst-case arbitration for a low priority application under PBS arbitration is therefore $P-S+B$. Figure 5.2 illustrates this scenario.

The rate part of a response model must capture the time spend executing and waiting on budget replenishments. The multi-rate model uses tokens to represent available budget, which allows it to include the effects of previous task execution on a task's response time. Each time a budget token is consumed a new budget token will become available after P time. This replenishment behaviour is conservative towards the worst-case budget replenishment behaviour of PBS arbitration, since by definition both a high and low priority task will have its budget replenished after waiting for a time equal to schedule scheduler's budget replenishment period.

5.3.2 Model construction

We propose to use a modification of the multi-rate response model defined for capturing the effects of TDM arbitration as base for our response model for capturing worst-case

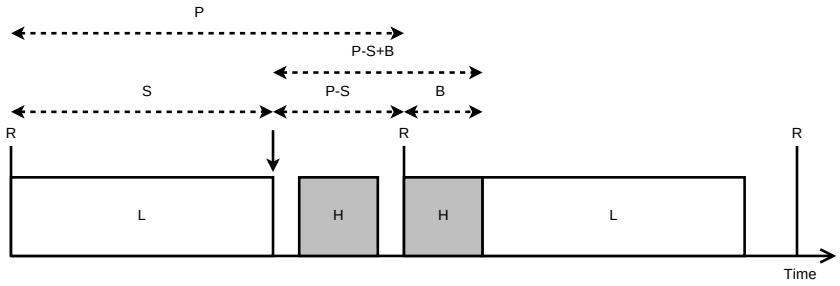


Figure 5.2: Worst-case task enabling for a low priority application under PBS arbitration.

PBS arbitration effects. The construction of the response model for PBS is exactly the same as for TDM, except for the execution time of its latency actor (the actor labeled L in figure 3.6):

- In the PBS response model of a tasj of a high priority application the execution time of the latency actor is 0.
- In the PBS response model of a task of a low priority application with slice size S that should run on a processor with replenishment period P and must share the processor with a high priority application with a budget of B, the execution time of the latency actor is $P - S + B$.

Other aspects regarding usage of the modified multi-rate model as a PBS response model - such as the composition of various response models and support for static ordering of tasks - remain the same as for the original TDM model. We therefore refer for more details about these aspects to [5].

5.3.3 Conservativity

We will not provide a formal proof of the conservativity of the proposed PBS response time model due to time constraints. However, the multi-rate response model is proven to be conservative under TDM arbitration, and the change to the latency node of the response model are such that we consider the likelihood of the PBS multi-rate model being conservative very high.

5.4 Summary

We have defined our flavour of PBS arbitration under consideration and introduced a response model that is able to accurately capture the worst-case effects of resource arbitration under PBS.

Chapter 6

PBS scheduling strategy

Scheduling is the act of determining when an application or task can make use of a certain resource. We can discern two different type of scheduling approaches based on the moment in time in which scheduling take place; during compile time or during runtime. *Compile time scheduling* is also called offline scheduling, referring to the fact that scheduling decisions are made in advance before the system's applications are actually executed. *Runtime scheduling* is also known as online scheduling. A runtime scheduler takes scheduling decisions while the application of the system are actually running. The choice between runtime scheduling versus compile time scheduling is a tradeoff between flexibility and performance: A runtime scheduler can take into account current information about the state of the system, such as which applications are running and the current resource requirements of these applications, but these decisions must be made fast in order to not waste too much processor time. A compile time scheduler cannot base its scheduling decisions on actual runtime information, but because it has virtually limitless time to make its scheduling decisions, it can base its decisions on significantly more complex analysis methods.

In this thesis we will use a combination of compile time and runtime scheduling such that we can have the advantages of both. The tasks of an application are scheduled using compile time scheduling: We determine the task to processor mapping for an application's tasks, and the static order in which the tasks execute that are mapped to the same processor. The decision which application gets access to a processor at any given time is made by the local scheduler of the processor during runtime. This allows us to perform all complex decisions such as determining the minimal budget that satisfies an applications timing requirements at compile time, while still allowing the system to consist out of an arbitrary combination of applications that can start and stop independently. Note that using a combination of both runtime and compile time scheduling also introduces the disadvantages of both: The minimum reservation requirements of an application determined during compile time scheduling cannot take into account whether or not an application actual needs its allotted processing time in each replenishment period, and

can therefore waste processor resources. Similarly, the decision which application can execute on a processor still has to made fast, limiting the complexity of the runtime scheduler that determines which application to can execute. However, these disadvantages are relatively minor; the more accurate minimum reservation bound obtained by temporal analysis can more than compensate for the small over-reservation that may occur in some periods, and the complexity restrictions of runtime schedulers are not an actual problem since determining which application can execute is not a complex choice anyway.

Running a combination of applications simultaneously in a environment with shared resources requires that we make certain decisions about the applications and the system in which they will run. These decisions include:

- Task to processor mapping, that is, determining for each application which task executes on which processor.
- Static ordering of the executions of tasks, that is, determining the order in which tasks from the same application that are mapped to the same processor are executed.
- A choice of runtime scheduler type per processor (i.e. TDM, PBS), and its parameters such as replenishment period.
- A number of decisions for each application arbitrated by a specific runtime scheduler:
  - The choice of budget for each application.
  - The choice of priority level for each application.

We will refer to the combination of an application's task-to-processor mappings and static order decisions as a *schedule* of an application. The choices for application schedules, runtime scheduler type and its replenishment period can have a significant impact on the timing behaviour of an application, and therefore an application's minimum resource requirements. We would like to assign to each application only the minimum amount of resources needed such that it can meet its timing requirements. It is therefore important that we make a good choice when making these decisions.

In chapter 4 we have already discussed one of these decisions; the choice of runtime scheduler type for the system's processors. We argued that using PBS arbitration makes sense when we know we have a combinations of applications in which one application has tight timing requirements and the other applications had more relaxed timing requirements. This chapter will propose a strategy how - given that we will be using PBS arbitrations as our scheduler of choice - we can try to make good choices for the remaining decisions, such that we end up with acceptably low minimum resource requirements for each application.

6.1 Optimization criteria

We strive to let each application use the least amount of processor time as possible, as this will increase the amount of processor time available to other application. We express an application's relative amount of processor usage as its *reservation*, which is the ration between the application's allotted budget B and the processor replenishment period P, such that $R = B/P$.

An embedded application often has to execute its tasks on more than one processor, where each processor may be optimized for a specific type of computation. If for such an application we reduce its reservation on one processor, we may be required to increase the application's budget on other processors in order to guarantee the application's timing requirements can still be met. Determining for which processor a decrease in reservation is more important is a choice that depends on knowledge outside the scope of our scheduling strategy. We therefore assume to know for each processor in the system a weight in the interval $[0, 1]$ that expresses the relative cost of resource reservation on that processor. Our proposed scheduling strategy will try to minimize the sum of weighted processor reservations for each application.

6.2 Scheduling methodology

Determining optimal application schedules and scheduler parameters is complex. Although there exist a few guidelines that can help narrowing down the number of reasonable choices, the actual effect of a scheduling decision on our optimization criterion is hard to predict, since every other scheduling decision can influence the net required minimum reservation. We therefore divide our search for scheduling decisions in a number of phases, in which in each phase we try to make reasonable scheduling decision.

We propose the following heuristic to determine application schedules and scheduler parameters given a set of application and their timing requirements:

1. We determine which application gets high priority based on the timing requirements of the applications.
2. We find one or more schedules for the high priority application.
3. We derive combinations of budget and replenishment periods that allow us to minimize the required reservation of the high priority application and give us options to potentially decrease the required reservation low priority applications.
4. We determine an upper bound to the worst-case number of context switches that can occur based on the schedule of the high priority application, which we use to eliminate replenishment periods for which a processor might spend too much time context switching.

5. We find schedules for the low priority applications and determine which of the combinations of budget replenishment periods that we found earlier yields the lowest required reservations.

We will examine each step in more detail in the following sections.

6.2.1 Determining the high priority application

The reasoning that lies at the base of our scheduling strategy is the following: Applications with tight timing requirements are more sensitive to scheduling decisions than applications with more relaxed timing requirements, since the maximum time the application has to complete its tasks is smaller. It therefore makes sense to base our scheduling decisions primary on the requirements of the application with tightest timing requirements.

PBS arbitration allows us to eliminate the arbitration time of the high priority application at the expense of increased arbitration time for low priority applications. This tradeoff is attractive if the high priority application would require significantly less over-reservation under PBS arbitration than it would require under other arbitration schemes. Since the application that requires the most over-reservation tends to be the application with the tightest timing requirements, we assign this application the high priority.

6.2.2 Determining high priority application schedules

We assume the system can have different processor types, and that each task in an application's task graph is annotated with information specifying on which type of processor the task must execute. We decide which task executes on which processor and in which static order by using an offline scheduler that tries a processor mapping for a task, and afterwards verifies whether this mapping still allows the timing requirements of the application to be satisfied. Timing verification of an actor mapping is done by creating a temporal analysis graph, and assumes for each yet unmapped actor a mapping that will give the best possible worst-case response time, such that valid schedules are not unnecessarily rejected. Note that because we have not determined values for local scheduler replenishment periods and application budget yet, the analysis graph for verifying task mappings of the high priority application cannot yet take into account the effects of resource arbitration. We can create an analysis graph without assumptions about resource arbitration effects by choosing an arbitrary replenishment period for each processor and settings the application's budget such that it is equal to the processors replenishment period. Doing this for our proposed multi-rate PBS response model will result in an analysis graph that consist of response models whose response time is equal to the task's worst-case computation time.

Figure 6.1 illustrates the scheduling procedure as a flow diagram. Task mappings are

tried in the order as defined by a precedence graph. This precedence graph is derived from the application's task graph by removing all arc with non-zero delay, such that the task of an actor without incoming arcs is a task that has all required input data and can therefore can be executed. If a processor mapping does not cause the application to violate its timing requirements it is kept, the task's actor will be removed from the precedence graph and a next task mapping will be tried. If a task mapping does cause a timing violation it is undone and a mapping to another processor is tried, until we either find a successful mapping or have tried all possible processor mappings for that actor. If we unsuccessfully tried all mappings of an actor then the previous actor mapping is undone an this previously mapped actor is tried on a different processor, until we have either found a valid mapping for each application or tried all possible mappings and fail to find a schedule.

6.2.3 Deriving suitable budget replenishment periods

The timing requirements of an application defines its maximum production period, which represent the maximum time the application can take for finishing one iteration without violating it timing requirements. We can exploit the fact that under PBS arbitration a high priority application only has to consume its budget when it has actual tasks to execution to derive budget and replenishment periods that allow the processor reservation of the high priority application to be the lowest value theoretically possible.

By setting the replenishment period of each PBS processor to be equal to the maximum production period of the high priority application, we can ensure the application is able to complete its iteration within its maximum production period by giving it a budget on each processor equal the computation times of the tasks mapped to the processor. This choice of application budget and processor period will give the lowest reservation possible for the application's schedule: Using either a larger budget replenishment period or smaller budget means the application does not have sufficient budget to perform a full iteration each production period, while using a smaller budget replenishment period or larger budget would increase the application's required resource reservation.

We can find other combinations of budget and replenishment periods for the high priority application that allow for the same minimum processor reservation by dividing both production period and sum of task computation times by their common factors. This essentially divides the maximum production period of the application into a number of smaller equally sized parts, in which the application gets a fraction of its original budget. The combinations of budget and periods obtained in this fashion might not allow the high priority application to satisfy its timing requirements, since that requires that the application spends exactly the same amount of budget in each part of the production period, something which keeps getting more and more unlikely with an increasing number of parts. However, in case some of these reduced replenishment period and budget combinations are valid for the high priority application, then using this smaller

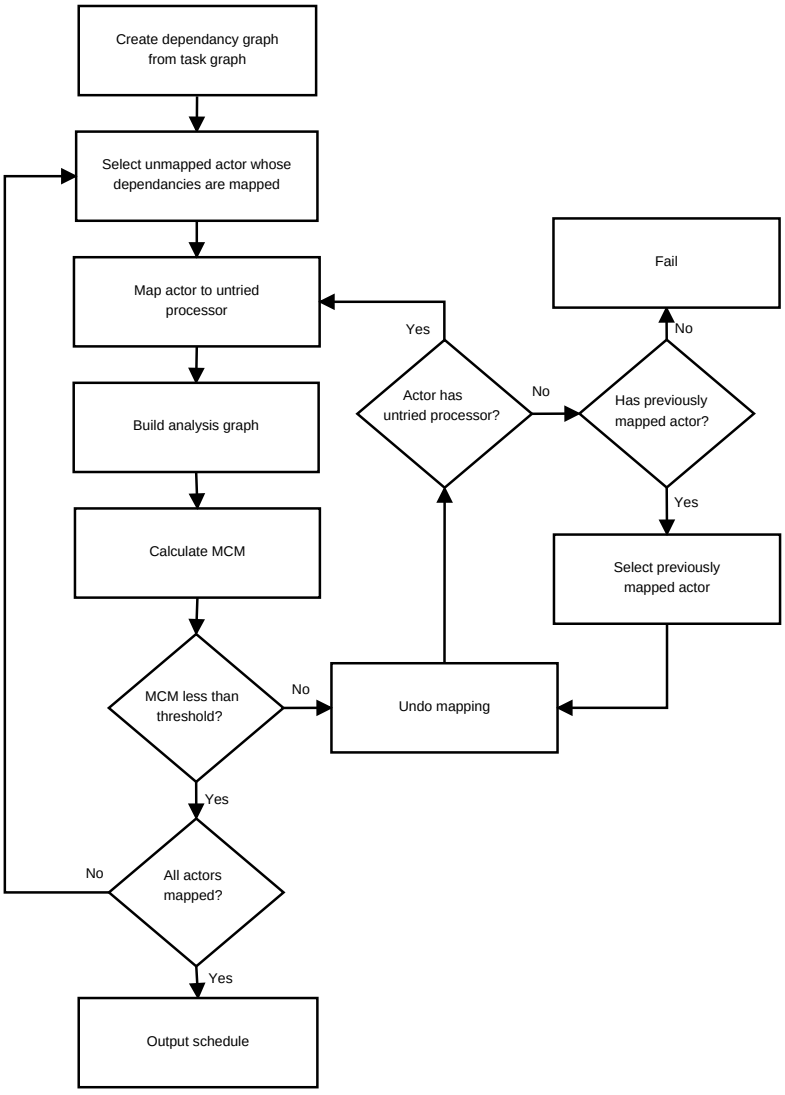


Figure 6.1: Simplified scheduling flow

replenishment periods may allow for a reduced required processor reservation for the low priority applications.

6.2.4 Bounding the worst-case context switch overhead

Every time the active application of a processor is changed we need to save context information of the previously running application and restore earlier saved context information of the newly active application. The time needed to perform these operations is called the context switch time of a processor. When performing a context switch the processor cannot execute tasks, which decreases the total available processor time for applications. Overestimating the time we spend on context switches means we underestimate the available processing time on a processor, essentially wasting resources. We therefore want to tightly bound the maximum number of context switches that can occur per replenishment period.

For a TDM scheduler the number of context switches that have to be performed per replenishment period is equal to the number of applications that run on the processor, since each application's slice is started exactly once per replenishment period. Bounding the worst-case number of context switches that can occur under PBS arbitration is less straightforward, since a high priority application can potentially cause a context switch every time it is not active on the processor when one of its task's becomes enabled. In order to still derive an upper bound on the maximum number of context switches that can occur per replenishment period, we must look at the the processors mappings and static order of the task of the high priority application.

We can annotate the task graph of an application with the static order information of the high priority application obtained during scheduling. Doing this creates a dataflow model that captures the execution order of the application's tasks as defined by both the inter-task data dependencies and the schedule's mapping decisions. By converting this model to single-rate we ensure that each task execution during one iteration of the application is represented by its own actor. We can derive an upper bound to the maximum number of context switches the high priority application can cause on a specific processor by examining the actors that represent task executions on the processor of interest. We define all actors representing tasks mapped to the processor of interest internal actors, and define all other actors to be external actors. Each arc that connects an external actor to an internal actor represents a data dependency between two tasks that are executed on different processors. If we count how many of the internal actors have incoming arcs from external actors, we obtain an upper bound on the maximum number of times a task of the high priority application can become enabled while the high priority application is not executing. We can refine our upper bound further by realising that each task in our modified single-rate task graph will fire exactly once per iteration. This means that two tasks executing on the same processor cannot have to wait both on data produced during the same iteration by the same external task. This implies that if two internal actors both have an incoming delayless arc from the same

external actor, then only the execution of the first task in the static order can be enabled by its data arrival such that it causes a context switch. The second task already has the required data thus can immediately execute after the first task is complete. However, note that if an external incoming arc has a non-zero delay, this delay represents data produced during earlier iterations, which means that both tasks could receive data from the same external actor during a different moment in time, and therefore both could force a context switch.

We would like to obtain a dataflow model in which any of the previously discussed redundant incoming external arcs are removed, such that only external arcs that can potentially cause a context switch can be counted. By realising that each internal actor is connected to each other by static order arcs, we can view the redundant external arcs as 'shortcuts' from an external actor to an internal actor that is also reachable through an internal actor earlier in the static order. We can obtain a dataflow model without these redundant external arcs in the following manner:

1. Annotate the application's task graph with static order information of a schedule and convert it to single-rate.

2. If there exist two arcs between the same source and sink actors then keep only the arc with the least delay, since this arc represents the tightest constraint on the task executions.

3. Remove all remaining arcs with non-zero delay, storing them for future use. We obtain an acyclic graph, since each cycle in a dataflow graph must have at least one delay.

4. Calculate a minimum equivalent graph. A minimum equivalent graph is a graph with the minimum amount of edges that has the same transitive closure - and thus reachability - of the original graph [14].

5. We add the previously removed arcs with non-zero delay back to the graph, since we assume these arcs represent the arrival of data that can occur at any time during an iteration, and therefore can cause context switches.

By counting the number of incoming external arcs for each internal actor in the obtained dataflow model we obtain an upper bound on the maximum number of task enablings per iteration while the high priority application is not running. Since we have seen in section 6.2.3 that we chose our replenishment period and high priority application budget such that we can at most execute one iteration per period, the maximum number of context switches per period on a processor under PBS arbitration is equal to two times the maximum number of high priority task enablings per iteration (one at the start of a task execution and one at the end), plus one context switch for each application that can execute on the processor. We use this obtained upper bound on the maximum number of context switches to calculate a minimum required replenishment period in which the time spent context switching does not exceed our acceptable limit.

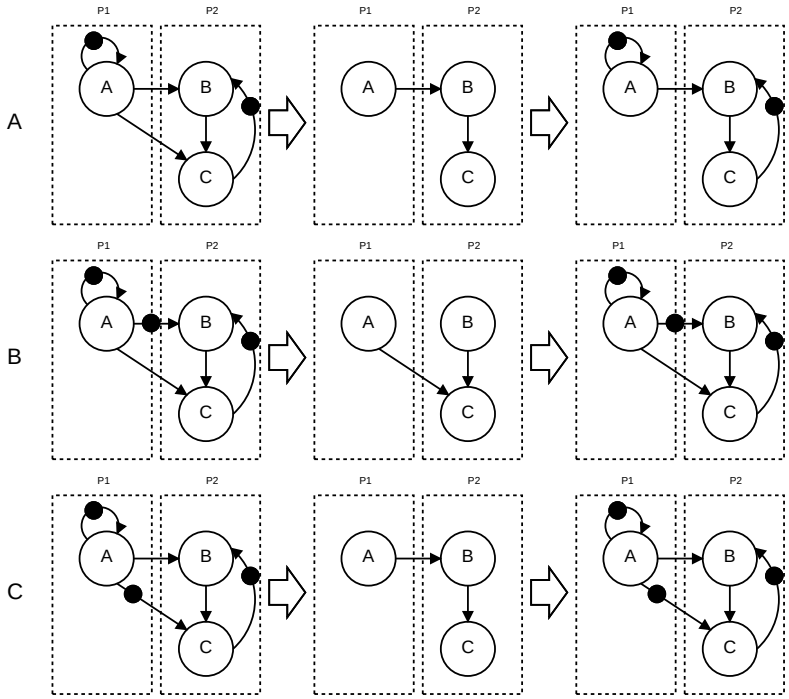


Figure 6.2: Various scenarios in which we try to remove redundant arcs to improve maximum context switches bound.

Figure 6.2 illustrates three scenarios in which we apply minimum equivalent graph reduction to remove redundant external arcs. Assume that the processors $P1$ and $P2$ have to be shared by two applications; one high priority application and one low priority application. In scenario $S1$ we are able to remove one redundant arc form the modified task graph of the high priority application, since actor C has already received its data from actor A by the time actor B starts to execute. The bound on the number of context switches in scenario A is therefore $0 + 2 = 2$ on processor $P1$ (zero due to external data dependencies, but one for each application) and $1 \cdot 2 + 2 = 4$ on processor $P2$ (2 due to actor B and one for each application). In scenario $S2$ actor B can be finished before actor A is finished, thereby causing context switches for both actor B and actor C, meaning the bound on the maximum number of context switcher per period is $2 \cdot 2 + 2 = 6$ on processor $P2$. In scenario $S3$ our technique overestimates the number of actors that can cause a context-switch. The arc between actor A and C cannot be removed using minimum equivalent graph reduction due to the fact that it as a non-zero delay. However, in this scenario C can never execute before B under self-timed execution due to its static order constraint, and therefore C will not actually cause a context switch. The estimated number of context switches on $P2$ is therefore $2 \cdot 2 + 2 = 6$.

Although our methodology of using minimum equivalent graph reduction on only the

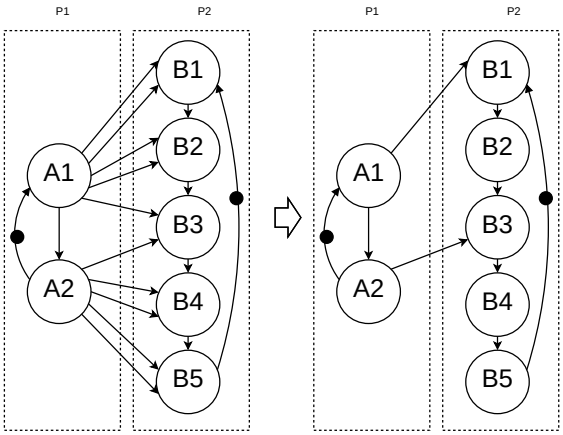


Figure 6.3: Removing redundant arcs to improve maximum context switches bound.

delayless arcs in a modified task graph is fairly primitive, it can nevertheless significantly improve the bound on the maximum number of context switches when we compare its result to the bound we would get by simply counting all actors with external incoming arcs. Figure 6.3 depicts a dataflow model before and after our procedure to reduce redundant arcs. Note that the original dataflow model is in the shape typically obtained by converting a multi-rate model to single-rate. Without removal of redundant arcs we might assume that ever actor mapped to processor $P2$ might cause a context switch. However, when we remove the redundant arcs with our proposed approach we immediately see that there are at most two actors whose task can get enabled while the high priority application is not executing.

6.2.5 Minimizing low priority application reservation

Once we have derived suitable budget replenishment periods and high priority application budgets that minimize the high priority application's required reservation, we try to determine which of the replenishment period combination can minimize the required reservation of the low priority task.

We start by searching for each low priority application a schedule for each of the processor-period combinations. This scheduling is performed with the application's budget set to the maximum allow budget (which typically is defined based on the number of application the system must run simultaneously). If we do not manage to find at least one period combination for which every low priority application has found a schedule then the interference of the high priority application is must likely too large to satisfy the timing requirements of all low priority applications. We can either try to increase our maximum allowed reservation, or restart the scheduling flow using a new high priority application schedule, which may interfere less with the low priority applications.

Assuming there is at least one period combination for which every low priority applica-

tion has found a schedule, we can start searching for the period combination that allows for the largest reservation reduction. We do this by sorting the processors in order of importance as indicated by their weight, such that the processor whose reservation cost is largest will be reduced first. For each processor we will performing a binary search to find the minimum reservation under which at least one period combination still has a valid schedule. We perform this search for each processor, reducing the reservation of the low priority applications on each processor to the minimum that does not violate their timing requirements. Once the reservation of every processor has been reduced, the period combinations with the remaining valid low schedules are stored, together with high priority application's schedule and minimum reservations.

Figure 6.4 depicts a flow diagram of the various steps in our scheduling strategy.

6.3 Runtime complexity

The runtime complexity of the scheduling strategy is determined by the runtime complexity of its subproblems. These some of these subproblems have an exponential worst-case behaviour, such as determining application schedules and conversion of a dataflow model to single-rate. Any scheduling strategy will thus always be exponential in its worst-case behaviour. However, exponential worst-case behaviour does not mean a scheduler strategy in not usable in practice, since the situations in which full worst-case behaviour is realised may be unlikely to occur.

One aspect that will have an significant impact on the actual performance of our scheduling strategy is the time it takes to verify whether a given application's (partial) schedule respects an application's timing requirements. Performing this verification requires us to build a temporal analysis graph based on the multi-rate response model and convert the obtained dataflow model to single-rate. A multi-rate response model is known to potentially cause a very large increase of actors and arc when converted to singe-rate [5]. The significance of this increase depends on the Greatest-Common-Divisor (GCD) between an application's budget and the task computation times, where a small GCD can cause a very large increase of actors and actors for the single-rate model. The likelihood that we have a combination of task execution times and initial application budget that yields a low GCD during normal scheduling is not very high, since an application's initial budget is often a number with many common factors, and worst-case task computation times are usually rounded upward to a nearest 10 or 100 nanoseconds. However, when we reduce an application's processor reservation to its minimum value we are performing a binary search through the possible budget values for an application. It is reasonably likely that during this search we will try at least one budget value that is either prime or otherwise has a small number of factors. The reservation reduction of low priority application is therefore the most computationally expensive part of our scheduling strategy. To give an indication of it's performance; the scheduling strategy experiment discussed in chapter 7 took roughly 30 days to complete, of which almost all time was spend on

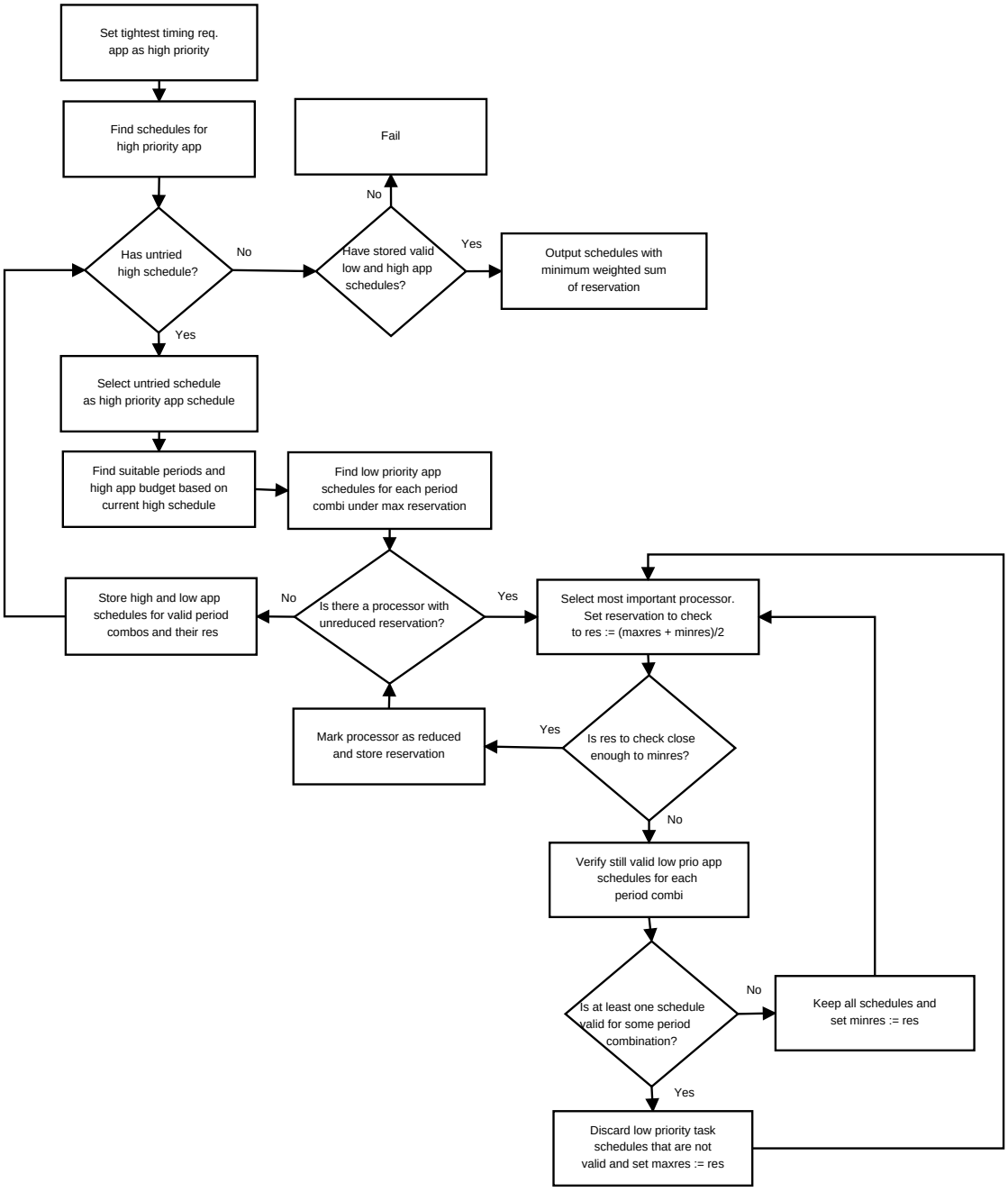


Figure 6.4: Flow representation of scheduling strategy

finding the minimum reservation for the low priority application.

6.4 Summary

We have introduced a scheduling strategy that given a set of applications and their timing requirements can obtain application schedules, scheduler parameters and minimum required resource reservation for all the applications in the system. Chapter 7 contains experiments comparing the results obtained with our proposed PBS approach to the results obtained by TDM scheduling.

Chapter 7

Experimental Results

We have performed various experiments during the course of this project. Some of these experiment were performed to gain insight in the behaviour of the various response models in literature, while others where to gain an understanding of the relation between a choice of a processor's replenishment period and an application's minimum reservation. Finally, we performed an experiment comparing the processor reservation obtained by our proposed response model and scheduling strategy for PBS arbitration to the results obtained by using a state-of-the-art response model and scheduling strategy for TDM arbitration.

7.1 Service-over-time

One of our first experiments entailed comparing the predicted worst-case task execution behaviour predicted by the various PBS response models. For this experiment we implemented the behaviour of a number of the more interesting response models, after which we used each implementation to calculate the finish times of simulated task enablings given our choice of scheduler parameters. By plotting the amount of finished task executions versus time we obtain each model's predicted service over time.

We assume that we have a processor under PBS arbitration with a replenishment period of 10. Both a high and a low priority application want to execute on their respective tasks on this processor. Both applications have a budget of 4, and the worst-case task computation time of the task of both applications is 3. We discern two scenario's for the task arrivals: In the first scenario both high and low priority task enablings arrive at time 0, representing a burst of input data becoming available. In the other scenario task enabling arrival periodically with a period of 10. We will consider the predicted worst-case behaviour by four different response models for PBS arbitration:

- Our proposed modified multi-rate response model. Abbreviated in the figures as

MR.

- The latency-rate response model, under assumption that we change its latency node similar to our proposed model. Abbreviated as LR.
- The latency-cyclic-rate response model, also under assumption that we change its latency node similar to our modified MR model. We abbreviated this model by LCR.
- Cai's response model. Abbreviated by JC.

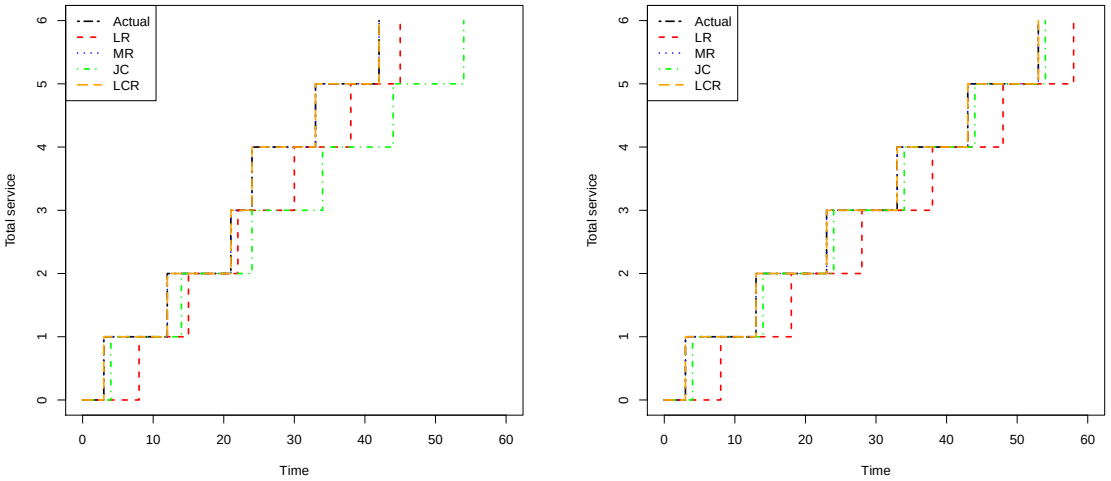
Additionally, we will include in each service comparison the actual service over time a task can realise, such that we get a better idea about the pessimism of the various response models.

Figure 7.1 shows the modelled worst-case service over time for the high priority task, in which task enablings either arrive in a burst at time 0 (figure 7.1a) or periodically (figure 7.1b). One of the things we can immediately observe in these service comparisons is that the multi-rate gives the same expresses the same service over time as the latency-rate-model. This is not surprising, since the multi-rate model is actually a generalization of the latency-rate model. Furthermore, both the multi-rate and latency-cyclic-rate models accurate capture the actual service over time of the high priority application in our two scenarios. Another noteworthy observation is that figure 7.1a shows that Cai's model can actually be more conservative than the latency-rate model. This is caused by the fact that Cai's model assumes that if a task's computation time C is less than an application's allotted budget B, the application can only execute $\lfloor \frac{B}{C} \rfloor$ tasks in each period before, after which it has to wait for a new budget replenishment before it can start on a next task. For our choice of budget and task computation times, this assumptions essentially wastes a quarter of an application's allotted budget each period.

Figure 7.2 depicts the modelled worst-case service over time for the low priority application. Note that we assume that a low priority task execution can suffer from interference from the task executions of the high priority application. The actual timing behaviour of the low priority application is therefore calculated by our in chapter 5 introduced exact response time formula. We can observe that both in figure 7.2a and in figure 7.2b none of the response time models give an exact prediction of the actual task's response time. This is not surprising, since the response time models do not know if or when the high priority application's task executes, and must therefore make worst-case assumptions about its behavior. Note that again the most accurate response times are given by the multi-rate and latency-cyclic rate models.

7.2 Period versus minimum reservation

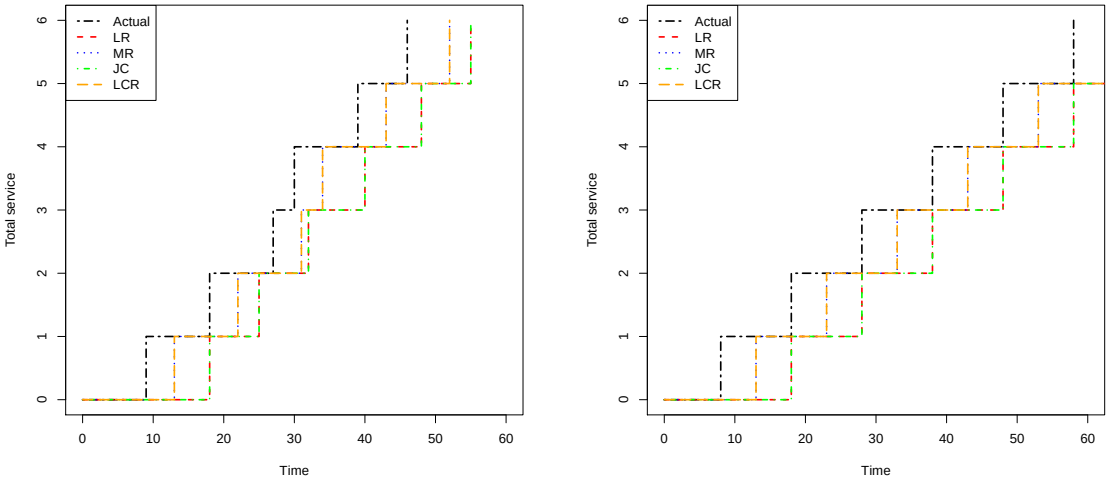
We have performed a number of experiments to gain a better understanding of the effect the choice of replenishment period can have on an application's required slice size. Given



(a) Burst enabling of tasks at time 0.

(b) Periodic enabling of tasks with period 10.

Figure 7.1: Total service over time for a high priority task under PBS arbitration.



(a) Burst enabling of tasks at time 0.

(b) Periodic enabling of tasks with period 10.

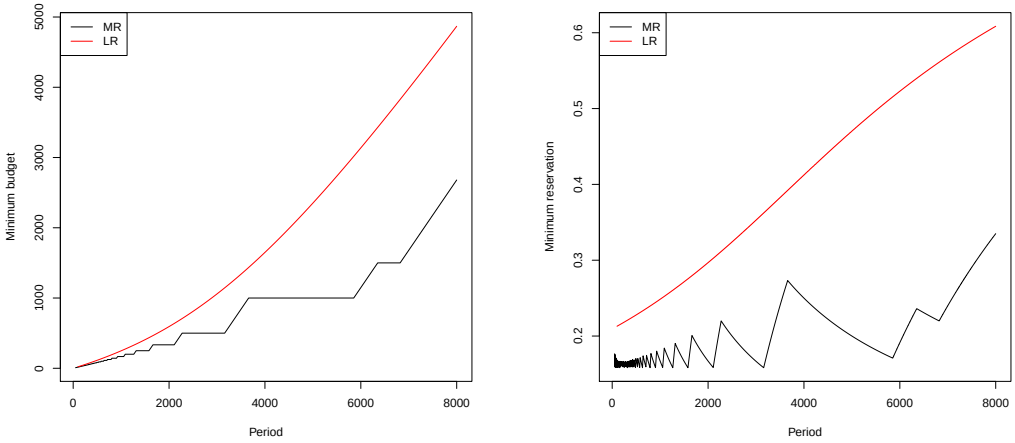
Figure 7.2: Total service over time for a low priority task under PBS arbitration.

an application and default choices for replenishment period and budget, we are interested in exploring the effect that increasing or decreasing a processor's replenishment period has on the application's minimum required budget.

The application whose behaviour we will explore during this experiment is a WLAN transceiver under TDM arbitration. This application is schedules on three processors, named EVP, ARM and SWC. Each processor has a replenishment period of 2000 and the application's budget on each processor is initially set to 900 (for a 45 percent reservation). We start by finding a schedule for the application using these default processing settings. Once we have obtain such a schedule, we change the replenishment period of the a processor, and determine the minimum budget the WLAN application needs on the processor whose replenishment period was changed such that it can still meet its timing requirements.

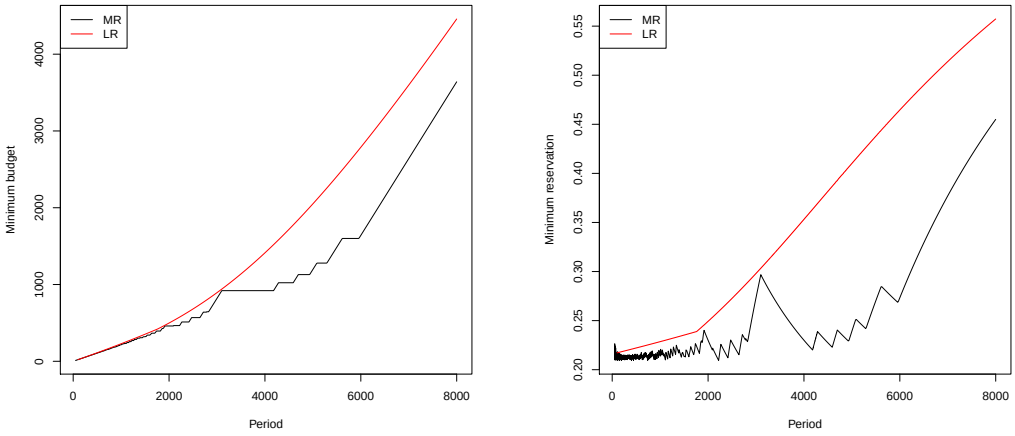
Figure 7.3 and figure 7.4 depict the found relationship between processor replenishment period and minimum required budget on respectively the ARM and EVP processor according to both the multi-rate and latency-rate response models. Observe that the figures illustrate that the latency-rate model is more conservative than the multi-rate model, since the minimum budget according to the latency-rate model is an upper bound for the multi-rate model. Whereas the period-budget relation depicted by the latency-rate model is a smooth curve, the multi-rate model depicts a step pattern in figure 7.3a. This step pattern is caused by the fact that an increase of processor period does not mean that we always have to increase our budget, since the current minimum budget is based on the worst-case behaviour of a task, and may therefore contain a certain amount of slack time. This points can be illustrated by imagining a TDM processor with replenishment period 9 and a task with computation time 2 that has a maximum allowed response time of 10. The minimum budget to ensure the task can meet its maximum response time is 2, which yields a worst-case response time of 9. However, increasing the period to 10 yields a worst-case response time of 10, still within the required response time.

If we wanted to determine a processor period that allows for the minimum amount of resource reservation on a processor, then figure 7.3b and 7.4b may give the impression that choosing an as small as possible period may be a good choice. However, this ignores the fact that a smaller period also increases the relative amount of time a processor spends on context switching. Figure 7.5 modifies the minimum reservation plot for a period to include the relative time spend on context switches. This context switch time is based on the assumption that the WLAN application has to share the processors with one other application, and that the processors have a worst-case maximum context switch time of 100ns. Figure 7.5a and figure 7.5b illustrate that many of the smaller period values are not an optimal choice as replenishment period.



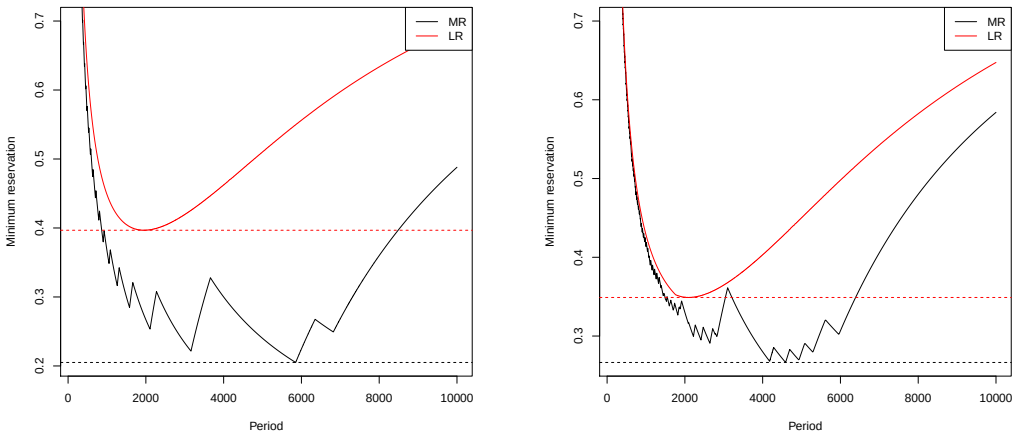
(a) Period vs minimum budget (b) Period vs minimum reservation

Figure 7.3: Effects of the choice of replenishment period of an ARM processor on the minimum resource requirements of a WLAN transceiver application.



(a) Period vs minimum budget (b) Period vs minimum reservation

Figure 7.4: Effects of the choice of replenishment period of an EVP processor on the minimum resource requirements of a WLAN transceiver application.



(a) ARM (b) EVP

Figure 7.5: Effects of the choice of replenishment period on the minimum processor reservation of on ARM and EVP processor, including effects of context switching.

7.3 Scheduling strategy

We have performed a number of experiments to determine the compare the results obtained by using our proposed PBS response time model and scheduling strategy to the results obtained by using a state-of-the-art response model and scheduling strategy for TDM arbitration. For this experiment we have used two actual applications from the software-defined radio domain: An implementation of a WLAN and a TDS-CSDM transceiver. We want to derive schedules and scheduler parameter such that we can run these two applications simultaneously in the same system.

We assume the system consists out of three processors; a general-purpose ARM processor, a EVP to handle detection, synchronization and demodulation tasks and an Software Codec processor (SWC) that handles baseband coding and decoding functionally. We assume each of these processor have a worst-case context switch time of 100 nanoseconds. We assume we would like to spend at most 10 percent of a processor's time on context switches.

The task graphs of both the WLAN and TDS-CDMA transceivers are depicted in figure 3.1. The number inside the actors represents the actor's worst-case computation time in nanoseconds, whereas the color coding denotes the different processors on which an actor must execute. Note that the green and gray colored notes do not represent actual task, but rather model respectively the source and timing constraints of the application. The WLAN application has a maximum production period of 40000 nanosecond, whereas the TDS-CDMA application has a maximum production period of 675000 nanoseconds. The WLAN transceiver thus clearly has considerably tighter timing requirements than

	EVP	ARM	SWC
WLAN	23.0	25.0	40.4
TDS-CDMA	20.4	25.0	7.5
CS	10.0	10.0	10.0
Total	53.4	60.0	57.9

(a) Reduction order EVP, ARM, SWC

	EVP	ARM	SWC
WLAN	28.3	16.7	42.8
TDS-CDMA	44.1	0.8	7.5
CS	10.0	10.0	10.0
Total	82.4	27.5	60.3

(b) Reduction order ARM, EVP, SWC

Table 7.1: Percentage of processor reservation required for running a WLAN and TDS-CDMA transceiver under TDM arbitration.

the TDS-CDMA transceiver.

The scheduling strategy for TDM arbitrations starts by determining the processor periods. It uses the reasoning that a lower replenishment period allows tasks to execute more often, and can therefore mitigate over-reservation due to tight latency requirements. It therefore sets each processor's period to the minimum replenishment period that does cause an application to spend more than 10 percent of its time context switching. Since we assumed each context switch takes at most 100ns and each application can cause one context switch per period under TDM arbitration, we set the replenishment period for each processor to 2000. The available initial budget on each processor is divided evenly between both applications; since we can spend up to 10 percent of the time context switching and want to run two application simultaneously, each application gets 45 percent of the processor's replenishment period as budget. After finding schedules for both the WLAN and TDS-CDMA application's, we try to reduce an application's reservation on each processor to the minimum value possible value. This reduction is done one processor at a time using a binary search, in which we reduce the processor we consider most important first. The resulting minimum percentage of minimum processor reservation under TDM arbitration is displayed in table 7.1. The results clearly illustrate that the order in which we reduce an application's processor reservation has a large impact on the eventually obtained minimum reservation per processor. This is not surprising, since when we minimize an application's budget on a processor we are essentially increasing the worst-case response times of task executions on the processor until the task execution can only barely be finished on time, leaving few slack time for task executions on other processors.

When we apply our PBS scheduling strategy the the first thing that is done is deciding which application should become the high priority application. Since the maximum production period of 40000ns of the WLAN application is much tighter than than the production period of the TDS-CDMA application the WLAN application will be set as high priority application. Since we currently have only one processor of each type in the system, after scheduling we will have found a WLAN application schedule that maps actors to processors such that the sum of task execution times assigned to the

	EVP	ARM	SWC
WLAN	15.1	8.4	5.0
TDS-CDMA	21.5	30.1	7.8
CS	6.0	1.5	2.5
Total	42.6	39.7	15.3

(a) Reduction order EVP, ARM, SWC

	EVP	ARM	SWC
WLAN	15.1	8.4	5.0
TDS-CDMA	41.5	0.9	7.5
CS	6.0	1.5	2.5
Total	62.6	10.8	15.1

(b) Reduction order ARM, EVP, SWC

Table 7.2: Percentage of processor reservation required for running a WLAN and TDS-CDMA transceiver under PBS arbitration.

EVP, ARM and SWC processor is respectively 6035, 3360 and 2000 nanosecond. We therefore know that we can meet the WLAN's timing requirements by setting each processor's replenishment period to 40000 and giving the application 6035, 3360 and 2000 nanoseconds of budget on the system's processors. Based on the common factors of the WLAN application's production period and mapped task computation time sum per processor we can calculate various combination of alternative period combination. However, when checking which of these processor period combinations and their implied budget are valid for the high priority application we end up with only the default period combination of 40000 on each processor. When determining the maximum number of context switches that can occur per period we calculate that based on a modified task graph that there can be at most 11, 2 and 4 tasks enabling on respectively the EVP, ARM and SWC while the high priority application is not enabled. This allows us to bound the maximum number of context switches to at most 24, 6 and 10 per period. Next we search for a schedule for the TDS-CDMA application, initially allowing the application a reservation of 45 percent of the processor's resources, thus setting its budget to 18000ns per period. Afterwards we reduce the TDS-CDMA application's reservation on each processor to the minimum value possible, starting with the processor whose reservation we want to reduce the most. The final minimum required reservation for both the WLAN and TDS-CDMA application - and the maximum time we will spend context switching between them - is given by table 7.2.

Table 7.3 quantifies the budget reservation improvement we obtain when using our proposed PBS model and scheduling strategy instead of the TDM scheduling strategy. As expected we see that the required reservation of the WLAN application can be decreased significantly under PBS when compared to TDM. Observe that the difference in minimum reservation increases when the processors importance decreases. This is again due to the fact that when we reduce processor reservations the first processor that we reduce will consume most of the slack time within an iteration. The minimum required reservation of the TDS-CDMA application is slightly increased on most processors under PBS arbitration, since its execution can suffer interference by the high priority application. A slightly remarkable result is that this is not the case for the EVP processor in table 7.3b, in which we can see that the application's minimum reservation actually slightly

	EVP	ARM	SWC
WLAN	7.9	16.6	35.4
TDS-CDMA	-1.1	-5.1	-0.3
CS	4.0	8.5	7.5
Total	10.8	20.0	42.6

(a) Reduction order EVP, ARM, SWC

	EVP	ARM	SWC
WLAN	13.2	8.3	37.8
TDS-CDMA	2.6	-0.1	-0.1
CS	4.0	8.5	7.5
Total	19.8	16.7	45.2

(b) Reduction order ARM, EVP, SWC

Table 7.3: Reservation improvement when using PBS arbitration instead of TDM arbitration.

improves. This is most likely another effect of the reservation minimization technique we used in combination with the significantly different replenishment period between the TDM and PBS approaches. Although the worst-case number of context switches per period under PBS arbitration has significantly increased for all processor (in case of the EVP from 2 to 24), the period increase from 2000ns to 40000ns more than compensates for this, such that the percentage of time we spend context switching is actually smaller under PBS than under TDM.

The time it took to run these experiment was considerable. A run of the scheduling strategy for the WLAN and TDS-CDMA application with full precision took approximately 30 days to complete in the current implementation of the analysis tool. The primary cause of this excessively long run time is the fact that we use the multi-rate model as our response model, which is known to potentially require very large single-rate temporal analysis models. We observed single temporal analysis steps that took more than a day to complete. Reducing the precision of the binary search step that minimizes the required processing reservation can substantially reduce the needed number of analysis steps that have to be performed, and therefore the required computation time. One such experiment with reduced precision allowed the scheduling strategy to complete within 20 days, while overestimating the minimum required reservation less than a half percent. However, due to time constraints we did not further explore the possible trade-off between analysis time and precision.

Our experiment proves that using PBS arbitration instead of TDM arbitration can be a valid option for certain applications. However, we should keep in mind that there are also cases in which PBS arbitration performs worse than TDM arbitration. For instance; experiments with running two WLAN applications under PBS arbitration failed because the high priority application causes too much interference such that the low priority application cannot meet its timing requirements, even when we set its budget on each processor to the largest value possible.

Chapter 8

Conclusions and further work

This chapter summarises the contributions discussed in this thesis, discusses their limitations, and contains suggestions for potential improvements and future work.

8.1 Contributions

We can summarise the contributions of this thesis as:

- We defined a generally applicable dataflow response model for capturing the worst-case temporal effects of PBS arbitration. The introduced response model does not make any restrictive assumptions and is most likely less conservative than the current state-of-the-art PBS response models.
- We formulated equations that can give the exact finish times of task executions under PBS arbitration.
- We defined an approach for bounding the worst-case number of context-switches under PBS arbitration. This bound is derived based on analysis of the task graph and schedule of the high priority application.
- We proposed a PBS scheduling strategy that given a set of applications and their timing requirements can find application schedules, scheduler parameters and the minimum required reservation of all applications.
- We conducted experiments with the proposed PBS response model and scheduling strategy. These experiments showed that for actual software-defined ratio applications PBS arbitration may allow a considerably smaller net resource reservation than TDM arbitration: Running WLAN and TDS-CDMA transceiver applications under PBS arbitration allows us to reduce the minimum required resource reservation on the highest weight processor to 40 percent of the required reservation under TDM arbitration, freeing 16 percent of the total processing time on the processor.

The reservation on lower weight processors can be reduced even more, where the largest reservation reduction requires only 26 percent of the resources needed to run the applications under TDM arbitration, freeing 45 percent of the processor's total processing time.

8.2 Limitations

We can identify several limitations that apply to this thesis's contributions:

- Our proposed response model is a multi-rate model with modified latency actor. Certain combination of application budget and task computation times can make the single-rate version of the multi-rate response model excessively large. The creation and analysis of such an excessively large dataflow model can take an considerable amount of time. The running time of our proposed scheduling strategy may therefore be unacceptably long, especially if there exist many valid replenishment period combinations, since we have to determine an application's minimum resource reservation for each period combination.

- The minimum reservation for a low priority application is determine using a binary search on the range of valid application budgets. An advantage of using this binary search is that we can find the budget that minimizes an application's reservation in a logarithmic number of steps. A problem of this approach is that it does not allow us to reduce an application's reservation in accordance with each processor's relative weight. It instead removes all of the application's slack time to reduce the reservation of a single processor, which may not leave any slack left to reduce reservation of other processors who may be equally important weight.

- PBS arbitration can yield improved net minimum reservation requirements in case the high priority application has significantly tighter timing requirements than the low priority applications. This improved reservation can be realised because setting the application with tightest latency requirements as high priority application allows us to remove all its over-reservation. This over-reservation tends be considerable for an application with tight timing requirements, such that the reservation reduction of the high priority application can more than compensate for the increased reservation requirements of the low priority applications. However, when we run a high and low application with roughly equivalent timing requirements or a large number of low priority applications, the reduced over-reservation of the high priority application may no longer compensate the increased required reservation of other applications.

- Comparing the amount of free processor reservation for PBS and TDM may not yield a totally fair comparison. Any additional application will most likely need more resource reservation when running as a low priority application on a PBS processor than processor is not reservation reductions of using PBS arbitration is

not totally fair since any new low priority application will potentially suffer interference of the high priority application, thereby increasing its required reservation.

8.3 Further work

Due to scope and time constraints there are several ideas and potential improvements that can serve as a stating point for future work:

8.3.1 Size reduction of multi-rate model

The proposed PBS response model is based on the multi-rate model [5]. The single-rate representation of the multi-rate model can be excessively large for some combination of task computation times and budget, severely increasing the time it can take to verify an application's timing requirements. How large the size expansion of the single-rate representation of the multi-rate model is depends on the greatest-common-divisor (GCD) between an application's budget and the task computation times. A small GCD can cause a very large increase of actors when we convert a multi-rate model to single-rate. We can limit the size expansion of single-rate conversion of a multi-rate model by guaranteeing that the GCD of budget and task computation times is at least some minimum value. We can guarantee this minimum sized GCD at the cost of increased conservativity by either increasing the modelled task computation times or decreasing the modelled budget to values that yield the desired GCD.

8.3.2 Improved upper bound on maximum number of context switches

We currently obtain an upper bound on the maximum number of context switches per processor by counting how many actors have incoming arcs from external actors. Although we try to take into account that certain incoming external arcs cannot cause additional context switches, there is a limit to how much we can reduce the over-estimation of the number of context switches without knowing more about the execution of an application's tasks. One way to improve our bound on the maximum number of context switches is if we know the best-case computation time of tasks.

Figure 8.1 depicts a task graph annotated with mapping and static ordering information. Actor A and B execute on processor $P1$ and actor C and D execute on processor $P2$. Since we do not know whether or not actor C is still executing while actor B finishes, we must assume that both actor C and actor D can cause a context switch to occur on processor $P2$. However, if we know the best-case computation times of actor C, we can potentially improve our bound: If the best case response time of actor C is larger than the worst-case response time of actor B, then we know that actor D cannot cause a

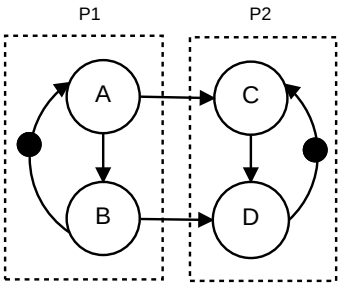


Figure 8.1: A task graph annotated with mapping and static order information.

context-switch, since it can only become enabled immediately after actor C has finished executing.

8.3.3 Scheduler switching during runtime

Although PBS arbitration can allow for lower resource reservation in certain scenario, a non priority based scheduler such as TDM may actual perform better in other scenarios. One way to ensure we can have the lowest reservation requirements in any scenario is by switching to the optimal resource arbitration mechanism during runtime. For instance; assume we are running an application with tight timing requirements under TDM arbitration. If we detect a request to start an application with tight timing requirements we first switch the arbitration type to PBS before starting the new application as a high priority application. The difficulty about switching processor arbitration during runtime is that we must be able to guarantee that each running application is still able to satisfy its timing requirements during and after the switch.

Bibliography

[1] O. M. Moreira, *Temporal Analysis and Scheduling of Hard Real-Time Radios on a Multi-processor*. PhD thesis, Eindhoven University of Technology, 2011.

[2] M. J. G. Bekooij, R. Hoes, O. M. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. D. Mol, S. Stuijk, V. Gheorghita, and J. Meerbergen, "Dataflow analysis for real-time embedded multiprocessor system design," in *Dynamic and Robust Streaming in and between connected consumer electronic devices*, vol. 3, pp. 81–108, Springer, 2005.

[3] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, SCOPES '07, pp. 11–22, ACM, 2007.

[4] A. Lele, "Data-flow based temporal analysis for TDM arbitration," Master's thesis, Eindhoven University of Technology, 2011.

[5] K. R. Butala, "Improved static data-flow model for TDM scheduler," Master's thesis, Delft University of Technology, 2012.

[6] M. Steine, M. J. G. Bekooij, and M. Wiggers, "A priority-based budget scheduler with conservative dataflow model," in *Proceedings of the 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, DSD '09, pp. 37–44, IEEE Computer Society, 2009.

[7] J. Staschulat and M. J. Bekooij, "Dataflow models for shared memory access latency analysis," in *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pp. 275–284, 2009.

[8] J. Cai, "Budget-scheduling for a real-time software-defined multi-radio on a heterogeneous multiprocessor system," Master's thesis, Eindhoven University of Technology, 2010.

[9] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, 1987.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static data flow," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 5, pp. 3255 –3258 vol.5, 1995.

[11] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York, NY, USA: Marcel Dekker, Inc., 1st ed., 2000.

[12] O. M. Moreira and M. J. G. Bekooij, "Self-timed scheduling analysis for real-time applications," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, no. 1, p. 083710, 2007.

[13] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi, "Throughput analysis of synchronous data flow graphs," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp. 25 –36, june 2006.

[14] H. T. Hsu, "An algorithm for finding a minimal equivalent graph of a digraph," *Journal of the ACM*, vol. 22, no. 1, pp. 11–16, 1975.